Linux, Locking and Lots of Processors

Peter Chubb — Principal Research Engineer
peter.chubb@data61.csiro.au
July 17, 2020

To give a complete rundown on the features and internals of Linux would take a very very long time — it's a large and complex operating system, with many interesting features.
Instead, I'm going to give a bit of history of POSIX OSes leading up to Linux, a (very brief) skim over the key abstractions and how they're implemented, and then talk about some things I've done over the last few years to improve Linux scalability. I'll also touch on Amdahl's law and Gunther's more general 'Universal Scalability Law' in this section, and talk about the way locking works for multi-processor support in the Linux kernel.

A little bit of history

- MULTICS in the ’60s
- Ken Thompson and Dennis Ritchie in 1967–70
- USG and BSD
- John Lions 1976–95
- Andrew Tanenbaum 1987
- Linus 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 a bit burnt-out programming MULTICS and wanted to port 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. And the key ingredients there were firstly that the OS did not interpret file contents — an ordinary file is just an array of bytes. Semantics are imposed by the user of the file. And secondly, a simple hierarchical naming system that hid details of disc layout and disc volumes. The inode-based filesystem is the core of POSIX-like OSes; it was sketched up over a lunchtime on a whiteboard by Denis Ritchie and Ken Thompson.

Other people found UNIX 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.

John Lions and Ken Robinson here at UNSW read Ritchie & Thompson (1974), and decided to try to use UNIX as a teaching tool. Ken sent off for the tapes, the department put them on a PDP-11, and started exploring. The licence 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.

(It's worth also looking at AUUGN 14(4)
`https://minnie.tuhs.org/Archive/Documentation/AUUGN/AUUGN-V14.`
which was a special history issue and includes an interview from Greg Rose who was involved in implementing the first UNIX system at UNSW, and an article by Tony McGrath, who was involved in the first port of UNIX at the University of Wollongong)

The University of California at Berkeley was another of those universities. In

1977, Bill Joy (then a postgrad, later the co-founder of Sun Microsystems) 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 (which later became `vi`). Later BSDs contained contributed code from other universities, including UNSW. The BSD tapes were freely shared between source licensees of AT&T's UNIX.

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, *samizdat*. 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.

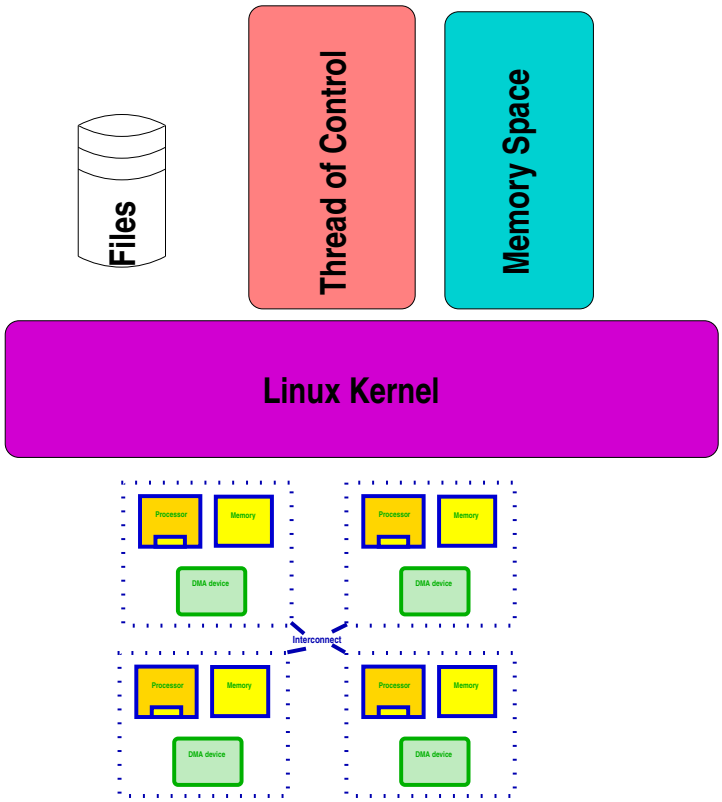
A little bit of history

- 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 — a single thread of control in an address space — and the file system model have remained the same. The IPC model (the so called Sys V shared memory, semaphores, and messages) (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 open-source-like patch-sharing.
Interestingly, most of these ideas were already in MULTICS. The difference is that in MULTICS they were designed in from the start (and delivered late) as opposed to delivering something that worked early, and adding features as they became desirable.

In the late 80s the core interfaces were standardised by the IEEE working with USENIX, in the POSIX standards.

Abstractions



As in any POSIX operating system, the basic idea is to abstract away physical memory, processors and I/O devices (all of 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). Unlike in a microkernel, some default policy is embedded in the kernel; but the general principle is to provide tools and mechanisms for an arbitrary range of policies.

Abstraction also occurs *inside* the kernel, to improve portability. Linux runs on 25 different architectures, with multiple variants of each.

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

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.

