NICTA

Linux, Locking and Lots of Processors

Peter Chubb

peter.chubb@nicta.com.au

Australian Government
Department of Broadband, Communications and the Digital Economy
Australian Research Council

NICTA Funding and Supporting Members and Partners

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
- Linux Torvalds 1991

The history of UNIX-like operating systems is a history of people being dissatisfied with what they have and wanting to do something better. It started when Ken Thompson got bored with MULTICS and wanted to write a computer game (Space Travel). He found a disused PDP-7, and wrote an interactive operating system to run his game. The main contribution at this point was the simple file-system abstraction.

Other people found it interesting enough to want to port it to other systems, which led to the first major rewrite — from assembly to C. In some ways UNIX was the first successfully portable OS.

After Ritchie & Thompson (1974) was published, AT&T became aware of a growing market for UNIX. They wanted to discourage it: it was common for AT&T salesmen to say, 'Here's what you get: A whole lot of tapes, and an invoice for $10 000'. Fortunately educational licences were (almost) free, and universities around the world took up UNIX as the basis for teaching and research.

The University of California at Berkeley was one of those univer-

sities. In 1977, Bill Joy (a postgrad) put together and released the first Berkeley Software Distribution — in this instance, the main additions were a pascal compiler and Bill Joy's `ex` editor. Later BSDs contained contributed code from other universities, including UNSW. The BSD tapes were freely shared between source licensees of AT&T's UNIX.

John Lions and Ken Robinson read Ritchie & Thompson (1974), and decided to try to use UNIX as a teaching tool here. Ken sent off for the tapes, the department put them on a PDP-11, and started exploring. The license that came with the tapes allowed disclosure of the source code for 'Education and Research' — so John started his famous OS course, which involved reading and commenting on the Edition 6 source code.

In 1979, AT&T changed their source licence (it's conjectured, in response to the popularity of the Lions book), and future AT&T

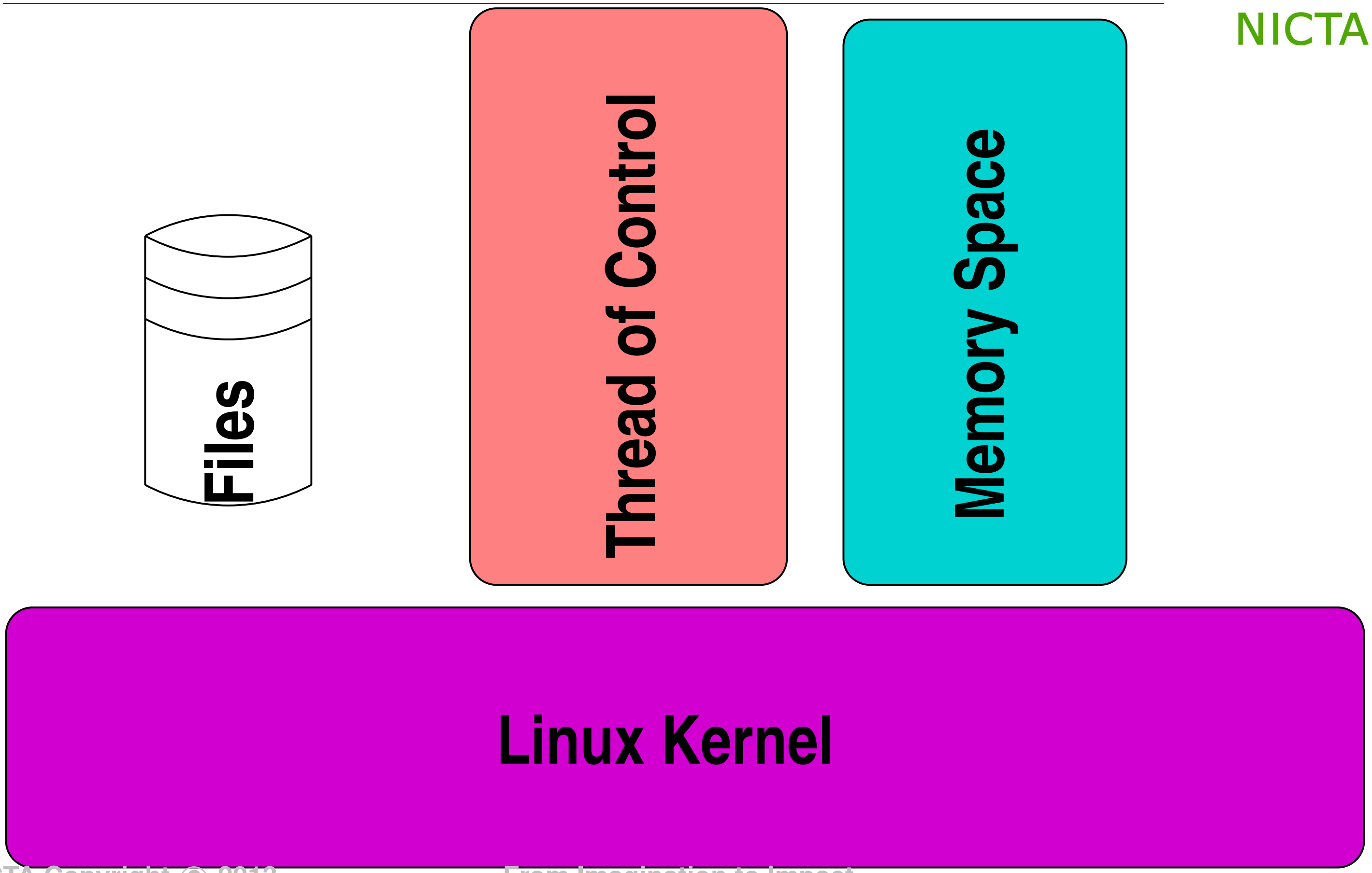
licensees were not able to use the book legally any more. UNSW obtained an exemption of some sort; but the upshot was that the Lions book was copied and copied and studied around the world. However, the licence change also meant that an alternative was needed for OS courses.
Many universities stopped teaching OS at any depth. One standout was Andy Tanenbaum's group in the Netherlands. He and his students wrote an OS called 'Minix' which was (almost) system call compatible with Edition 7 UNIX, and ran on readily available PC hardware. Minix gained popularity not only as a teaching tool but as a hobbyist almost 'open source' OS.
In 1991, Linus Torvalds decided to write his own OS — after all, how hard could it be? — to fix what he saw as some of the shortcomings of Minix. The rest is history.

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)

The UNIX core concepts have remained more-or-less the same since Ritchie and Thompson published their CACM paper. The process model and the file system model have remained the same. The IPC model (inherited from MERT, a different real-time OS being developed in Bell Labs in the 70s) also is the same. However there have been some significant additions.
The most important of these were Paged Virtual Memory (introduced when UNIX was ported to the VAX), which also introduced the idea of Memory-mapped files; TCP/IP networking, Graphical terminals, and multiprocessing, in all variants, master-slave, SMP and NUMA. Most of these improvements were from outside Bell Labs, and fed into AT&T's product via an open-source like patch-sharing.
In the late 80s the core interfaces were standardised by the IEEE, in the so-called POSIX standards.

ABSTRACTIONS



As in any POSIX operating system, the basic idea is to abstract away physical memory, processors and I/O devices (which can be arranged in arbitrarily complex topologies in a modern system), and provide threads, which are gathered into processes (a process is a group of threads sharing an address space and a few other resources), that access files (a file is something that can be read from or written to. Thus the file abstraction incorporates most devices). There are some other features provided: the OS tries to allocate resources according to some system-defined policies. It enforces security (processes in general cannot see each others' address spaces, and files have owners).

PROCESS MODEL

NICTA

- 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

The POSIX process model works by inheritance. At boot time, an initial process (process 1) is hand-crafted and set running. It then sets up the rest of the system in userspace.

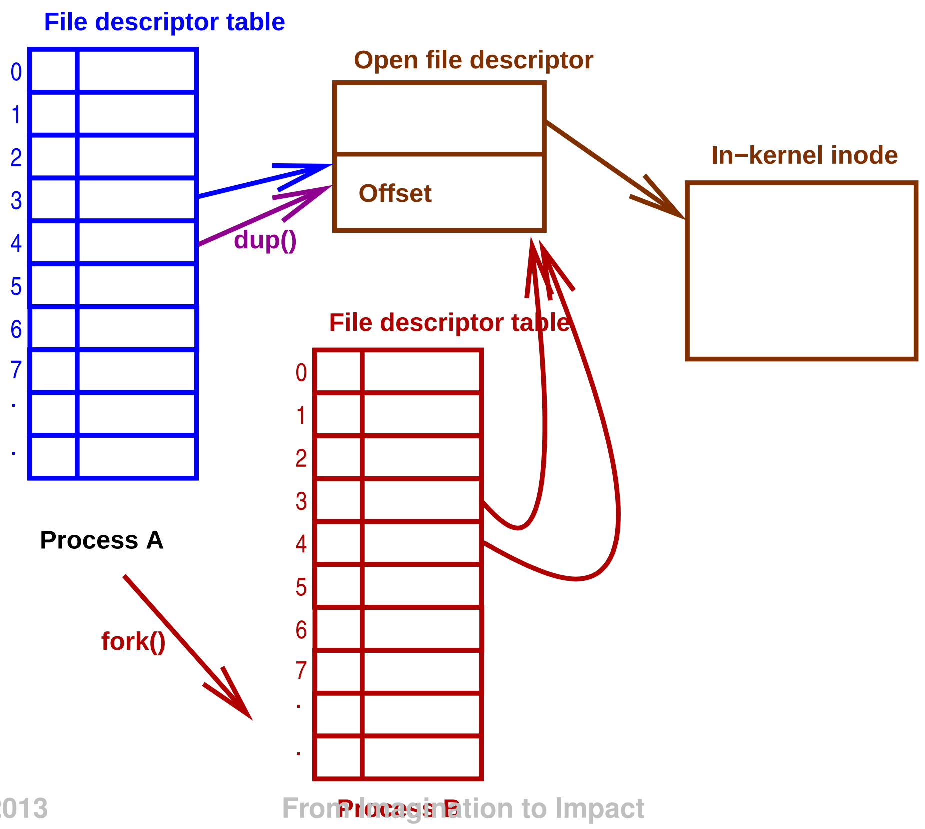
FORK() AND EXEC()

- 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

First I want to review the UNIX process model. Processes clone themselves by calling **`fork()`**. The only difference between the child and parent process after a **`fork()`** is the return value from **`fork()`** — it is zero in the child, and the value of the child's process ID in the parent. Most properties of the child are logical *copies* of the parent's; but open files and shared memory segments are *shared* between the child and the parent.
In particular, seek operations by either parent or child will affect and be seen by the other process.

NICTA

Files and Processes:



Each process has a file descriptor table. Logically this is an array indexed by a small integer. Each entry in the array contains a flag (the **`close-on-exec`** flag and a pointer to an entry in an *open file table*. (The actual data structures used are more complex than this, for performance and SMP locking).

When a process calls **`open()`**, the file descriptor table is scanned from 0, and the index of the next available entry is returned. The pointer is instantiated to point to an *open file descriptor* which in turn points to an in-kernel representation of an index node — an *inode* — which describes where on disc the bits of the file can be found, and where in the buffer cache can in memory bits be found. (Remember, this is only a logical view; the implementation is a lot more complex.)

A process can *duplicate* a file descriptor by calling **`dup()`** or **`dup2()`**. All **`dup`** does is find the lowest-numbered empty slot in

the file descriptor table, and copy its target into it. All file descriptors that are dups share the open file table entry, and so share the current position in the file for read and write.
When a process **`fork()`**s, its file descriptor table is copied. Thus it too shares its open file table entry with its parent.