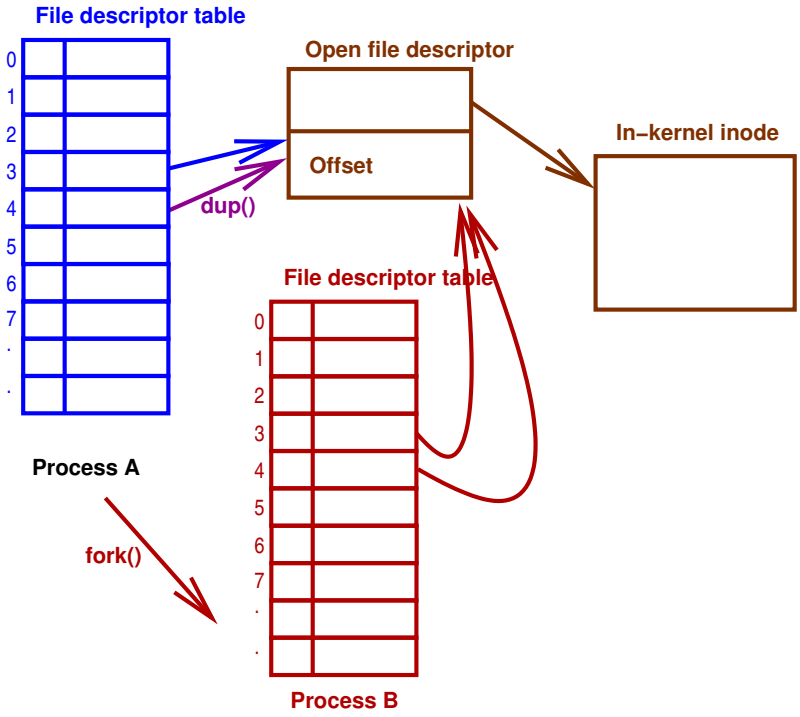
`fork()` and `exec()`

- 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.

`fork()` and `exec()`

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*.

When a process calls **`open()`**, the file descriptor table is scanned from 0, and the index of the next available entry is returned. (In 5th edition UNIX this was a linear scan of a fixed-size array; later Unices improved both the data structure (to allow unlimited FDs) and the scanning (to replace O(n) with a faster algorithm).

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, and its open files have the same close-on-exec flags as those in its parent.

You can think of a file descriptor as a capability to an object. A file descriptor can be created by any process that has the appropriate rights on an object, and then can be passed around, either by inheritance, or through interprocess communication (**`pipe()`** on System-V, UNIX-domain sockets on other systems). The rights to the object can be dropped after creating the file descriptor. Such capabilities cannot however be revoked.

`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.

Other attributes of the process can also be changed (e.g., changing the owner by **`setuid()`**). Most state-changing system calls operate on the current process; the **`fork()`**/**`exec()`** model allows all these to be used without having to either create a massively complex system call that specifies everything about a new task (as VMS and systems derived from it do) or having the locking complexity of being able to operate on a different process.

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. When certain special characters are typed (typically ^C, ^\, and ^Z), the controlling terminal's driver generates signals to the foreground process instead of passing through the character.

The problem with `fork()`

- Almost perfect in original system
- But:
  - Address spaces now bigger and managed with pages
    - Slow to copy page tables
  - Multi-threading breaks semantics
    - Child no longer an exact copy — only one thread `fork()`ed
    - Much more per-process state, not all inheritable

In the first Unix systems, the MMU used sets of base limit registers to create segements. Because the 16-bit address space was small, and main memory was typically at most 256k, swapping of entire programs to secondary storage was (for the time) fast, and common. The initial implementation of `fork()` just caused a swapped copy to be made, adjusted the return value to 0 and continued.
Originally, on 16-bit machines with 16-bit words, the entire address space was copied. This took only a few hundreds of cycles. When 32-bit systems came along, along with bloat from added layers of abstraction, shared libraries, etc., it started to take too long. Two competing mechanisms were implemented: in UCB-derived kernels, `vfork()` was implemented, that copied the process control information, but shared the address space. It was expected that the only things a process would (usually) do between `vfork()` and calling either `exit()` or `exec()` was fiddle around with file descriptors. The parent was paused after `vfork()` until the child called `exit()` or `exec()`

AT&T kernels instead implemented copy-on-write for the address spaces. Only pagetable information was copied, a much smaller job than copying an entire address spce. All pages in both parent and child were then set read-only; the first time a write occurred, the page was cloned and made writeable. But nowadays, proceses have grown huge. A significant amount of time is spent copying page tables on `fork()`; for the common case where there are a few system calls then `exec()` replaces them all this is mostly wasted work. What's more, as multicore systems are now common, so are multi-threaded processes. Only one thread is copied into the child; but the states of all locks are inherited because they're just values in the (copied) address space.
In addition, modern POSIX proceses have many more attributes: memory locks, SIGIO, containerisation state, sockets, message queues, timers, etc., etc. Some of these sometimes make sense to inherit, but many do not. So the simple fork+exec model doesn't work as well as it used to.

Because of the constrained semantics of `vfork()` modern POSIX systems use only the Copy-On-Write fork— if `vfork()` is provided, it is an alias for **`fork()`**. However, even COW `fork()` on 64-bit systems is beginning to be a bottleneck, so maybe `vfork()` will become popular again. And on multithreaded programs, `fork()` itself has restrictions: only async-signal-safe functions can be called between `fork()` and `exec()`.
Recent Linux kernels (in the last ten years) have had a **`clone()`** system call as well, that allows fairly fine grain control over what is inherited. In particular, by inheriting (rather than marking as COW) the address space, one can implement multiple threads in one process.
And there is a **`posix_spawn()`** call nowadays for creating and initialising a process. It is harder to use than **`fork()`**, in my opinion.
See Baumann et al. (2019) for a nice rant on `fork()`.

Permissions Model

- Based on logged-in-users
- UID, GID, Other — rwx
- Mainly for File access.