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);
}
```

So a typical chunk of code to start a process looks something like this. **`fork()`** returns 0 in the child, and the process id of the child in the parent. The child process closes the three lowest-numbered file descriptors, then calls **`dup()`** to populate them again from the file descriptors for input and output. It then invokes **`execve()`**, one of a family of exec functions, to run *prog*. One could alternatively use **`dup2()`**, which says which target file descriptor to use, and closes it if it's in use. Be careful of the calls to **`close`** and **`dup`** as order is significant!
Some of the exec family functions do not pass the environment explicitly (**`envp`**); these cause the child to inherit a copy of the parent's environment.
Any file descriptors marked *close on exec* will be closed in the child after the **`exec`**; any others will be shared.

STANDARD FILE DESCRIPTORS

0	Standard Input
1	Standard Output
2	Standard Error

➜ Inherited from parent

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

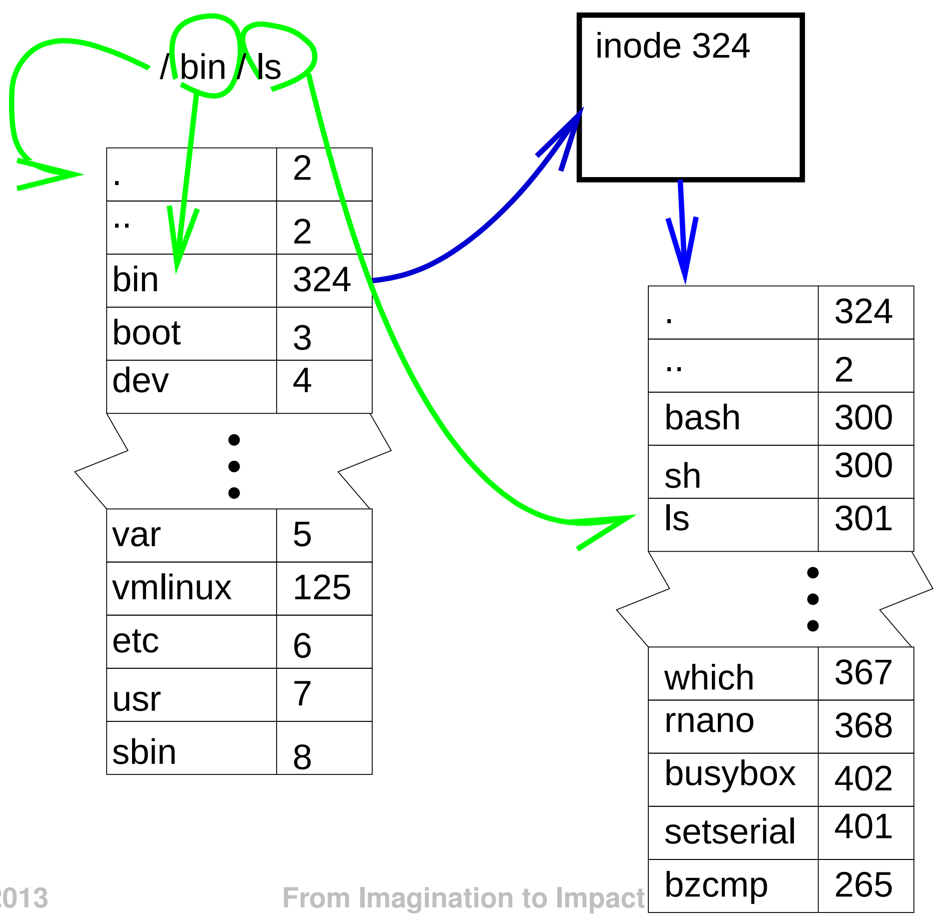
There are three file descriptors with conventional meanings. File descriptor 0 is the standard input file descriptor. Many command line utilities expect their input on file descriptor 0.
File descriptor 1 is the standard output. Almost all command line utilities output to file descriptor 1.
File descriptor 2 is the standard error output. Error messages are output on this descriptor so that they don't get mixed into the output stream. Almost all command line utilities, and many graphical utilities, write error messages to file descriptor 2.
As with all other file descriptors, these are inherited from the parent.
When you first log in, or when you start an X terminal, all three are set to point to the *controlling terminal* for the login shell.

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

The file model is very simple. In operating systems before UNIX, the OS was expected to understand the structure of all kinds of files: typically files were organised as fixed (or variable) length records with one or more indices into them. By contrast, UNIX regular files are just a stream of bytes.
Originally in UNIX directories were also just files, albeit with a structure understood by the kernel. To give more flexibility, they are now opaque to userspace, and managed by each individual filesystem.

FILE MODEL



The diagram shows how the kernel finds a file.
If it gets a file name that starts with a slash (/), it starts at the root of the directory hierarchy (otherwise it starts at the current process's current directory). The first link in the pathname is extracted (**`"bin"`**) by calling into the filesystem, and searched for in the root directory.
That yields an inode number, that can be used to find the contents of the directory. The next pathname component is then extracted from the name and looked up. In this case, that's the end, and inode 301 contains the metadata for **`"/bin/ls"`**.

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

Linux has many different filesystem types. Each has its own directory layout. Pathname lookup is abstracted in the Virtual FileSystem (VFS) layer. Traditionally, looking up the name to inode (**`namei`**) mapping has been slow; Linux currently uses a cache to speed up lookup.
At any point in the hierarchy a new filesystem can be grafted in using **`mount`**; **`namei()`** hides these boundaries from the rest of the system.
Symbolic links haven't been mentioned yet. A symbolic link is a special file that holds the name of another file. When the kernel encounters one in a search, it replaces the name it's parsing with the contents of the symbolic link.
Also, because of changes in the way that pathname lookups happen, there is no longer a function called `namei()`; however the files containing the path lookup are still called `namei.[ch]`.

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

This leads to a general principle: start with KISS

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()`

The kernel is written in C, but with a few extras. Code and data marked `__init` is used only during initialisation, either at boot time, or at module insertion time. After it has finished, it can be (and is) freed.
Code and data marked `__exit` is used only at module removal time. If it's for a built-in section, it can be discarded at link time. The build system checks for cross-section pointers and warns about them.
`__percpu` data is either unique to each processor, or replicated.
The kernel build systenm can do some fairly rudimentary static analysis to ensure that pointers passed from userspace are always checked before use, and that pointers into kernel space are not passed to user space. This relies on such pointers being declared with **`__user`** or **`__kernel`**. It can also check that variables that are intended as fixed shape bitwise entities are always

used that way—useful for bi-endian architectures like ARM.
Almost every agregate data structure, from lists through trees to page tables has a defined type-safe iterator.
And there's a new built-in, `container_of` that, given a type and a member, returns a typed pointer to its enclosing object.

C DIALECT

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

The kernel is written in a style that does not use **`#ifdef`** in C files. Instead, feature test constants are defined that evaluate to zero if the feature is not desired; the GCC optimiser will then eliminate any resulting dead code.

SCHEDULING

NICTA

Goals:

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

Because Linux runs on machines with up to 4096 processors, any scheduler must be scalable, and preferably O(1) in the number of runnable processes. It should also be 'fair' — by which I mean that processes with similar priority should get similar amounts of time, and no process should be starved. In addition, it should not load excessively a low-powered system with only a single processor (for example, in your wireless access point)
Because Linux is used by many for desktop/laptop use, it should give good interactivity, and respond 'snappily' to mouse/keyboard even if that compromises absolute throughput.
And finally, the scheduler should be aware of the caching. packaging and memory topology of the system, so it when it migrates tasks, it can keep them close to the memory they use, and also attempt to save power by keeping whole packages idle where possible.

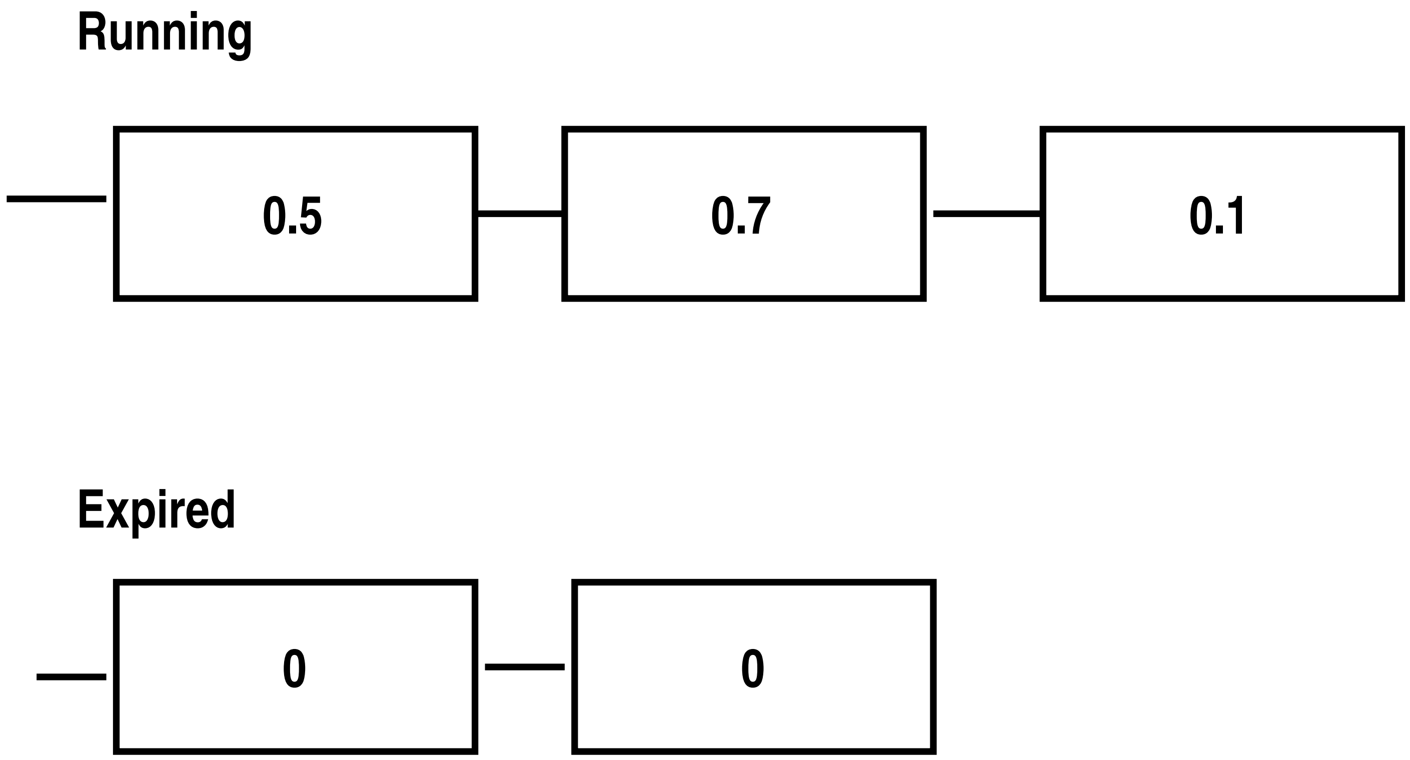
SCHEDULING

Implementation:

- Changes from time to time.
- Currently ‘CFS’ by Ingo Molnar.

Linux has had several different schedulers since it was first released. The first was a very simple scheduler similar to the MINIX scheduler. As Linux was deployed to larger, shared, systems it was found to have poor fairness, so a very simple dual-entitlement scheduler was created.

Dual Entitlement Scheduler



The idea here was that there were two queues: a deserving queue, and an undeserving queue. New and freshly woken processes were given a timeslice based on their 'nice' value. When a process's timeslice was all used up, it was moved to the 'undeserving' queue. When the 'deserving' queue was empty, a new timeslice was given to each runnable process, and the queues were swapped. (A very similar scheduler, but using a weight tree to disribute time slice, was used in Irix 6)
The main problem with this approach was that it was O(n) in the number of runnable and running processes—and on the big iron with 1024 processors, that was too slow. So it was replaced in the early 2.6 kernels with an O(1) scheduler, that was replaced in turn (when it gave poor interactive performance on small machines) with the current 'Completely Fair Scheduler'

SCHEDULING

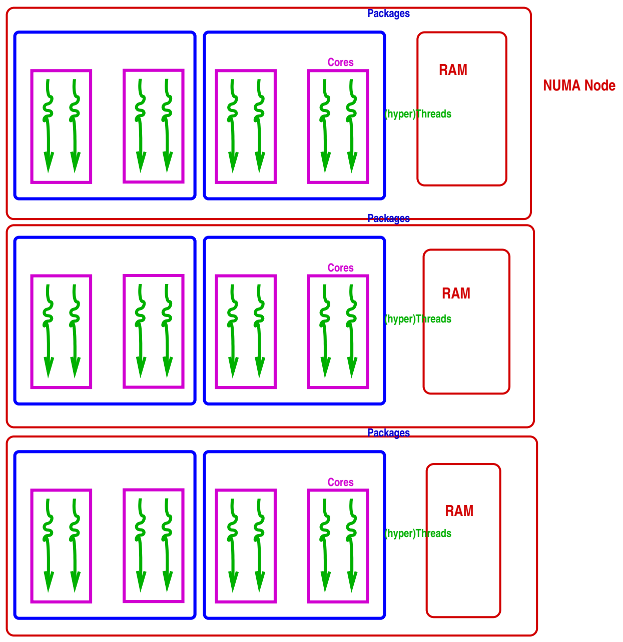
1. Keep tasks ordered by effective CPU runtime weighted by nice in red-black tree
2. Always run left-most task.

Devil's in the details:

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

The scheduler works by keeping track of run time for each task. Assuming all tasks are cpu bound and have equal priority, then all should run at the same rate. On a sufficiently parallel machine, they would always have equal runtime.
The scheduler keeps a period during which all runnable tasks should get a go on the processor — this period is by default 6ms scaled by the log of the number of available processors. Within a period, each task gets a time quantum weighted by its *nice*. However there is a minimum quantum; if the machine is overloaded, the period is stretched so that the minimum quantum is 0.75ms.
To avoid overflow, the scheduler tracks 'virtual runtime' instead of actual; virtual runtime is normalised to the number of running tasks. It is also adjusted regularly to avoid overflow.
Tasks are kept in vruntime order in a red-black tree. The leftmost

node then has the least vruntime so far; newly activated entities also go towards the left — short sleeps (less than one period) don't affect vruntime; but after awaking from a long sleep, the vruntime is set to the current minimum vruntime if that is greater than the task's current vruntime. Depending on how the scheduler has been configured, the new task will be scheduled either very soon, or at the end of the current period.



Your typical system has hardware threads as its bottom layer. These share functional units, and all cache levels. Hardware threads share a *core*, and there can be more than one core in a *package* or *socket*. Depending on the architecture, cores within a socket may share memory directly, or may be connected via separate memory buses to different regions of physical memory. Typically, separate sockets will connect to different regions of memory.

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
- Somehow identify cooperating threads, co-schedule on same package?

The rest of the complications in the scheduler are for hierarchical group-scheduling, and for coping with non-uniform processor topology.
I'm not going to go into group scheduling here (even though it's pretty neat), but its aim is to allow schedulable entities (at the lowest level, tasks or threads) to be gathered together into higher level entities according to credentials, or job, or whatever, and then schedule those entities against each other.
Locality, however, is really important. You'll recall that in a NUMA system, physical memory is spread so that some is local to any particular processor, and other memory is a long way off. To get good performance, you want as much as possible of a process's working set in local memory. Similarly, even in an SMP situation, if a process's working set is still (partly) in-cache it should be run on a processor that shares that cache.

Linux currently uses a ‘first touch’ policy: the first processor to write to a page causes the page to be allocated to its nearest memory. On **`fork()`**, the new process is allocated to the same node as its parent. **`exec()`** doesn’t change this (although there is an API to allow a process to migrate before calling **`exec()`**.
So how do processors other than the boot processor ever get to run anything?
The answer is in runqueue balancing.

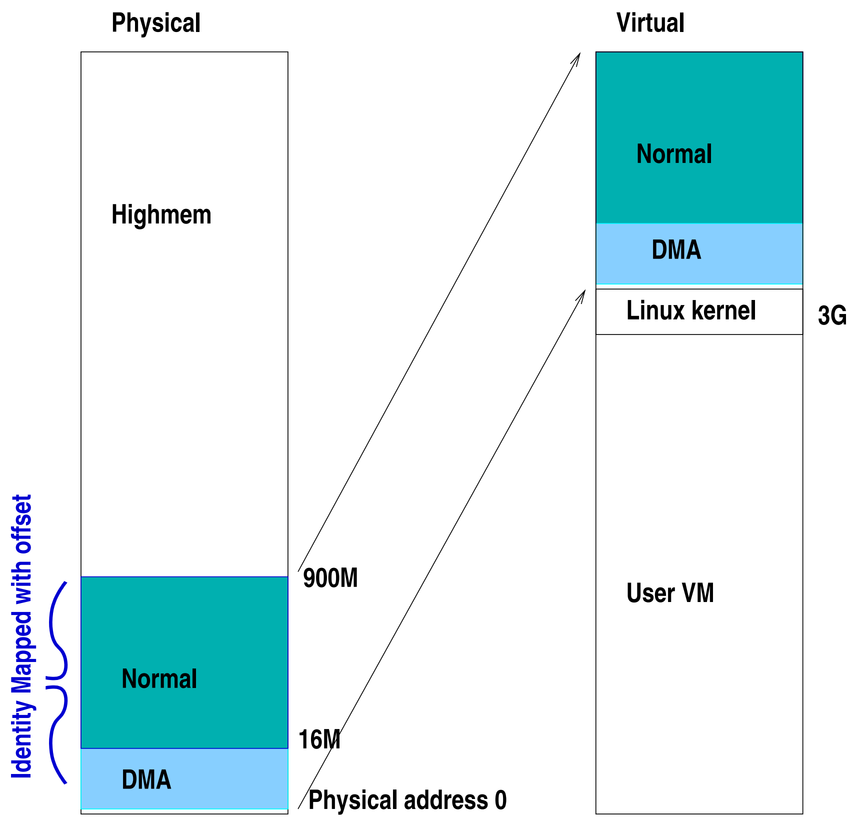
SCHEDULING

- 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)

There is one runqueue for each lowest schedulable entity (hyperthread or processor). These are grouped into ‘domains’. Each domain has its ‘load’ updated at regular intervals (where load is essentially sum of vruntime/number of processors).
One of the idle processors is nominated the ‘idle load balancer’. When a processor notices that rebalancing is needed (for example, because it is overloaded), it kicks the idle load balancer. The idle load balancer finds the busiest domains, and tries to move tasks around to fill up idle processors near the busiest domain. It needs more imbalance to move a task to a completely idle node than to a partly idle node.
Solving this problem perfectly is NP-hard — it’s equivalent to the bin-packing problem — but the heuristic approach seems to work well enough.

Memory Management

Memory in *zones*



Some of Linux's memory handling is to account for peculiarities in the PC architecture. To make things simple, as much memory as possible is mapped at a fixed offset, at least on X86-derived processors. Because of legacy devices that could only do DMA to the lowest 16M or memory, the lowest 16M are handled specially as **`ZONE_DMA`** — drivers for devices that need memory in that range can request it. (Some architectures have no physical memory in that range; either they have IOMMUs or they do not support such devices).
The linux kernel maps itself in, and has access to all of user virtual memory. In addition, as much physical memory as possible is mapped in with a simple offset. This allows easy access for in-kernel use of physical memory (e.g., for page tables or DMA buffers).
Any physical memory that cannot be mapped is termed 'High-

mem' and is mapped in on an ad-hoc basis. It is possible to compile the kernel with no 'Normal' memory, to allow all of the 4G 32-bit virtual address space to be allocated to userspace, but this comes with a performance hit.

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`

Direct mapped pages can be referred to by *logical addresses*; there are a simple pair of macros for converting between physical and logical addresses for these. Anything not mapped must be referred to by a **`struct page`** and an offset within the page. There is a **`struct page`** for every physical page (and for some things that aren't memory, such as MMIO regions). A **`struct page`** is less than 10 words (where a word is 64 bits on 64-bit architectures, and 32 bits on 32-bit architectures).

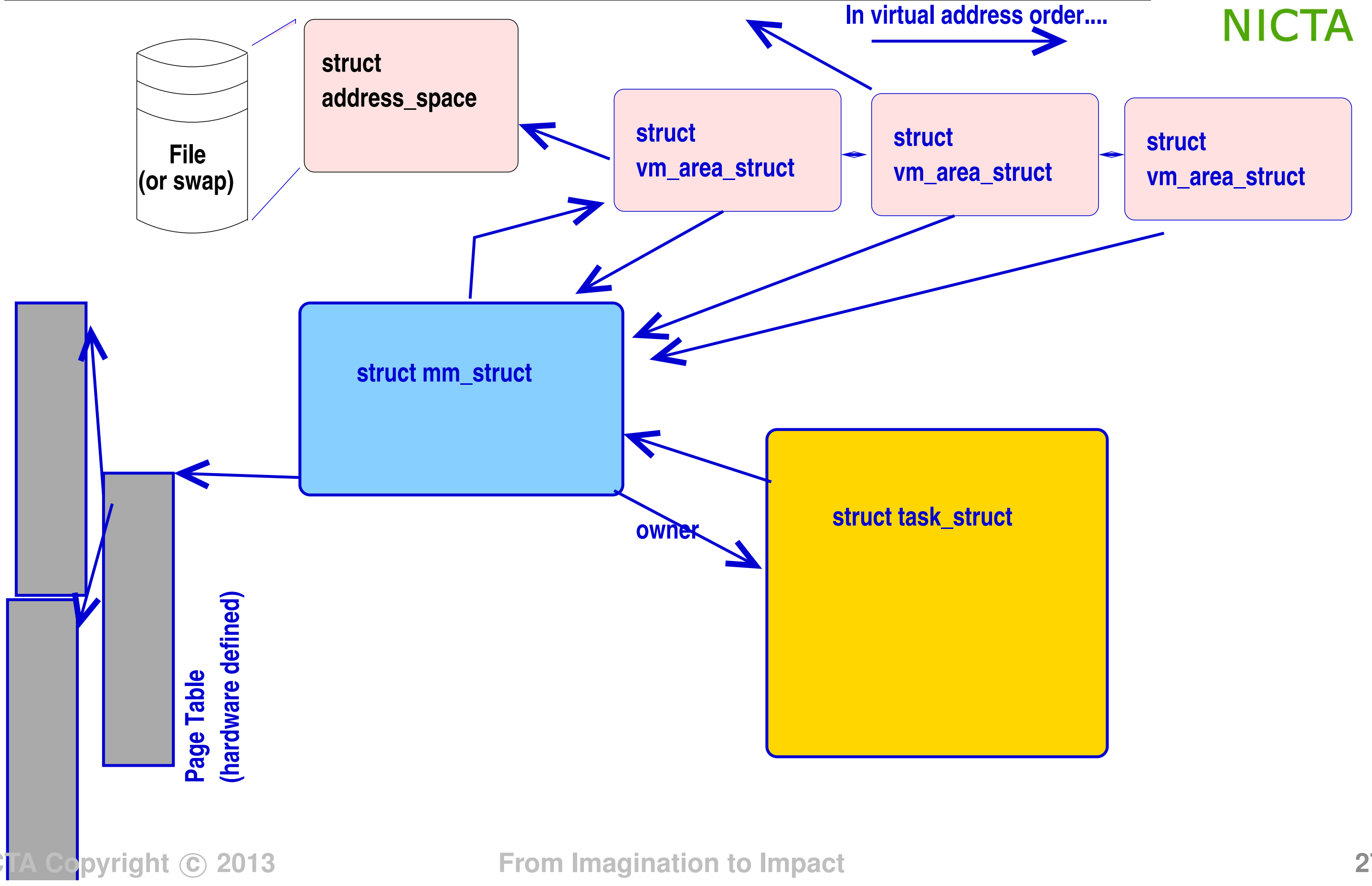
`struct page`:

- 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)

A **`struct page`** lives on one of several lists, and is in an array from which the physical address of the frame can be calculated. Because there has to be a **`struct page`** for every frame, there's considerable effort put into keeping them small. Without debugging options, for most architectures they will be 6 words long; with 4k pages and 64bit words that's a little over 1% of physical memory in this table.

A frame can be on a free list. If it is not, it will be in an active list, which is meant to give an approximation to LRU for the frames. The same pointers are overloaded for keeping track of compound frames (for SuperPages). Free lists are organised per memory domain on NUMA machines, using a buddy algorithm to merge pages into superpages as necessary.

Memory Management



Some of the structures for managing memory are shown in the slide. What's not visible here are the structure for managing swapping out, NUMA locality and superpages.
There is one **`task_struct`** for each thread of control. Each points to an **`mm_struct`** that describes the address space the thread runs in. Processes can be multi-threaded; one, the first to have been created, is the *thread group leader*, and is pointed to by the **`mm_struct`**. The **`struct mm_struct`** also has a pointer to the page table for this process (the shape of which is carefully abstracted out so that access to it is almost architecture-independent, but it always has to be a tree), a set of mappings held both in a red-black tree (for rapid access to the mapping for any address) and in a double linked list (for traversing the space).
Each *VMA* (*virtual memory area*, or **`struct vm_area_struct`**)

describes a contiguous mapped area of virtual memory, where each page within that area is backed (again contiguously) by the same object, and has the same permissions and flags. You could think of each **`mmap()`** call creating a new VMA. **`munmap()`** calls that split a mapping, or **`mprotect()`** calls that change part of a mapping can also create new VMAs.

MEMORY MANAGEMENT

NICTA

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

NICTA

migratepage Move pages between NUMA nodes

Others. . . And other housekeeping

Each VMA points into a **`struct address_space`** which represents a mappable object. An **`address_space`** also tracks which pages in the page cache belong to this object.
Most pages will either be backed by a file, or will be anonymous memory. Anonymous memory is either unbacked, or is backed by one of a number of swap areas.

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()`**

When a fault happens, the kernel has to work out whether this is a normal fault (where the page table entry just isn't instantiated yet) or is a userspace problem. Kernel faults are rare: they should occur only in a few special cases, and when accessing user virtual memory. They are handled specially.
It does this by first looking up the VMA in the red-black tree. If there's no VMA, then this is an unmapped area, and should generate a segmentation violation. If it's next to a stack segment, and the faulting address is at or near the current stack pointer, then the stack needs to be extended.
If it finds the VMA, then it checks that ther attempted operation is allowed — for example, writes to a read-only operation will cause a Segmentation Violation at this stage. If everything's OK, the code invokes **`handle_mm_fault()`** which walks the page table in an architecture-agnostic way, populating 'middle' directories

on the way to the leaf. Transparent superpages are also handled on the way down.
Finally **`handle_pte_fault()`** is called to handle the fault, now it's established that there really is a fault to handle.

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.

There are a number of different states the pte can be in. Each PTE holds flags that describe the state.
The simplest case is if the PTE is zero — it has only just been instantiated. In that case if the VMA has a fault handler, it is called via `do_linear_fault()` to instantiate the PTE. Otherwise an anonymous page is assigned to the PTE.
If this is an attempted write to a frame marked copy-on-write, a new anonymous page is allocated and copied to.
If the page is already present in the page cache, the PTE can just be reinstantiated – a 'minor' fault. Otherwise the VMA-specific fault handler reads the page first — a 'major' fault.
If this is the first write to an otherwise clean page, it's corresponding `struct page` is marked dirty, and a call is made into the writeback system — Linux tries to have no dirty page older than 30 seconds (tunable) in the cache.

DRIVER INTERFACE

Three kinds of device:

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

There are esentially three kinds of devices that can be attached to a computer system:

1. *platform devices* exist at known locations in the system's IO and memory address space, with well known interrupts. An example are the COM1 and COM2 ports on a PC.

2. Devices on a bus such as PCI or USB have unique identifiers that can be used at run-time to hook up a driver to the device. It is possible to enumerate all devices on the bus, and find out what's attached.

3. Devices on a bus such as i^2c or ISA have no standard way to query what they are.

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);
```