The first use of UNIX was to play a video game, single user. It's amazing that any permission model at all was in the filesystem — MSDOS and similar never had any. UNIX was very early on used at AT&T corporate for document processing. The permissions model fits well for mostly cooperative work: people who collaborate are in the same group, and so can read/write files that're marked group read/write.

Some things had to be done at an elevated privilege though. For example, in the early UNIX, **`mkdir()`** was not a system call. A directory was just a file (like any other) whose structure was understood by the kernel. So setting the 'This is a directory' bit in an inode was a privileged operation' and writing directories was privileged. A user-mode helper lived in `/etc` — but it had to run at an elevated privilege. Enter the 'setuid' permission bit: when an executable in a file with 'setuid' is execed, its UID is set to that of the owner of the file. Of course, this only works if the code in that executable is trustworthy. But the early setuid programs were very simple and easy to see they were correct.

The notion of groups has changed since the original. It used to be that a process was only a member of one group at a time; the **`newgrp()`** system call could be used to move to any group the UID was marked as member of. The BSD UNIX variants introduced the idea that a process could be in a set of groups at once, and apply group permission to files in accordance with any of the groups a process is in.

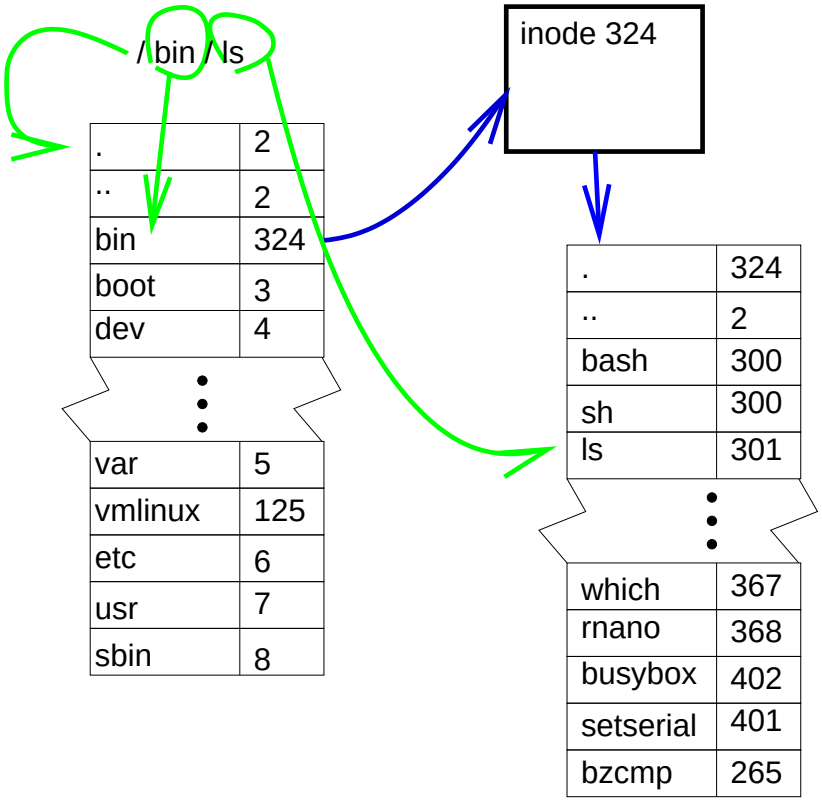
File model

- 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 based on who you are.

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. One very common organisation was essentially an image of a punched-card deck! By contrast, UNIX regular files are just a collection of bytes, indexed from zero.

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. The added flexibility makes directory operations more expensive, but allows Linux to deal with over thirty different filesystems, with varying naming models and on-disk structures.

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 for the current process (otherwise it starts at the current process's current directory). The first link in the pathname is extracted (`"bin"`) by calling into the filesystem code, and searched for in that 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"`.
Every process has a 'current root directory' — the privileged `chroot()` system call allows a process to change its root directory to any directory it can see. This is a necessary feature to provide *containers*, as it provides namespace isolation for files.

namei

- translate name → 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 (done naively, it involves reading a block from the disk for each pathname component); Linux currently uses a cache to speed up lookup. This cache in turn has become a scalability bottleneck for large SMP systems.

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. Some filesystems encode the symbolic name into the directory entry, rather than having a separate file.

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. Many of the utilities that are common on UNIX started out as much simpler programs wrapped in shell scripts; as people elaborated the scripts to provide more functionality, they became less maintainable or too slow, and eventually were refactored into a compiled language.
(Multics used the opposite approach, with around 3000 pages of complex design documents developed and reviewed before a line of code was written. Many of the early design decisions had to be reversed when hardware became available and implementation started).

Linux 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`

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 system 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, and for ensuring that appropriate conversions happen between on-disk, or on-the-wire, and host endianness.

Linux C Dialect

- Extra iterators
  - *type_name*`_foreach()`
- Extra O-O accessors
  - `container_of()`
- Macros to register Object initialisers

Object-oriented techniques are used throughout the kernel, but implemented in C.
Almost every aggregate 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.
In addition there is a family of macros to register initialisation functions. These are ordered (early, console, devices, then general), and will run in parallel across all available processors within each class.

Linux C Dialect

- Massive use of inline functions
- Quite a big use of CPP macros
- Little `#ifdef` use in code: rely on optimiser to elide dead code.

The kernel is written in a style that attempts not to 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. Because the kernel is huge, but not all files are included in every build, there has to be a way to register initialisation functions for the various components. The Linux kernel is quite object-oriented internally; but because it runs on the bare metal, functions that would usually be provided by language support have to be provided by the OS, or open coded. The **`container_of()`** macro is a way to access inheritance; and the *`xxx`*`_`**`initcall()`** macros are a way to handle initialisation. Obviously, initialisation has to be ordered carefully; but after interrupts are set up, all the processors are on line, and the system has a console, the remaining device initialisers are run; then all the general initialisers.