Each driver for a bus that identifies devices by some kind of ID declares a table of IDs of devices it can driver. You can also specify device IDs to bind against as a module parameter.

DRIVER INTERFACE

NICTA

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

All drivers have an initialisation function, that, even if it does nothing else, calls a *bus*_**`register_driver()`** function to tell the bus subsystem which devices this driver can manage, and to provide a vector of functions.
Most drivers also have an **`exit()`** function, that deregisters the driver.
When the bus is scanned (either at boot time, or in response to a hot-plug event), these tables are looked up, and the 'probe' routine for each driver that has registered interest is called.
The first whose probe is successful is bound to the device. You can see the bindings in **`/sys`**

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,
};
```

Platform devices are made to look like bus devices. Because there is no unique ID, the platform-specific initialisation code registers platform devices in a large table.
Here's an example, from the SLUG. Each platform device is described by a **`struct platform_device`** that contains at the least a name for the device, the number of 'resources' (IO or MMIO regions) and an array of those resources. The initialisation code calls **`platform_device_register()`** on each platform device. This registers against a dummy 'platform bus' using the name and ID.
The 8250 driver eventually calls **`serial8250_probe()`** which scans the platform bus claiming anything with the name 'serial8250'.

Driver Interface

non-enumerable buses: Treat like platform devices

At present, devices on non-enumerable buses are treated a bit like platform devices: at system initialisation time a table of the addresses where devices are expected to be is created; when the driver for the adapter for the bus is initialised, the bus addresses are probed.

Summary

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

MULTIPROCESSORS

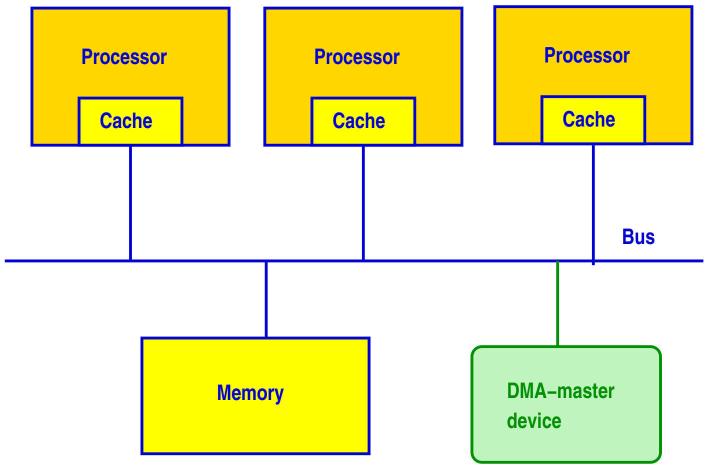
- ➜ Moore's law running out of steam
- ➜ So scale out instead of up.
- ➜ Works well only for some applications!

For a long time people have been attempting 'scale-out' instead of 'scale-up' solutions to lack of processing power. The problem is that for a uniprocessor, system speed increases (including I/O and memory bandwidth) are linear for a geometric increase in cost, so it's cheaper to buy two machines than to buy one more expensive machine with twice the actual performance. As a half-way point, multicore machines have become popular — especially as the limits to Moore's Law scalability are beginning to be felt.

MULTIPROCESSORS

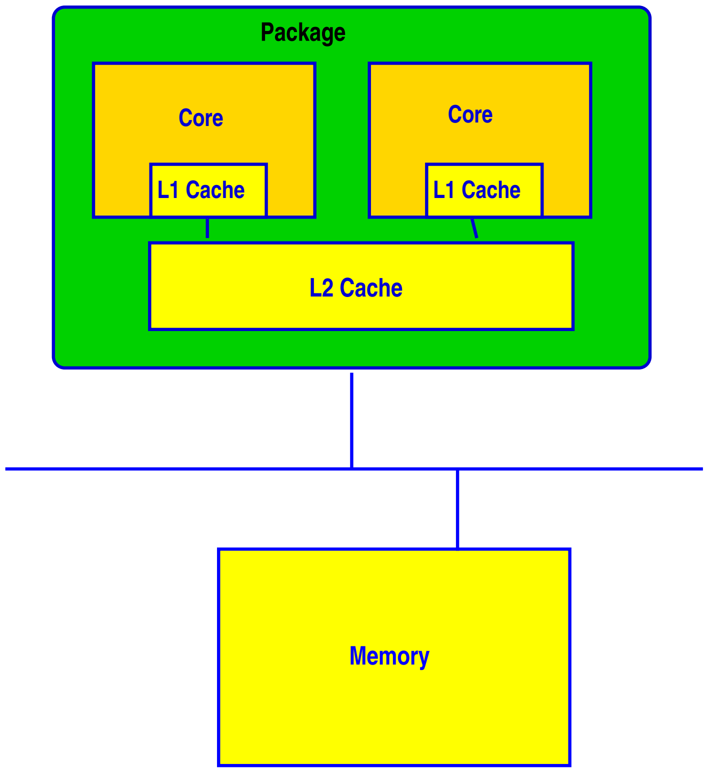
Classical symmetric Multiprocessor (SMP)

- Processors with local caches
- Connected by bus
- Separated cache hierarchy
- $\Rightarrow$ cache coherency issues



In the classical multiprocessor, each processor connects to shared main memory and I/O buses. Each processor has its own cache hierarchy. Typically, caches are *write-through*; each cache *snoops* the shared bus to invalidate its own cache entries.
There are also non-symmetric multiprocessor designs. In these, some of the processors are designated as having a special purpose. They may have different architectures (e.g., the IBM Cell processors) or may be the same (e.g., the early m68k multiprocessors, where to avoid problems in the architecture, one processor handled page faults, and the other(s) ran normal code).
In addition, the operating system can be structured to be asymmetric: one or a few cores can be dedicated to running OS code, the rest, user code.

MULTIPROCESSORS



Multicore (Chip Multiprocessor, CMP)

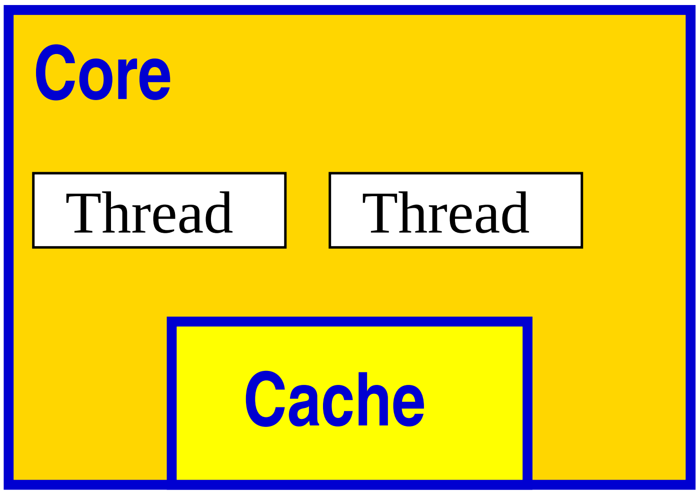
- per-core L1 caches
- Other caches shared
- Cache consistency addressed in h/w

It has become common for manufacturers to put more than one processor in a package. The exact cache levels that are shared varies from architecture to architecture; L1 cache is almost always shared; L2 sometimes, and L3 almost never, although the small number of cores per package mean that broadcast cache-coherency policies can be made to work.

MULTIPROCESSORS

Symmetric Multithreading (SMT)

- Multiple functional units
- Interleaved execution of several threads
- Fully shared cache hierarchy



Almost every modern architecture has multiple functional units. As instruction-level parallelism often isn't enough to keep all of the units busy, some architectures offer some kind of *symmetric multithreading*, (one variant of this is called *hyperthreading*). The main difference between architectures is whether threads are truly concurrent (x86 family), interleaved (Niagara) or switched according to event (Itanium switches on L3 cache miss). These don't make a lot of difference from the OS point of view, however.

MULTIPROCESSORS

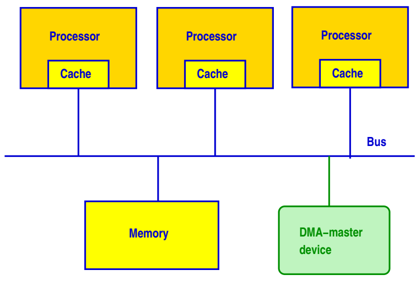
Cache Coherency:

Processor A writes a value to address x

then...

Processor B reads from address x

Does Processor B see the value Processor A wrote?



Whenever you have more than one cache for the same memory, there's the issue of coherency between those caches.

MULTIPROCESSORS

Snoopy caches:

- Each cache watches bus write traffic, and invalidates cache lines written to
- Requires *write-through* caches.

On a sufficiently small system, all processors see all bus operations. This obviously requires that write operations make it to the bus (or they would not be seen).

MULTIPROCESSORS

- Having to go to the bus every time is s l o w.
- Out-of-order execution on a single core becomes problematic on multiple cores
  - *barriers*

As the level of parallelism gets higher, broadcast traffic from every processor doing a write can clog the system, so instead, caches are *directory-based*.
In addition, although from a single-core perspective loads and stores happen in program order, when viewed from another core they can be out-of-order. More later...

CACHE COHERENCY

- There are many different coherency models used.
- We'll cover MESI only (four states).
  - Some consistency protocols have more states (up to ten!)
- 'memory bus' actually allows complex message passing between caches.

'Under the hood' that simple bus architecture is a complex message-passing system. Each cache line can be in one of a number of states; cache line states in different caches are coordinated by message passing.
This material is adapted from the book chapter by McKenney (2010).

CACHE COHERENCY

MESI:

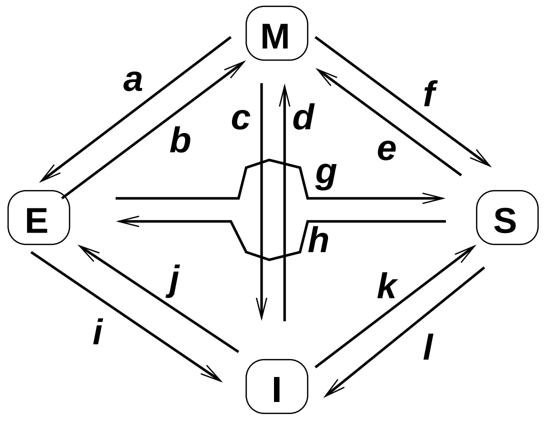
Each cache line is in one of four states:

M Modified

E Exclusive

S Shared

I Invalid



One commonly used protocol is the MESI protocol. Each cache line (containing more than one datum) is in one of four states. In the *Modified* state, the cache line is present only in the current cache, and it has been modified locally. In the *Exclusive* state, the cache line is present only in the current cache, but it has not been modified locally. In the *Shared* state, the data is read-only, and possibly present in other caches with the same values. The *Invalid* state means that the cache line is out-of-date, or doesn't match the 'real' value of the data, or similar. In any event, it is not to be used.

CACHE COHERENCY

MESI protocol messages:

Caches maintain consistency by passing messages:

Read

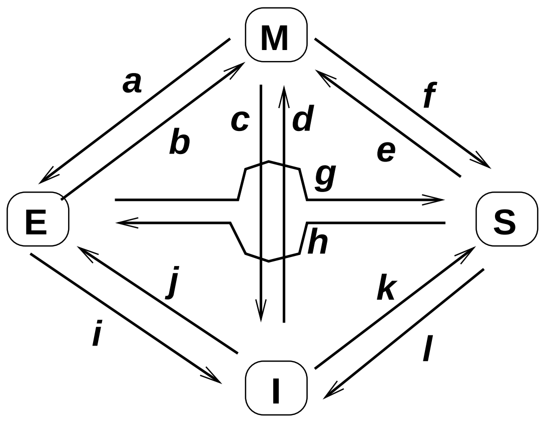
Read Response

Invalidate

Invalidate acknowledge

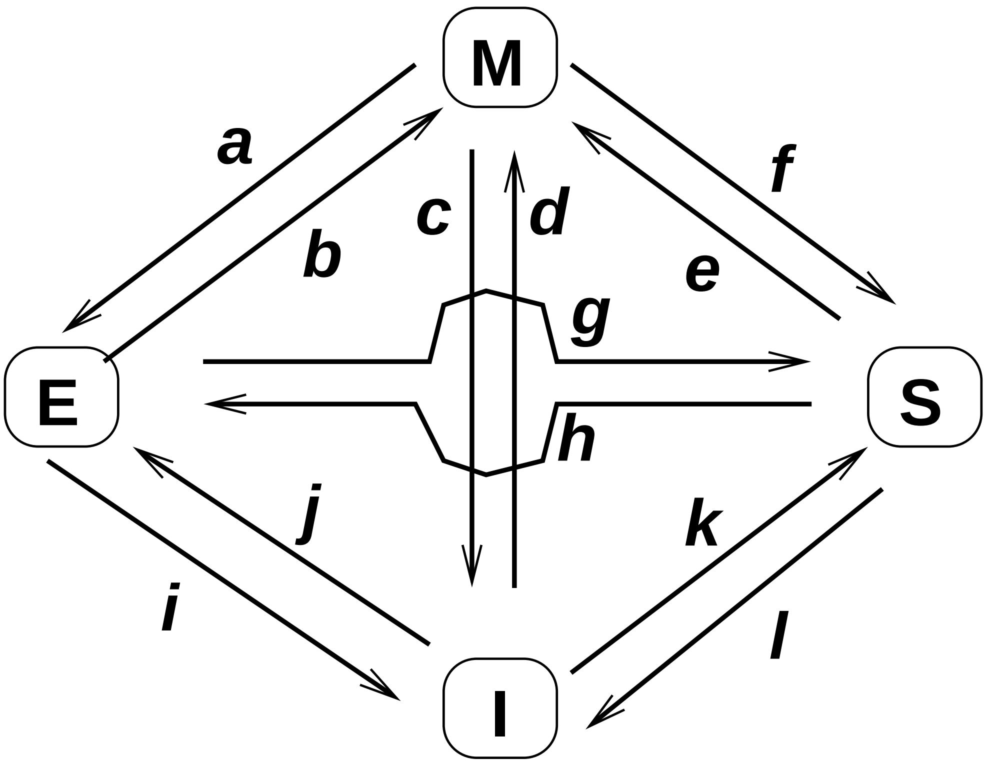
Read invalidate

Writeback



A *Read* message contains a physical address. A *read response* message contains the data held in that address: it can be provided either by main memory or by another processor's cache. An *invalidate* message contains a physical address. It says to mark the cache line containing that address as invalid. Each cache that contains the line must generate an *invalidate acknowledge* message; on small systems that do not use directory-based caching, *all* caches have to generate an *invalidate acknowledge*. *Read invalidate* combines both *read* and *invalidate* messages into one: presumably the processor is getting a cache line to write to it. It requires both *read response* and *Invalidate Acknowledge* messages in return. The *Writeback* message contains a physical address, and data to be written to that address in main memory. Other processors' caches may *snoop* the data on the way, too. This message allows caches to eject lines in the *M* state to make room for more data.

Cache Coherency



a $M \rightarrow E$ A cache line is written back to memory (*Writeback* message) but the processor maintains the right to modify the cacheline

b $E \rightarrow M$ The processor writes to a cache line it already had exclusive access to. No messages are needed.

c $M \rightarrow I$ The processor receives a *read invalidate* message for a cacheline it had modified. The processor must invalidate its local copy, then respond with both a *Read Response* and an *Invalidate Acknowledge* message.

d $I \rightarrow M$ The processor is doing an atomic operation (read-modify-write) on data not in its cache. It sends a *Read Invalidate* message; it can complete its transition when it has received a full set of *Invalidate acknowledge* responses.

e $S \to M$ The processor is doing an atomic operation on a data item that it had a read-only shared copy of in its cache. It cannot complete the state transition until all *Invalidate acknowledge* responses have been received.

f $M \to S$ Some other processor reads (with a *Read* message) the cache line, and it is supplied (with a *Read Response*) from this cache. The data may also be written to main memory.

g $E \to S$ Some other processor reads data from this cache line, and it is supplied either from this processor's cache

or from main memory. Either way, this cache retains a read-only copy.

h $S \to E$ There can be two causes for this transition: either all other processors have moved the cacheline to *Invalid* state, so this is the last copy; or this processor has decided it wants to write fairly soon to this cacheline, and has transmitted an *Invalidate* message. In the second case, it must wait for a full set of *Invalidate Acknowledge* responses before completing the transition.

i $E \to I$ Some other processor does an atomic operation on a datum held only by this processor's cache. The transition is initiated by a *Read Invalidate* message; this processor responds with both *Read Response* and *Invalidate Acknowledge*.

j $I \rightarrow E$ This processor is attempting to store to an address not currently cached. It will transmit a *Read Invalidate* message; and will complete the transition only when a *Read Response* and a full set of *Invalidate Acknowledge* messages have been received. The cacheline will usually move $E \rightarrow M$ soon afterwards, when the store actually happens.

k $I \rightarrow S$ This processor wants to get some data, It sends *Read* and receives *Read Response*

l $S \rightarrow I$ Some other processor is attempting to modify the cache line. An *Invalidate* message is received; an *Invalidate Acknowledge* is sent.

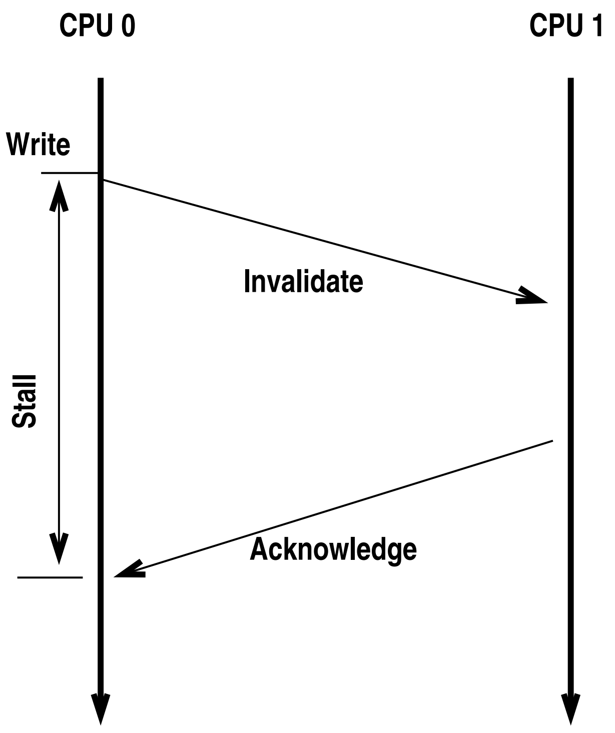
Cache Coherency

- Why don't *Invalidate Acknowledge* storms saturate interconnect?

⇒ simple bus doesn't scale; add *directory* to each cache that tracks who holds what cache line.

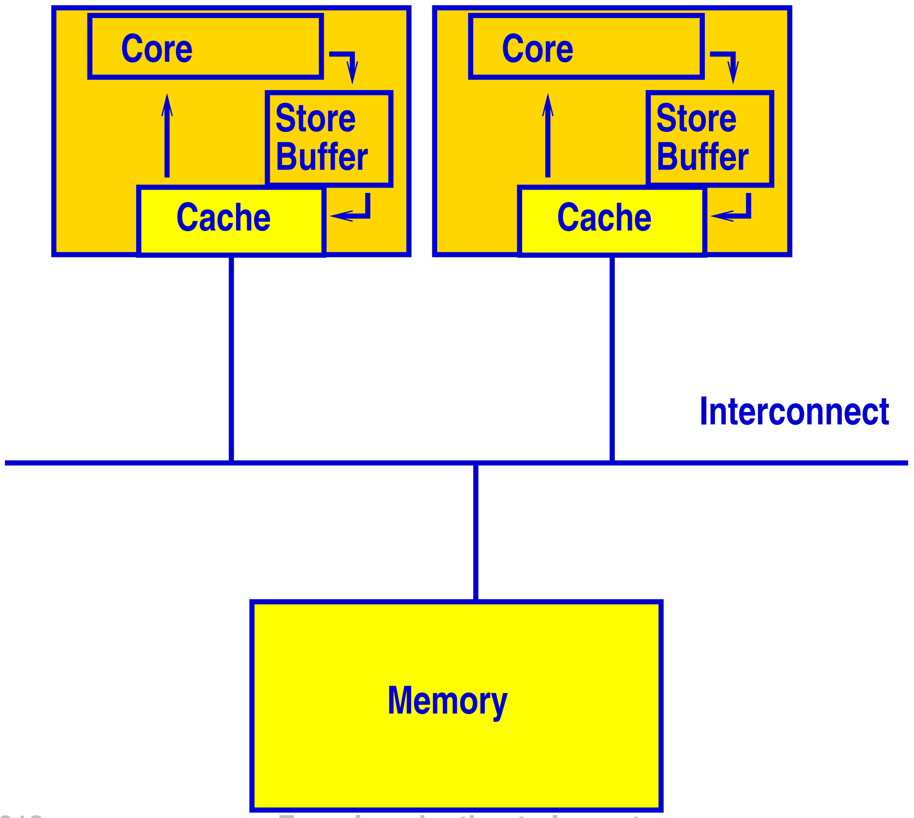
With all this bus traffic, cache line bouncing can be a major concern tying up the interconnect for relatively long periods of time. In addition, if a cache is busy, because of the necessity to wait for a remote transaction to complete, the current processor is stalled

Cache Coherency



While waiting for all the invalidate-acknowledgements, the processor can make no forward progress.

CACHE COHERENCY

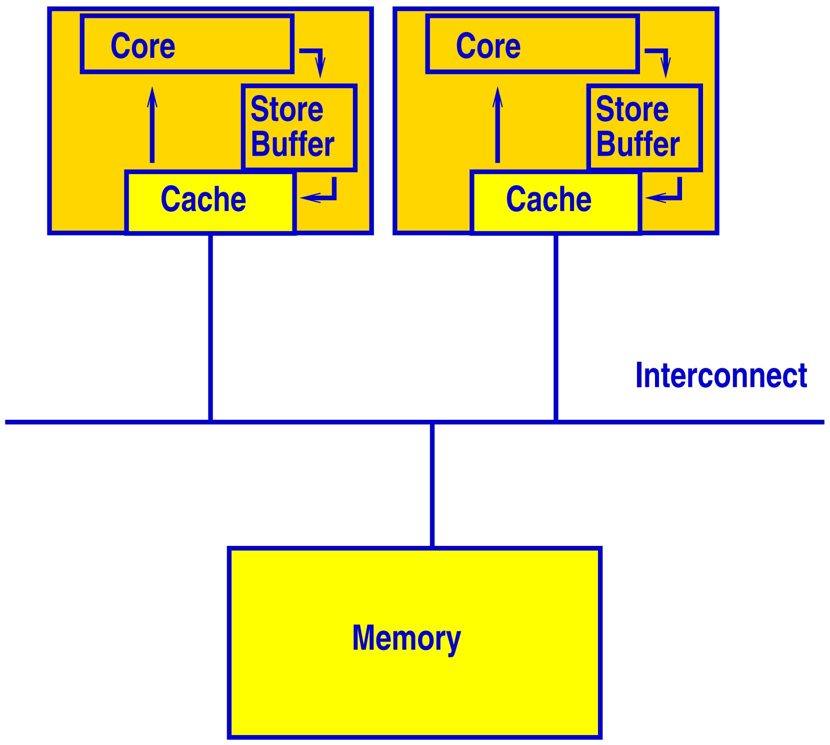


In most architectures this latency is hidden by queueing up stores in a *store buffer*. When the processor does a write to a cache-line in *Invalid* or *Shared* states, it sends a read-invalidate or a invalidate message, and then queues the write to its store buffer. It can then continue with its next operation without stalling.

CACHE COHERENCY

NICTA

Problems:



```
a = 1
b = a + 1
assert(b == 2)
```

If this is all it did, there would be problems. Imagine **a** is not in the current cache (state *Invalid*).

1. `a` not in cache, sends *Read Invalidate*
2. $a \leftarrow 1$ in store buffer
3. starts executing `b=a+1`, needs to read `a`
4. gets *Read Response* with `a==0`
5. loads `a` from cache
6. applies store from store buffer writing 1 to cache
7. adds one to the loaded value of a and stores it into `b`
8. Assertion now fails because `b` is 1.

CACHE COHERENCY

Solution is *Store Forwarding*

Processors snoop their store buffers as well as their caches on loads.

Local ops seen in program order

Insufficient in a Multiprocessor.

CACHE COHERENCY

CPU 0	**CPU 1**

`a = 1`
`b = 1`
	`while (b==0) continue;`
	`assert(a == 1);`

Start with `a` in CPU 1's cache; `b` in CPU 0.

Cache Coherency

1. CPU0: $a \leftarrow 1$. New value of `a` to store buffer, send *Read Invalidate*.
2. CPU1: reads `b`, sends *Read* message.
3. CPU0: executes $b \leftarrow 1$. It owns this cache line, so no messages sent.
4. CPU0 gets *Read*; sends value of `b` (now 1), marks it Shared.
5. CPU1 receives *Read Response*, breaks out of while loop, and the assertion fails.
6. CPU1 gets the *Read Invalidate* message and sends

Cache Coherency

the cache line containing `a` to CPU 0.

7. CPU0 finally gets the *Read Response*

The hardware cannot know about data-dependencies like these, and needs help from the programmer. Almost every MP-capable architecture has some form of *barrier* or *fence* instruction, that waits until anything in the store buffer is visible to all other processors, and also tweaks the out-of-order engine (if any) to control whether stores and loads before this instruction can be occur afterwards, and vice versa.

NICTA

Invalidate Queues:

- Invalidates take too long (busy caches, lots of other processors sending Invalidates)
- So buffer them:
  - Send Invalidate Acknowledge immediately
  - Guarantee not to send any other MESI message about this cache line until Invalidate completes.
- Can give *more* memory ordering problems McKenney (2010)

Busy caches can delay *Invalidate Acknowledge* messages for a long time. This latency is hidden using an 'Invalidate Queue'. A processor with an invalidate queue can send an *Invalidate Acknowledge* as soon as it receives the *Invalidate Request*. The *Invalidate Request* is queued instead of being actioned immediately.
Placing an Invalidate Request in the queue is a promise not to send any MESI protocol messages that relate to that cache line until the line has been invalidated.

Cache Coherency

Barriers: (also called *fences*)

Write barrier Waits for store buffer to drain

Read barrier Waits for Invalidate Queue to drain

Memory barrier Waits for both

All barriers also tweak out-of-order engine.

In addition to waiting for queues to drain, barriers tell the out-of-order execution engine (on OOO processors) not to move writes past a write barrier, or reads before a read barrier. This ensures in a critical section that instructions don't 'leak'.

CACHE COHERENCY

Guarantees:

1. Each processor sees its own memory accesses in program order
2. Processors may reorder writes only if they are to different memory locations
3. All of a processor's loads before a read barrier will be perceived by all processors to precede any load after that read barrier
4. Likewise for stores and a write barrier, or loads and stores and a full barrier

Particular processors may give stronger guarantees than these. In particular, X86 is fairly strongly ordered, and sometimes you can get away without a barrier that would be needed, say, on Alpha.

Cache Coherency

Another example:

```
  void foo() {        void bar(void) {
     a = 1;              while (!b)
     mb();                   ;
     b = 1;              assert (a == 1);
}                     }
```

Assume `a == 0` in *Shared* state; `b == 0` in *Exclusive* state in CPU 0; CPU0 does `foo()`, CPU1 does `bar()`

Cache Coherency

1. CPU0 puts $a \leftarrow 1$ into store buffer; sends *Invalidate*.
2. CPU 1 starts `while (!b)`; sends *Read* and stalls.
3. CPU 1 gets *Invalidate*, puts it into queue, and responds.
4. CPU 0 gets *Invalidate Acknowledge*, completes $a \leftarrow 1$, moves past barrier.
5. CPU 0 does $b \leftarrow 1$; no need for store buffer.
6. CPU 0 gets *Read* for `b`, sends *Read Response*, transition to *Shared*
7. CPU 1 gets *Read Response* with `b == 1`, breaks out of `while(...)`.

CACHE COHERENCY

8. CPU 1 reads `a` from cache, it's still 0 so assertion fails.
9. CPU 1 completes *Invalidate* on `a` Too late!

CACHE COHERENCY

Fix:

```
  void foo() {          void bar(void) {
    a = 1;                 while (!b)
    wmb();                       ;
    b = 1;                  rmb();
}                          assert (a == 1);
                        }
```

Adding a read barrier (or a full barrier) after the loop makes sure that the read of `a` cannot happen before the read of `b` — so (after a lot of coherency traffic) this will cause the assertion not to trigger.

CACHE COHERENCY

NICTA

DMA and I/O consistency:

- PCI MMIO writes are *posted* (i.e., queued) and can occur *out of order* WRT other I/Os.
- PCI I/O writes are *strongly ordered* and are effectively a barrier. wrt MMIO writes.
- Depending on architecture, CPU memory barriers may or may not be enough to serialise I/O reads/writes.
  - DMA not necessarily coherent with CPU cache: some bus-mastering devices need cache flushes.

We've talked a lot about memory consistency between processors – what about DMA? What of IPI?
Different I/O devices differ; some respect cache coherency, others do not. Read the docs!
In particular note that MMIO writes are 'posted' — they can be queued. The PCI spec says that any read from a device must force all writes to the device to complete.
Other buses have different memory-ordering issues. Be aware of them!
Some cases (from McKenney (2010)): a processor is very busy, and holds onto a cache line, so that when a device's DMA complete interrupt arrives, it still has old data in its cache.
Context switching also needs appropriate memory barriers, so if a thread is migrated from one processor to another, it sees its current data.

LOCKING AND ALL THAT. . .

- Shared data an issue
- How to maintain data consistency?
- Critical Sections
- Lock-free algorithms

In general to get good scalability, you need to avoid sharing data. However, sometimes shared data is essential.
Single word reads and writes are atomic, always; the visibility of such operations can be controlled with barriers as above.
When updating some data requires more than one cycle, there's the possibility of problems, when two processors try to update relate data items at the same time. For instance, consider a shared counter. If it has the value 2 to start with, and CPU0 wants to add 3 and CPU1 wants to add 1, the result could be 3, 5 or 6. Only 6 is the answer we want!
To solve this problem, we identify *critical sections* of code, and lock the data during those sections of code. However, locks have problems (in particular heavily contended locks cause much cache coherency traffic) and so for some common cases it's possible to use lock-free algorithms to ensure consistency.

Locking and all that. . .

Lock Granularity:

- Coarse grained: 'Big Kernel Lock'
  - Kernel becomes a *monitor* Hoare (1974)
  - At most one process/processor in kernel at a time
  - Limits kernel scalability
- Finer grained locking
  - Each group of related data has a lock
  - If carefully designed, can be efficient

Good discussion of trade-offs in ch.10Schimmel (1994),

The simplest locking is to treat all kernel data as shared, and use a single Big Kernel Lock to protect it. The Kernel then essentially becomes a Hoare Monitor.
The main problem with this is that it doesn't scale to very many processors. For systems that do not spend much time in the kernel, and do not have very many processors, this is a good solution though. It has also been used as a first step in converting from a uniprocessor OS to a multiprocessor OS.
The next step is generally identifying large chunks of data (e.g., the process table, the namei cache etc.) and providing a single lock for each.
As systems have grown, it's been noticed that finer-grain access to these things is desirable, and locks have become finer over time.

LOCKING AND ALL THAT. . .

Lock Data not Code!

The one thing to remember, is that we're locking data, not code. Be very clear over what data items need to be kept consistent, and insert appropriate locks into the code that manipulates that data.

Locking and all that. . .

Uniprocessor Considerations:

- Just need to protect against preëmption
- (and interrupt handlers)
  - disable/enable interrupts is sufficient
  - some processors have multiple interrupt levels
    `spl0...spl7`

On a uniprocessor, the only way for multiple threads of control to access the same data at a time is by a context switch, either to interrupt context, or to a different thread. As context switches happen only in two circumstances: voluntarily, or in response to an interrupt, it suffices to disable interrupts.
Some processors have multiple interrupt levels (e.g., M68K); operating systems for such architectures can selectively disable interrupts.

LOCKING AND ALL THAT...

Simple spin locks:

```
spinlock(volatile int *lp)
{
   while (test_and_set(*lp))
      ;
}
spinunlock(volatile int *lp)
{
   *lp = 0;
}
```

The simplest way to implement a spinlock, is to use an atomic test-and-set instruction sequence. On x86, this is usually a compare-exchange sequence; on m68k a `tas` instruction; ARM instead uses a send and receive event pair.
Such atomic operations usually have implicit memory barriers.

LOCKING AND ALL THAT. . .

NICTA

Issues with simple Spinlocks:

- Need to disable interrupts as well
- Wasted time spinning
- Too much cache coherency traffic
- Unfairness

Spinlocks have problems. To prevent context switching, you need to disable interrupts as well as hold the spinlock. While spinning, the processor uses power and wastes CPU cycles. Evey test-and-set instruction sends a *Read Invalidate* message and gets all the *Invalidate Acknowledge* results if there's any contention at all. And if there's a code sequence where a processor releases the lock, then reacquires it, it's likely not to let any other processor in (because the lock is locally cache-hot).

LOCKING AND ALL THAT...

NICTA

Ameliorating the problem:

- Spin on read Segall & Rudolph (1984)

```
    while (*lp || test_and_set(*lp))
      ;
```

  - Most processors spin on `while(*lp)`
  - The `test_and_set()` invalidates all locks, causes coherency traffic.
  - Better than plain `test_and_set()` but not perfect.
  - Unfairness still an issue

Segall & Rudolph (1984) suggested guarding the test-and-set with a read. That way, the spinning processors all have the cache-line in *Shared* state, and spin on their own local caches until the lock is released. This reduces coherency traffic, although there'll still be a thundering horde at unlock time. And the resulting lock is still unfair.

LOCKING AND ALL THAT. . .

- Ticket locks Anderson (1990), Corbet (2008)
  - Queue of waiting processes,
  - Always know which one to let in next — *Deterministic latencies run to run*
  - As implemented in Linux, coherency traffic still an issue.

If you've ever bought Cheese at David Jones, you'll be familiar with a ticket lock: each person takes a ticket with a number on it; when the global counter (controlled by the server) matches the number on your ticket, you get served. Nick Piggin put these into the Linux kernel in 2008; his contribution was to put the ticket and the lock into a single word so that an optimal code sequence could be written to take the lock and wait for it.

LOCKING AND ALL THAT. . .

```
lock(lp)
{
   x = atomic_inc(lp->next);
   while (lp->current != x)
      ;
}