Internal Abstractions

- MMU
- Memory consistency model
- Device model

Linux has many internal abstractions for portability. The two biggest are the MMU, where differences in page table layout are largely hidden from most of the code, using per-architecture macros to walk and update them; and the memory consistency model that the Linux kernel provides.

The Linux kernel runs on architectures with very different underlying consistency models, the weakest being DEC Alpha, the strongest X86. Code that runs on all of these is expected to include memory barriers, and acquire/release instructions, that allow it to work correctly on all architectures. Linux thus provides its own memory consistency model. It's worth taking a look in `tools/memory-model` in the kernel source, to see an executable version of the model, that allows one to check if any particular outcome can occur, given a sequence of loads, stores, and memory-consistency-affecting operations (such as atomic operations, fences, barriers, aquire/release macros, etc).

For more detail on memory models, see McKenney (2010).

I'll cover the device model later. Essentially, it treats all devices as being available via a run-time bus enumeration *even where the underlying buses are not enumerable*.

Scheduling

Goals:

- dispatch O(1) in number of runnable processes, number of processors
  - good uniprocessor performance
- ‘fair’
- Good interactive response
- topology-aware
- O(log n) for scheduling in number of runnable processes.

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); and, at a higher level, applications should not be able to get more CPU by spawning more threads/processes.
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.

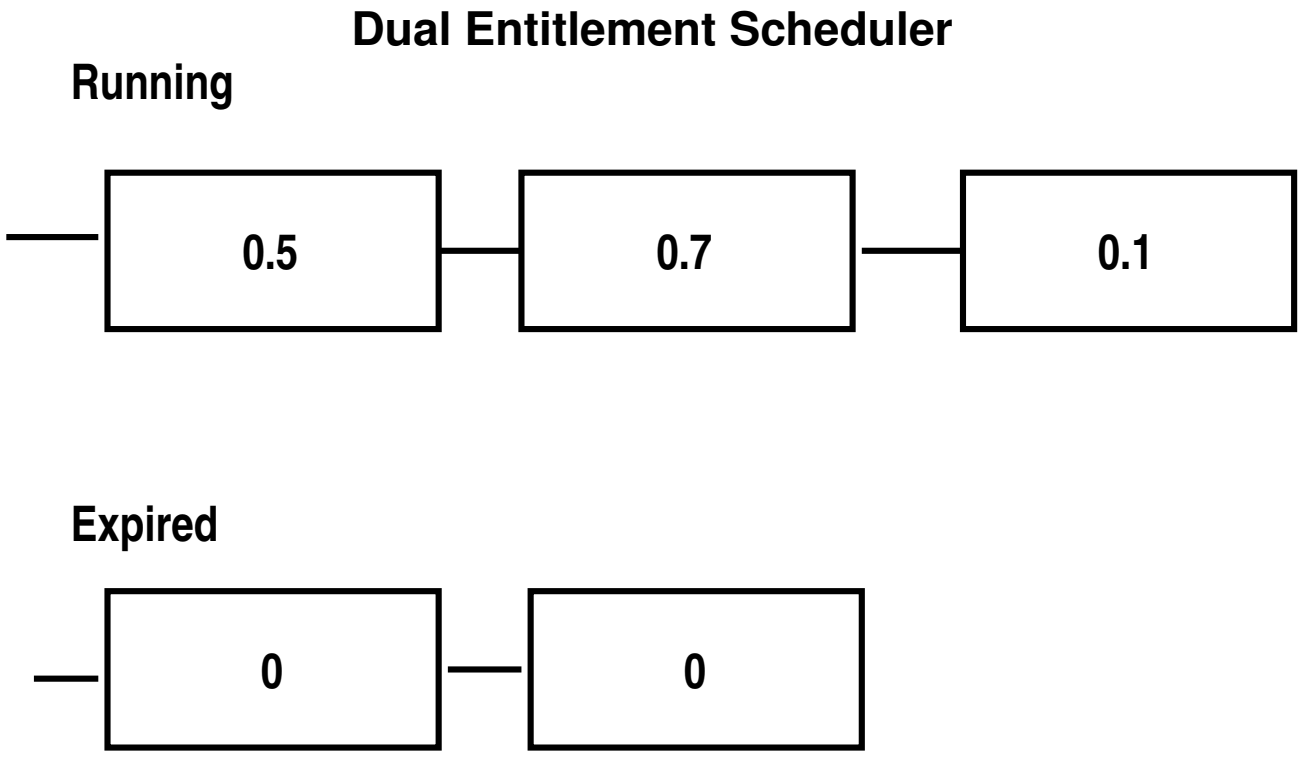
Scheduling

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.

Scheduling



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 distribute 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 or more 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 so-called 'Completely Fair Scheduler'

Scheduling

CFS:

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

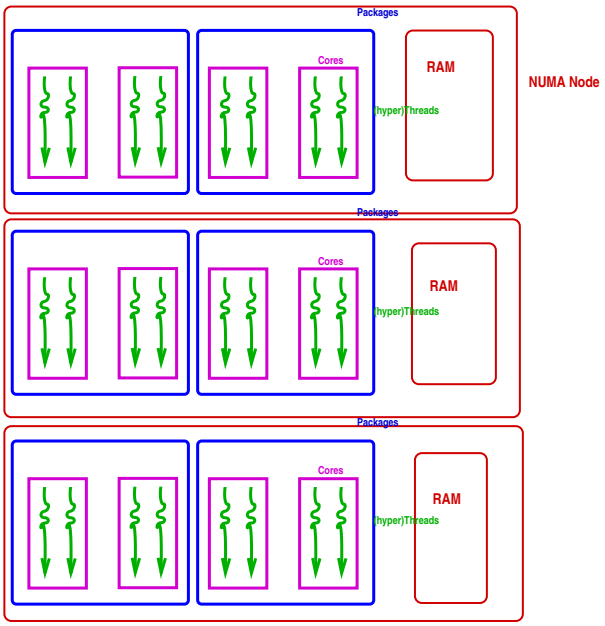
Devil's in the details:

- 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_2 of the number of available processors. Within a period, each task gets a time quantum (the period divided by the number of tasks) 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' (`vruntime`) instead of actual; virtual runtime is normalised to the number of running tasks. It is also adjusted regularly to avoid overflow (this adjustment means the algorithm isn't totally fair: CPU-bound processes end up being penalised with respect to I/O-bound processes, but this is probably what is wanted for good interactivity)

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.
In any case, the scheduler forces child processes to run before their parents immediately after a **`fork()`**, to minimise the amount of page duplication for copy-on-write.

Scheduling



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 (can power them off)
- Somehow identify cooperating threads, co-schedule 'close by'?

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 cgroup, 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 can be a long way off (in terms of access time). 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. It turns out from some recent work (Lepers et al. (2015)) that bandwidth between nodes should also be taken into account for optimal performance, but this hasn't yet made it into the Linux kernel.

Linux currently uses a ‘first touch’ policy: the first processor to write to a page causes the frame for the page to be allocated from that processor’s nearest memory. On **`fork()`**, the new process’s memory is allocated from the same node as its parent, and it runs on the same node (although not necessarily on the same core). **`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.

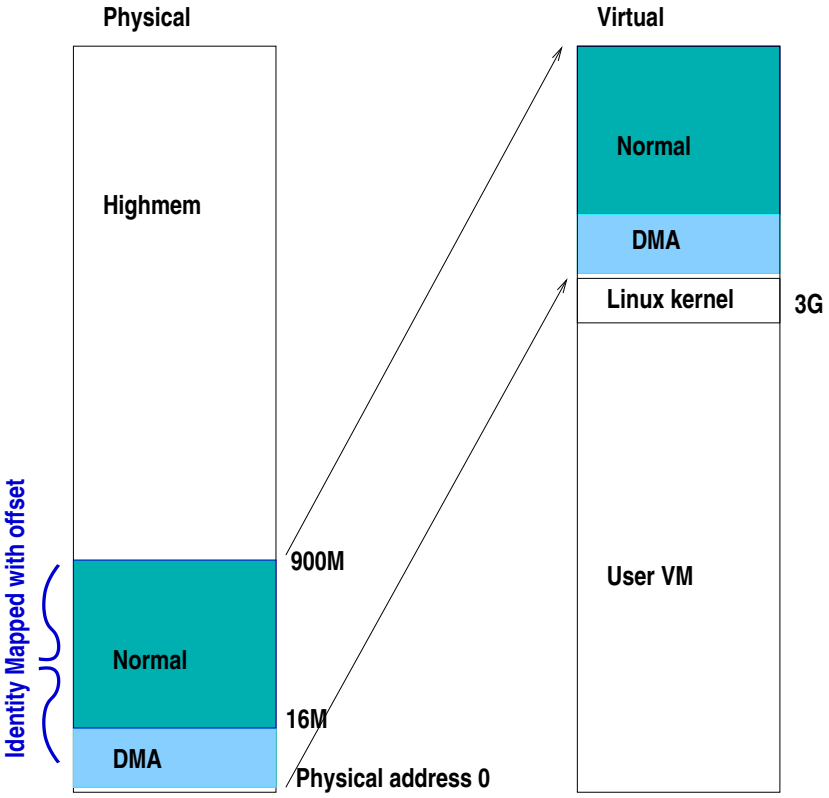
Scheduling

- 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 most of the time.

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 (e.g., because there is more than 4G of RAM on a 32-bit machine) is termed 'Highmem' 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.

The boundary between user and kernel can be set at configuration time; for 64-bit x86_64 systems it's at 2^{63} – i.e., all addresses with the highest bit set are for the kernel.

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).