unlock(lp)
{
    lp->current++;
}
```

Logically, the code sequence is as in the slide (I've omitted the necessary read barriers). It's assumed that the hardware provides a way to atomically increment a variable, do a write memory barrier and return its old value. If the lock was heavily contested it would make sense to have the ticket counter and the current ticket in separate cache lines, but this makes the lock larger.

Locking and all that...

- Multi-reader locks
  - Count readers, allow more than one
  - Single writer, blocks readers
- BRlocks

There are other locks used. There are many data structures that are read-mostly. Concurrent reads can allow forward progress for all readers. Multi-reader locks are slightly more expensive to implement than simple spin locks or ticket locks, but can provide good scalability enhancements.
Another lock used is the so-called brlock, or local-global lock. These use per-cpu spinlocks for read access. A process attempting write access needs to acquire all the per-cpu spinlocks. This can be *very* slow.

LOCKING AND ALL THAT. . .

NICTA

Other locking issues:

- Deadlocking
- Convoying
- Priority inversion
- Composibility issues

All locks have other issues. Lock ordering is important: if two processes each have to take two locks, and they do not take them in the same order, then they can deadlock.
Because locks serialise access to a data structure, they can cause processes to start proceeding in lock step — rather like cars after they're released from traffic lights.
It's possible for a process to be blocked trying to get a lock held by a lower priority process. This is usually only a problem with sleeping locks — while holding a spinlock, a process is always running. There have been many solutions proposed for this (usually involving some way to boost the priority of the process holding the lock) but none are entirely satisfactory.
Finally there are issues with operation composability: how do you operate on several data structures at once, when each is protected by a different lock? The simplest way is always to take

all the locks, but to watch for deadlocks.

LOCKING AND ALL THAT...

Sleeping locks:

semaphore Dijkstra (1965)

mutex just a binary semaphore

adaptive spinlock

In 1965, Edsgar Dijkstra proposed a way for processes to co-operate using a *semaphore*. The idea is that there is a variable which has a value. Any attempt to reduce the value below zero causes the process making the attempt to sleep until the value is positive. Any attempt to increase the value when there are sleeping processes associated with the semaphore wakes one of them.
A semaphore can thus be used as a counter (of number of items in a buffer for example), or if its values are restricted to zero or one, it can be used as a mutual-exclusion lock (mutex).
Many systems provide *adaptive spinlocks* — these spin for a short while, then if the lock has not been acquired the process is put to sleep.

LOCKING AND ALL THAT...

NICTA

Doing without locks: (See Fraser & Harris (2007) and McKenney (2003))

- Atomic operations
  - Differ from architecture to architecture
  - compare-exchange common
  - Usually have implicit memory barrier
  - Fetch-*op* useful in large multiprocessors: *op* carried out in interconnect.

Simple operations can be carried out lock free. Almost every architecture provides some number of atomic operations. Architectures that do not, can emulate them using spinlocks.
Some architectures may provide fetch-and-add or fetch-and-mul etc., instructions. These are implemented in the interconnect or in the L2 cache, and provide a way of atomically updating a variable and returning its previous or subsequent value.

LOCKING AND ALL THAT. . .

NICTA

Optimism:

- Generation counter updated on write
- Check before and after read/calc: if changed retry
- Also called *Sequence Locks*

If you can afford to waste a little time, then an optimistic lock can be used. The idea here is that a reading process grabs a generation counter before starting, and checks it just after finishing. If the generation numbers are the same, the data are consistent; otherwise the operation can be retried.
Concurrent writers still need to be serialised. Writers as their last operation update the generation count.

SCALABILITY

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

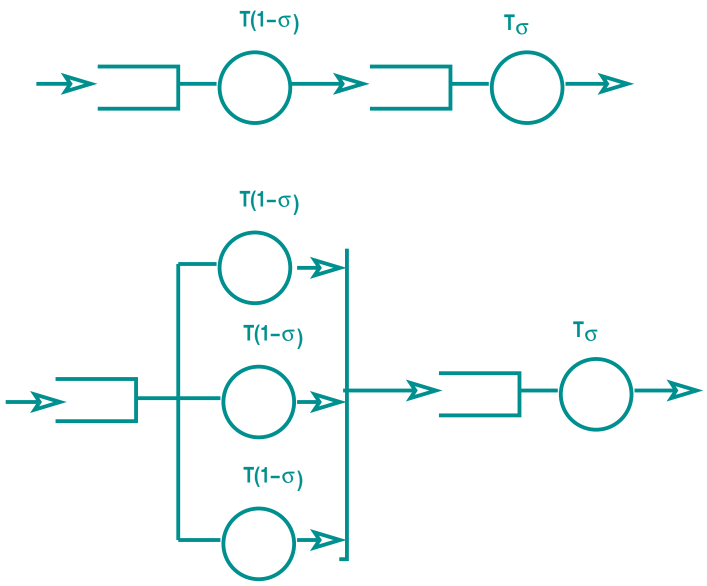
We've seen that because of locking and other issues, some portion of the multiprocessor's cycles are not available for useful work. In addition, some part of any workload is usually unavoidably serial.

SCALABILITY

Amdahl's law:

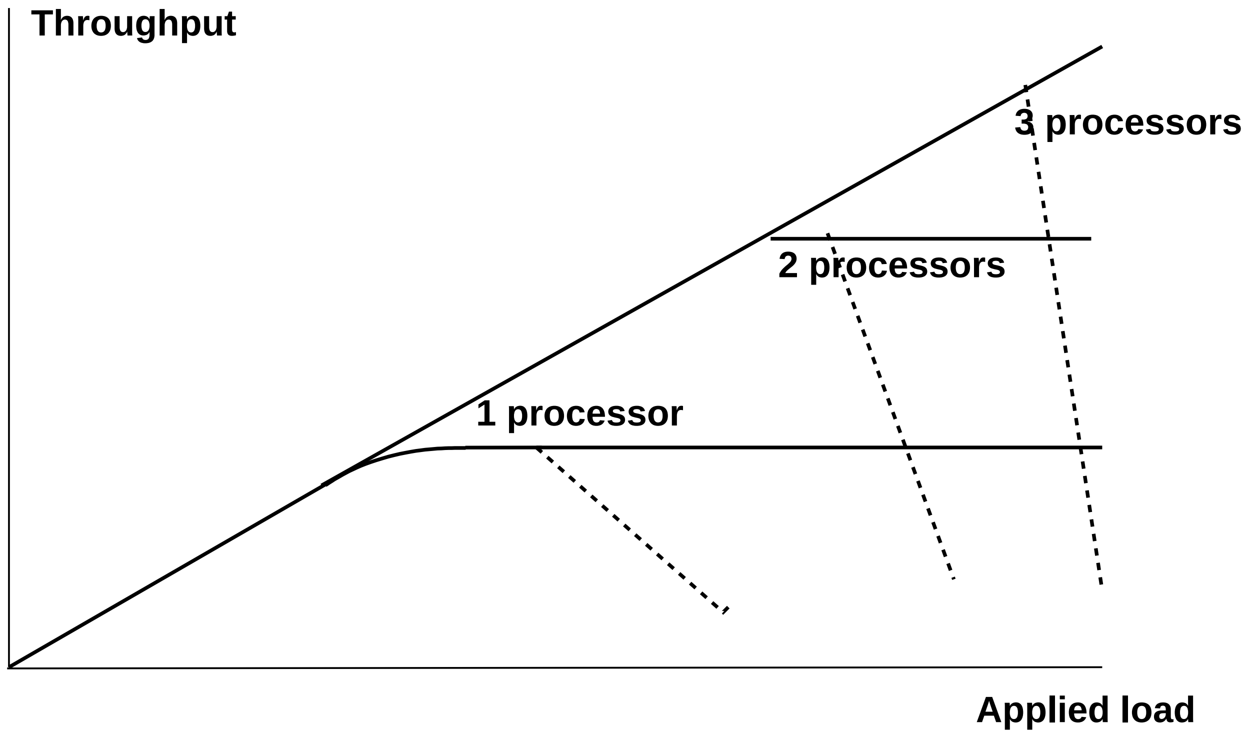
If a process can be split such that σ of the running time cannot be sped up, but the rest is sped up by running on p processors, then overall speedup is

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



It's fairly easy to derive Amdahl's law: perfect speedup for p processors would be p (running on two processors is twice as fast, takes half the time, than running on one processor).