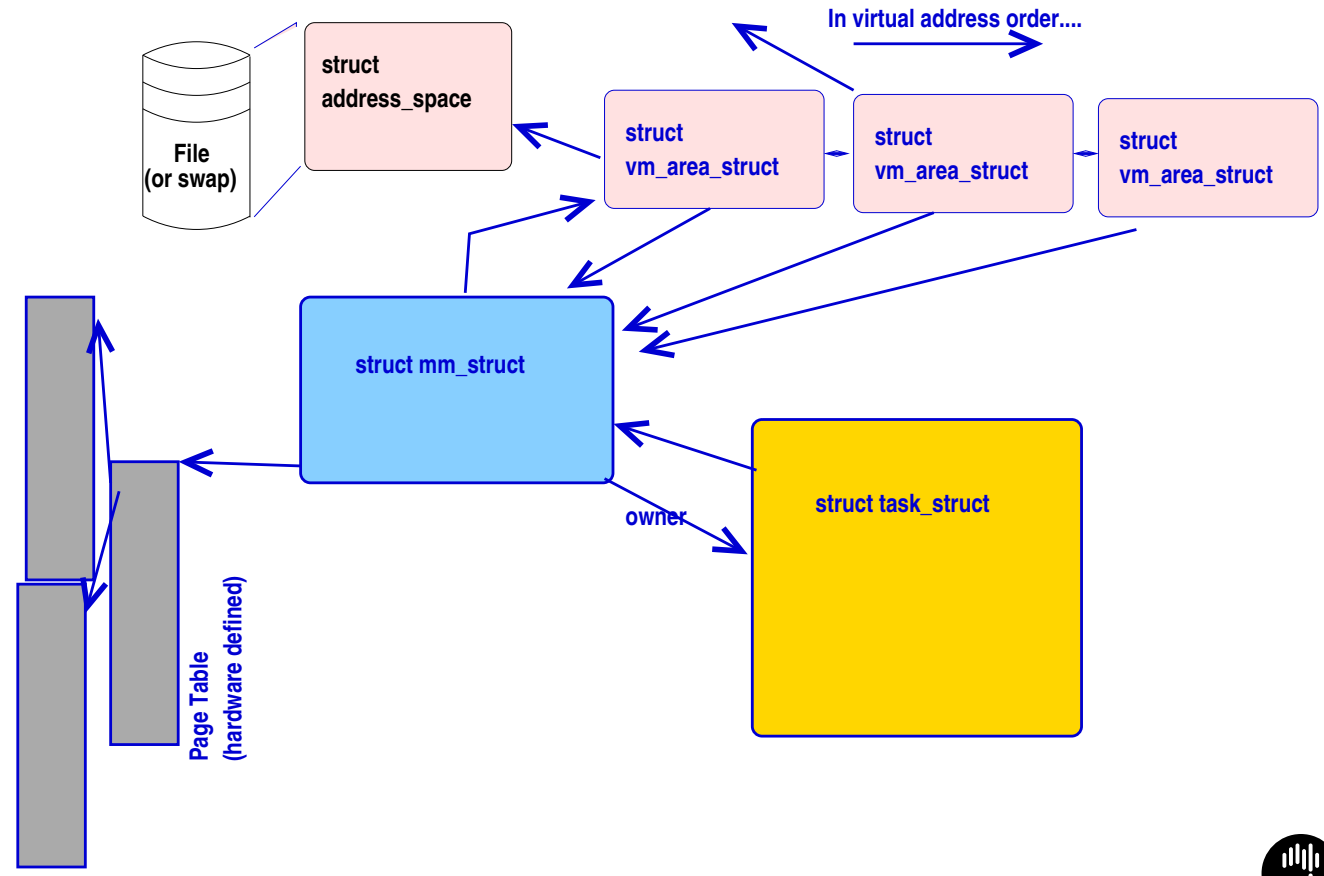
Memory Management

`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 huge pages). 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, huge pages, and transparent 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 to use the standard abstractions), 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. Any **`munmap()`** calls that split a mapping, or **`mprotect()`** calls that change part of a mapping can also create new VMAs.

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

  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.
The kernel first looks 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, unless it's next to a stack segment, and the faulting address is at or near the current stack pointer, in which case the stack needs to be extended.
If it finds the VMA, then it checks that the 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, its 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 essentially 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. The operating system needs to be told what's available.

Driver Interface

Enumerable buses:

```
static DEFINE_PCI_DEVICE_TABLE(cp_pci_tbl) = {
  { PCI_DEVICE(PCI_VENDOR_ID_REALTEK,
        PCI_DEVICE_ID_REALTEK_8139), },
  { PCI_DEVICE(PCI_VENDOR_ID_TTTECH,
        PCI_DEVICE_ID_TTTECH_MC322), },
  { },
};
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

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 (old way):

```
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'.
Most platforms have moved away from using these platform devices in favour of using a Device Tree (see later),

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.

Device Tree

- Describe board+peripherals
  - replaces ACPI on embedded systems
- Names in device tree trigger driver instantiation

Current kernels have moved away from putting platform devices into C code, in favour of using a *flattened device tree*, which describes the topology of buses, devices, clocks and regulators, so a single kernel can run on more than one board.
Each node in a device tree (which you can find in **`arch/arm/boot/fts/*`** for ARM processors) contains a 'compatible' field that says which driver to invoke, and other (node-specific) entries that give the addresses, interrupts, clocks, etc., necessary to configure and use the device the node represents.

Device Tree

```
uart_A: serial@84c0 {
        compatible = "amlogic,meson6-uart", "amlogic,me
        reg = <0x84c0 0x18>;
        interrupts = <GIC_SPI 26 IRQ_TYPE_EDGE_RISING>;
        status = "ok";
};
```


This example from the Odroid C2's device tree shows the main bits. The **`compatible`** line both triggers the **`probe`** routine for drivers that register for it, and is searched for by drivers to find the rest of the information. The **`reg`** line gives the address and size of the registers for the UART. And so on.

Containers

- *Namespace* isolation
- Plus Memory and CPU isolation
- Plus other resources

In hierarchy of control groups

Used to implement, e.g., **`Docker`**

`chroot`, which has been present in UNIX kernels since edition 7, and in every Linux release, provides filesystem namespace isolation. Work derived from UNSW and USyd's fair share scheduler and Limits system was sold to Sun in 2000, and was developed to provide 'zones'. Each zone could be assigned a proportion of the CPU and memory, and a part of the process ID and user ID namespaces, in a way that was mostly transparent to processes running in the zone.

In Linux kernel 3.8, the same kind of thing was implemented (the actual implementation was independent), to provide *Linux Containers*.

Controllers can be configured separately for various resources, including but not limited to, CPU, memory, user ID, socket namespace, and **`chroot`** used for filesystem namespace. These have control files mounted in a hierarchy under **`/sys/fs/cgroup`**, which can be manipulated directly; but it is generally better to use either **`lxc`**, **`libvirt`**, or **`Docker`** as middleware to manipulate more than one controller at a time.

On a single socket system with a small NUMA factor, Linux containers provide reasonable isolation at low overhead. Where there is significant asymmetry in the NUMA topology, or where the NUMA factor is large, containers can fail to isolate because of contention on communications channels it doesn't control (see Lepers et al. (2015) for details).

Summary

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

The linux kernel keeps changing really fast. However, core abstractions like the ones mentioned have been reasonably stable for the last few years: churn is mostly in new drivers, and new features (like containerisation and RISC-V support)

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

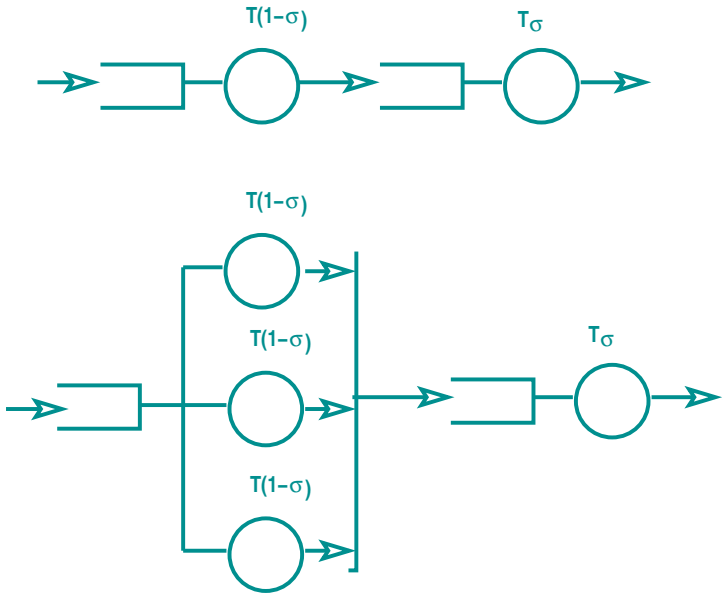
We've seen that because of locking and other issues, some portion of the multiprocessor's cycles are not available for useful work. In addition, some part of any workload is usually unavoidably serial.

Scalability

Amdahl's law:

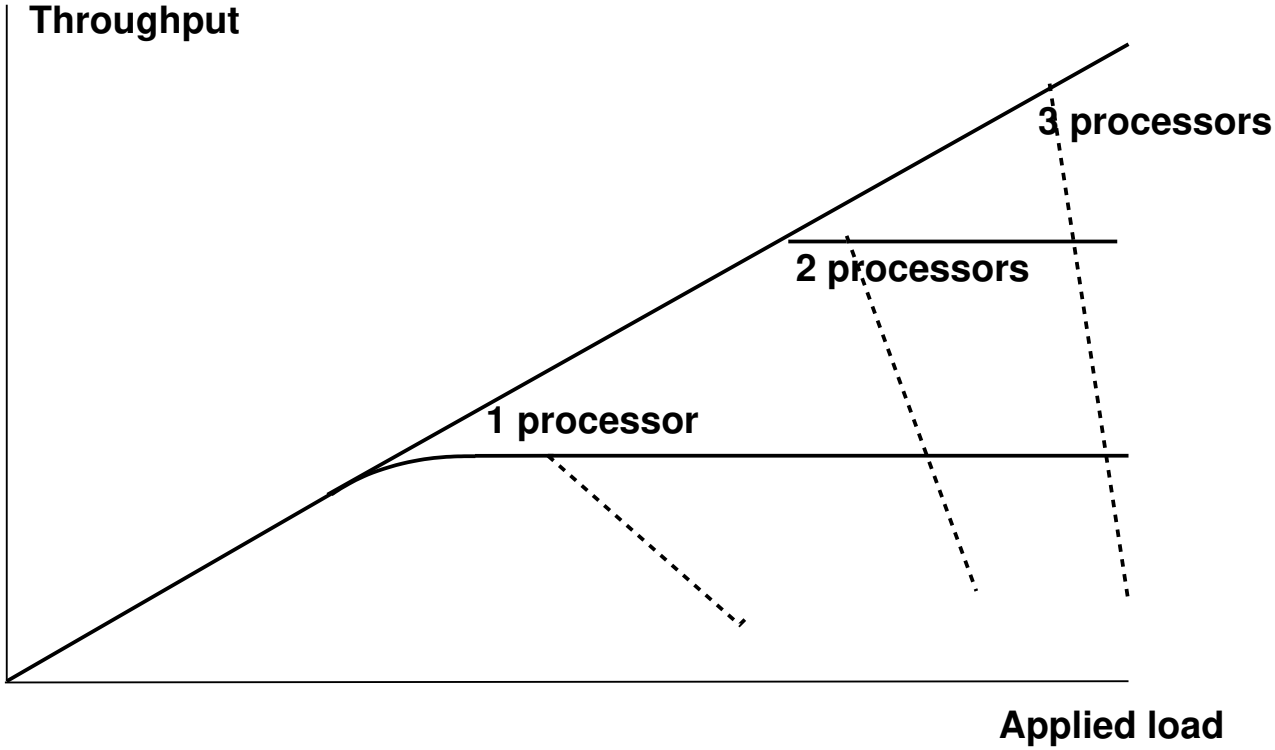
If a process can be split such that σ of the running time cannot be sped up, but the rest is sped up by running on p processors, then overall speedup is