The time taken for the workload to run on p processors if it took 1 unit of time on 1 processor is $\sigma + (1-\sigma)/p$. Speedup is then $1/(\sigma+(1-\sigma)/p)$ which, multiplying by p/p gives $p/(p\sigma+1-\sigma)$, or $p/(1+\sigma(p-1))$

SCALABILITY



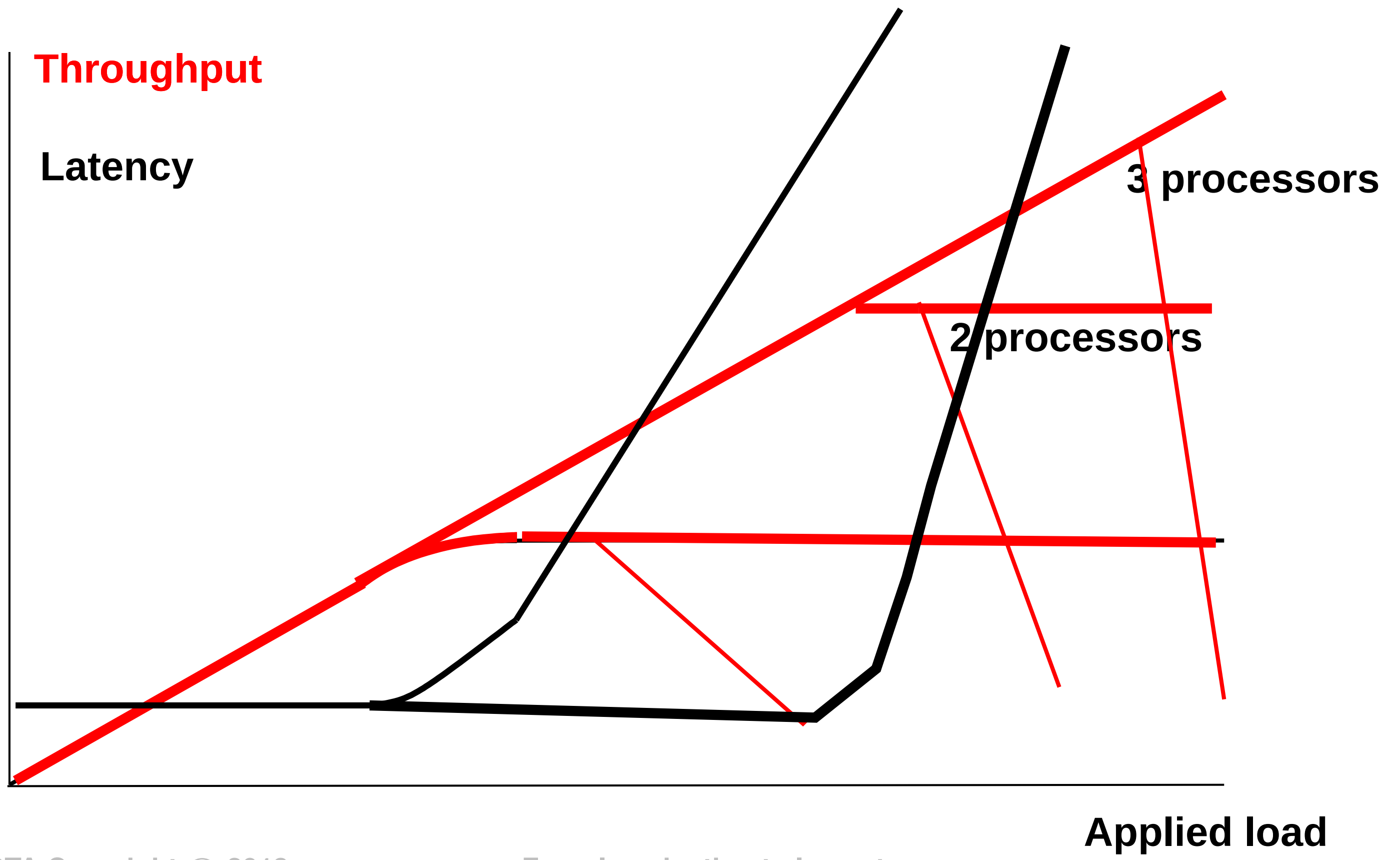
The general scalability curve looks something like the one in this slide. The Y-axis is throughput, the X-axis, applied load. Under low loads, where there is no bottleneck, throughput is determined solely by the load—each job is processed as it arrives, and the server is idle for some of the time. Latency for each job is the time to do the job.

As the load increases, the line starts to curve. At this point, some jobs are arriving before the previous one is finished: there is queueing in the system. Latency for each job is the time spent queued, plus the time to do the job.

When the system becomes overloaded, the curve flattens out. At this point, throughput is determined by the capacity of the system; average latency becomes infinite (because jobs cannot be processed as fast as they arrive, so the queue grows longer and longer), and the bottleneck resource is 100% utilised.

When you add more resources, you want the throughput to go up. Unfortunately, because of various effects we'll talk about later that doesn't always happen...

SCALABILITY



This graph shows the latency ‘hockey-stick’ curve. Latency is determined by service time in the left-hand flat part of the curve, and by service+queueing time in the upward sloping right-hand side.
When the system is totally overloaded, the average latency is infinite.

SCALABILITY

NICTA

Gunther's law:

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

where:

N is demand

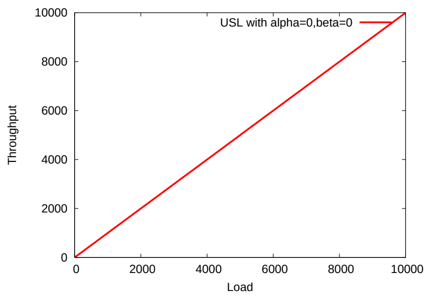
α is the amount of serialisation: represents Amdahl's law

β is the coherency delay in the system.

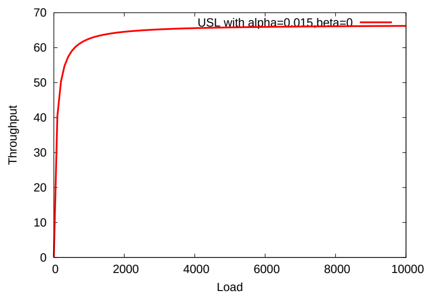
C is Capacity or Throughput

Neil Gunther (2002) captured this in his 'Universal Scalability Law', which is a closed-form solution to the machine-shop-repairman queueing problem.
It has two parameters, α which is the amount of non-scalable work, and $beta$ which is to account for the degradation often seen in system-performance graphs, because of cross-system communication ('coherency' or 'contention', depending on the system).
The independent variable N can represent applied load, or number of logic-units (if the work per logic-unit is kept constant).

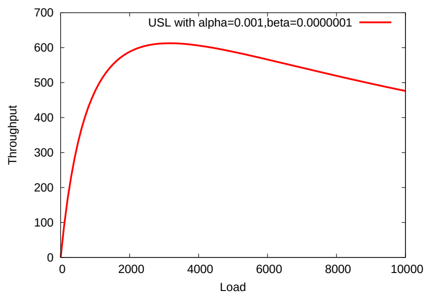
Scalability



$\alpha = 0, \beta = 0$



$\alpha > 0, \beta = 0$

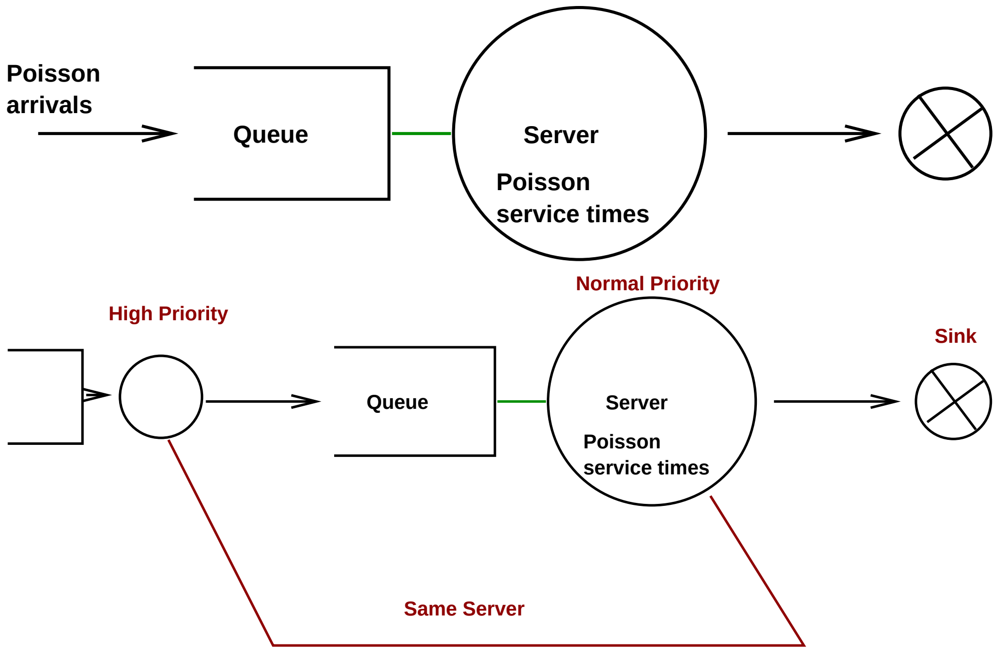


$\alpha > 0, \beta > 0$

Here are some examples. If α and β are both zero, the system scales perfectly—throughput is proportional to load (or to processors in the system).
If α is slightly positive it indicates that part of the workload is not scalable. Hence the curve plateaus to the right. Another way of thinking about this is that some (shared) resource is approaching 100% utilisation.
If in addition β is slightly positive, it implies that some resource is contended: for example, preliminary processing of new jobs steals time from the main task that finishes the jobs.

NICTA

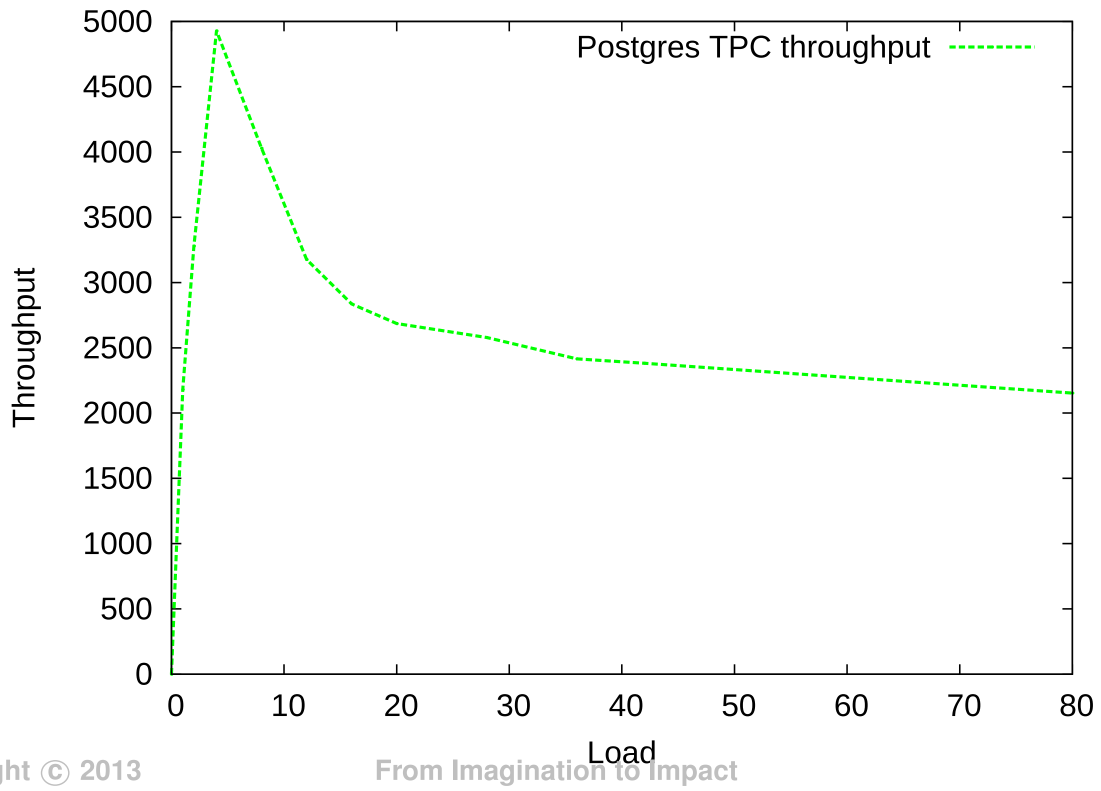
Queueing Models:



You can think of the system as in these diagrams. The second diagram has an additional input queue; the same servers service both queues, so time spent serving the input queue is stolen from time servicing the main queue.

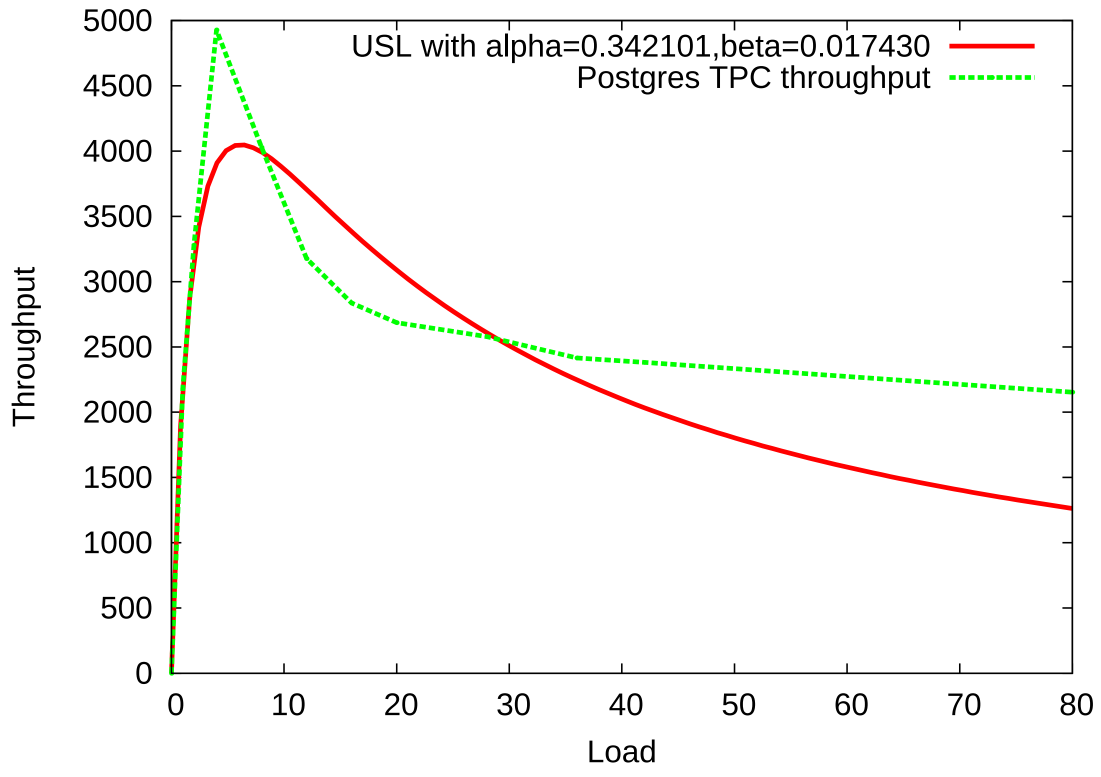
SCALABILITY

Real examples:



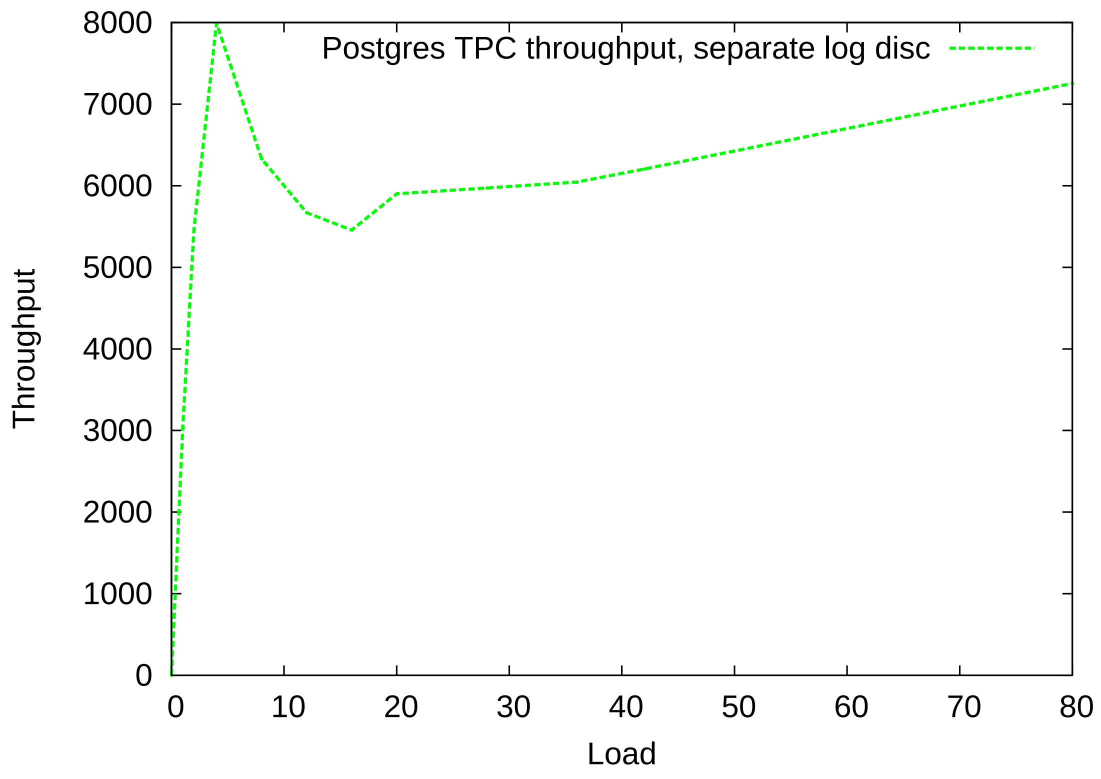
These graphs are courtesy of Etienne, Adrian and the Rapilog team. This is a throughput graph for TPC-C on an 8-way multiprocessor using the ext3 filesystem with a single disk spindle. As you can see, $\beta > 0$, indicating coherency delay as a major performance issue.

SCALABILITY



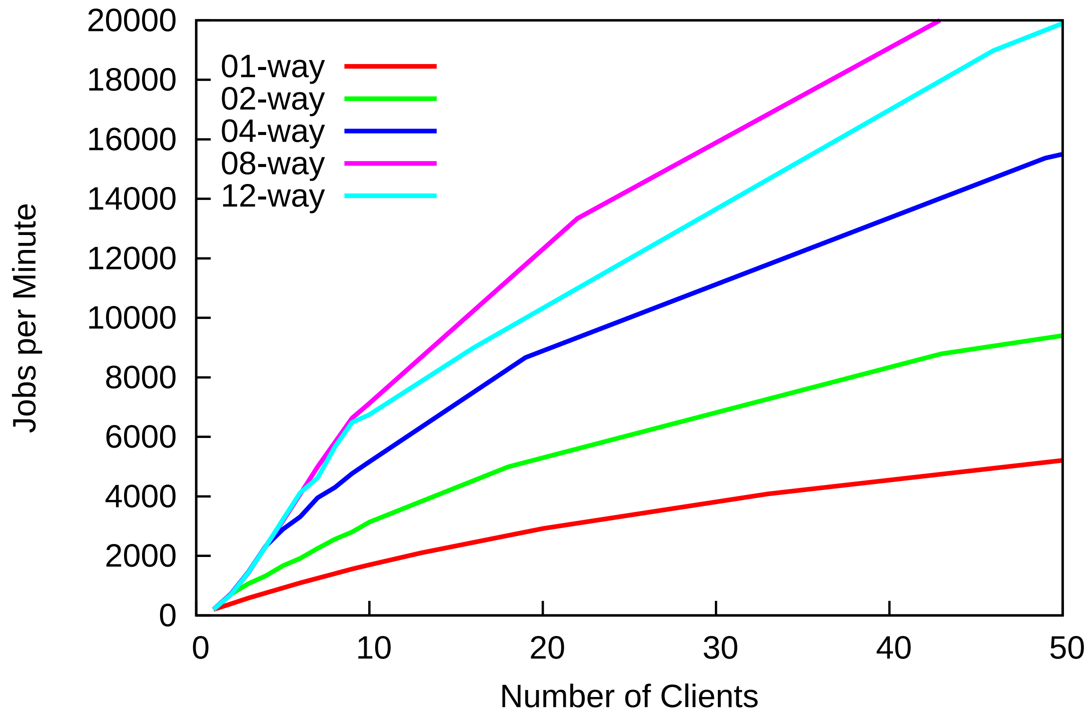
Using R to fit the scalability curve, we get $\beta = 0.017, \alpha = 0.342$ — you can see the fit isn't perfect, so fixing the obvious coherency issue isn't going to fix the scalability entirely.

SCALABILITY



Moving the database log to a separate filesystem shows a much higher peak, but still shows a $\beta > 0$. There is still coherency delay in the system, probably the file-system log. From other work I've done, I know that ext3's log becomes a serialisation bottleneck on a busy filesystem with more than a few cores — switching to XFS (which scales better) or ext2 (which has no log) would be the next step to try.

Another example:



This shows the reaim-7 benchmark running on various numbers of cores on an HP 12-way Itanium system. As you can see, the 12-way line falls below the 8-way line — α must be greater than zero. So we need to look for contention in the system somewhere.

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
```

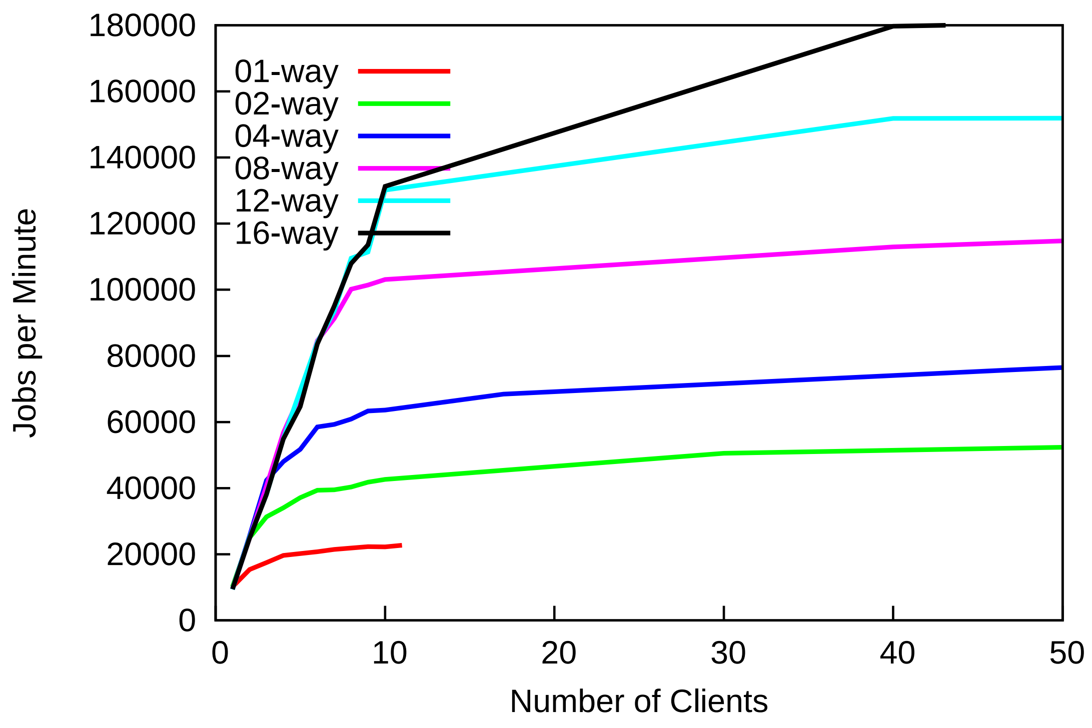
Lockmetering shows that a single spinlock in find_lock_page() is the problem:

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);
```

So replace the spinlock with a rwlock, and bingo:

SCALABILITY



The scalability is much much better.

TACKLING SCALABILITY PROBLEMS

NICTA

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

Fixing a performance problem for your system can break someone else's system. In particular, algorithms that have good worst-case performance on large systems may have poorer performance on small systems that algorithsm that do not scale. The holy grail is to find ways that work well for two processor and two thousand processor systems.

TACKLING SCALABILITY PROBLEMS

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

Performance and scalability work is like peeling an onion. Solving one bottleneck just moves the overall problem to another bottleneck. Sometimes, the new bottleneck can be *worse* than the one fixed.
Just like an onion, performance problems can make you cry.

DOING WITHOUT LOCKS

NICTA

Avoiding Serialisation:

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

If you can reduce serialisation you can generally improve performance on multiprocessors. Two locking techniques are presented here.

Doing without Locks

Sequence locks:

- Readers don't lock
- Writers serialised.

If you have a data structure that is read-mostly, then a sequence lock may be of advantage. These are less cache-friendly than some other forms of locks.

DOING WITHOUT LOCKS

Reader:

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

DOING WITHOUT LOCKS

Writer:

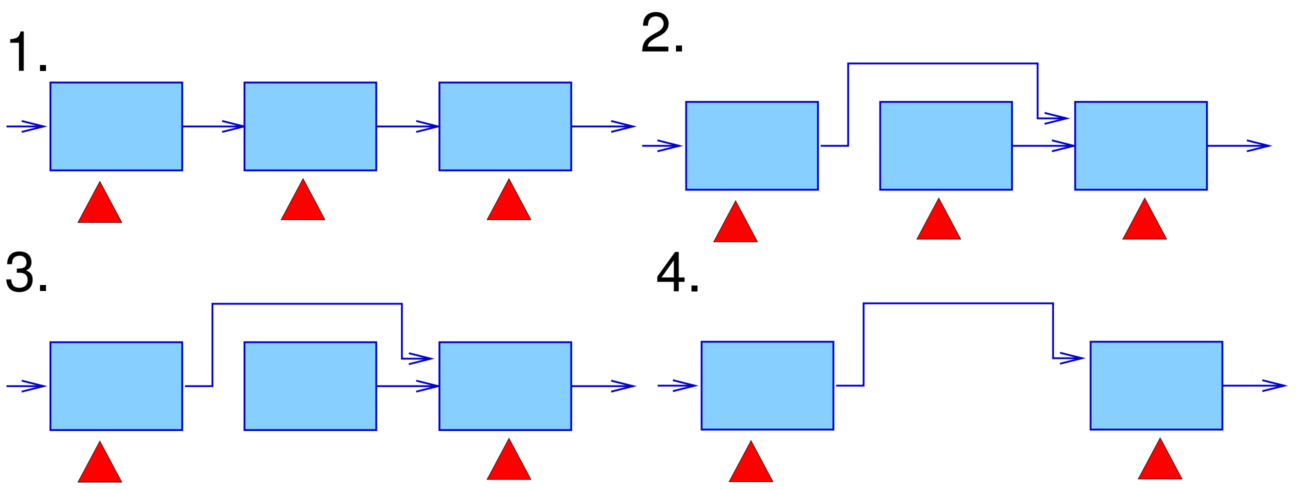
```
spinlock(&lck);
seq++; wmb()
 ...
wmb(); seq++;
spinunlock(&lck);
```

The idea is to keep a sequence number that is updated (twice) every time a set of variables is updated, once at the start, and once after the variables are consistent again. While a writer is active (and the data may be inconsistent) the sequence number is odd; while the data is consistent the sequence is even.
The reader grabs a copy of the sequence at the start of its section, spinning if the result is odd. At the end of the section, it rereads the sequence, if it is different from the first read value, the section is repeated.
This is in effect an optimistic multi-reader lock. Writers need to protect against each other, but if there is a single writer (which is often the case) then the spinlocks can be omitted. A writer can delay a reader; readers do not delay writers – there's no need as in a standard multi-reader lock for writers to delay until all readers are finished.

This is used amongst other places in Linux for protecting the current time-of-day.

DOING WITHOUT LOCKS

RCU: McKenney (2004), McKenney et al. (2002)



Another way is so called *read-copy-update*. The idea here is that if you have a data structure (such as a linked list), that is very very busy with concurrent readers, and you want to remove an item in the middle, you can do it by updating the previous item's *next* pointer, but you cannot then free the item just unlinked until you're sure that there is no thread accessing it.
If you prevent preëmption while walking the list, then a sufficient condition is that every processor is either in user-space or has done a context switch. At this point, there will be no threads accessing the unlinked item(s), and they can be freed.
Inserting an item without locking is left as an exercise for the reader.
Updating an item then becomes an unlink, copy, update, and insert the copy; leaving the old unlinked item to be freed at the next quiescent point.

Background Reading

References

Anderson, T. (1990), 'The performance of spin-lock alternatives for shared memory multiprocessors', *IEEE Transactions on Parallel and Distributed Systems* **1**(1), 6–16.
URL: `http://www.cs.washington.edu/homes/tom/pubs`

Corbet, J. (2008), 'Ticket spinlocks', *Linux Weekly News* .
URL: `http://lwn.net/Articles/267968/`

Dijkstra, E. W. (1965), 'Cooperating sequential processes

Background Reading

(ewd-123)'.
URL: `http://www.cs.utexas.edu/users/EWD/ewd01xx/`

Fraser, K. & Harris, T. (2007), 'Concurrent programming without locks', *ACM Trans. Comput. Syst.* **25**.
URL: `http://doi.acm.org/10.1145/1233307.1233309`

Hoare, C. (1974), 'Monitors: An operating system structuring concept', *CACM* **17**, 549–57.

McKenney, P. E. (2003), 'Using RCU in the Linux 2.5 kernel', *Linux Journal* **1**(114), 18–26.

Background Reading

URL:
http://www.linuxjournal.com/article/6993

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. (2010), 'Memory barriers: A hardware

Background Reading

view for software hackers',
http://www.rdrop.com/users/paulmck/scalabil

McKenney, P. E., Sarma, D., Arcangelli, A., Kleen, A., Krieger, O. & Russell, R. (2002), Read copy update, *in* 'Ottawa Linux Symp.'.
URL:
http://www.rdrop.com/users/paulmck/rclock/r

Ritchie, D. M. (1984), 'The evolution of the UNIX time-sharing system', *AT&T Bell Laboratories Technical Journal* **63**(8), 1577–1593.

Background Reading

URL:

ftp://cm.bell-labs.com/who/dmr/hist.html

Ritchie, D. M. & Thompson, K. (1974), 'The UNIX time-sharing system', *CACM* **17**(7), 365–375.

Schimmel, C. (1994), *UNIX Systems for Modern Architectures*, Addison-Wesley.

Segall, Z. & Rudolph, L. (1984), Dynamic decentralized cache schemes for an MIMD parallel processor, *in* 'Proc. 11th Annual International Symposium on Computer Architecture', pp. 340–347.