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



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



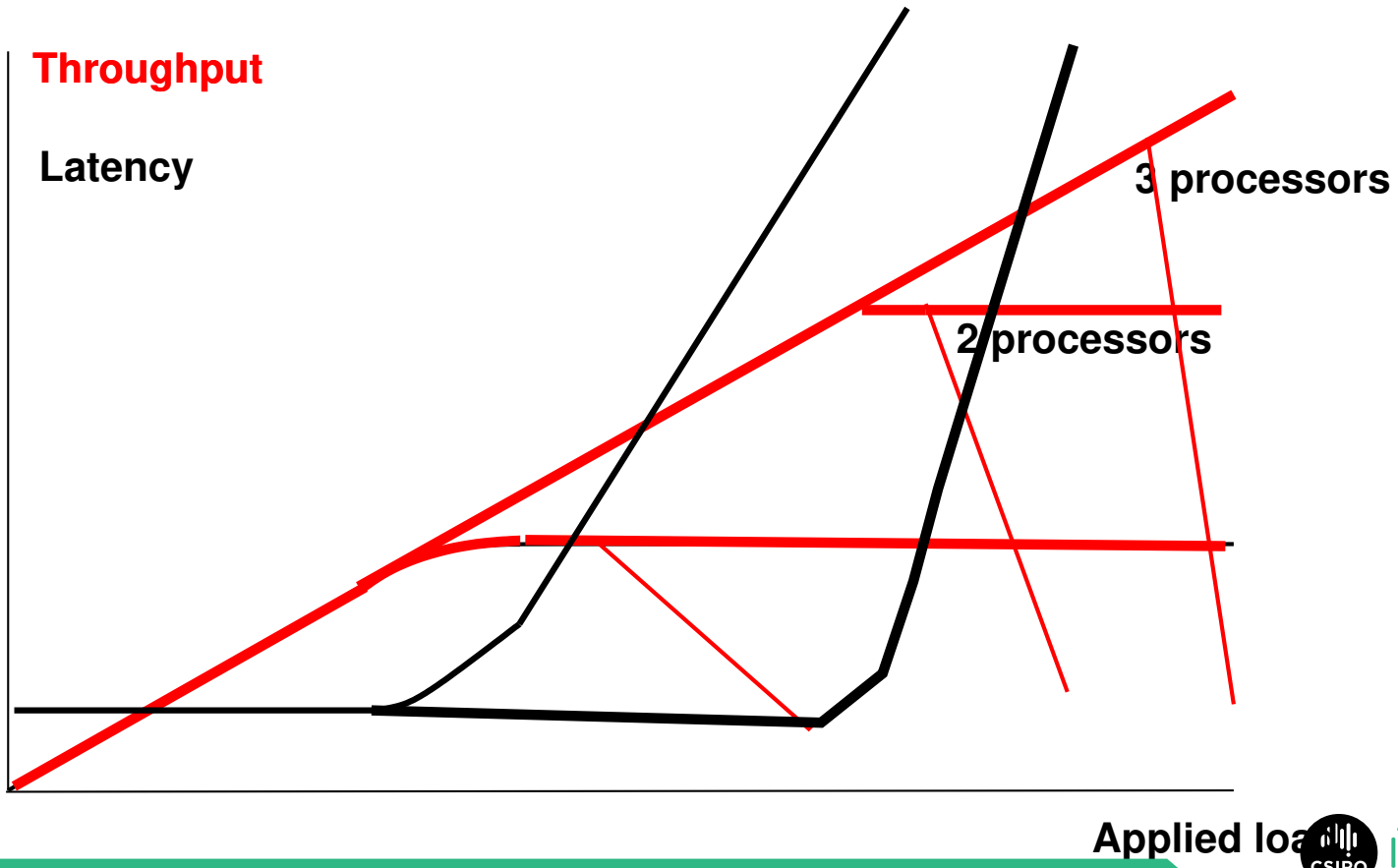
The general scalability curve looks something like the one in this slide. The Y-axis is throughput, the X-axis, applied load. Under low loads, where there is no bottleneck, throughput is determined solely by the load—each job is processed as it arrives, and the server is idle for some of the time. Latency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

Gunther's law:

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

where:

N is demand

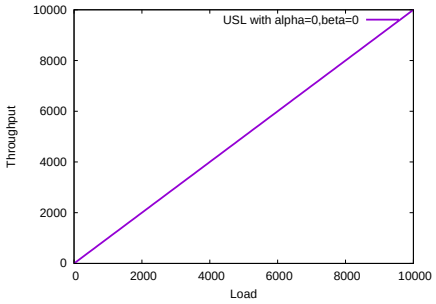
α is the amount of serialisation: represents Amdahl's law

β is the coherency delay in the system.

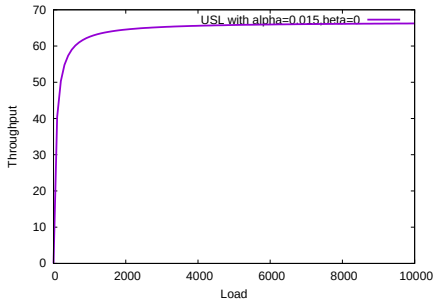
C is Capacity or Throughput

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).

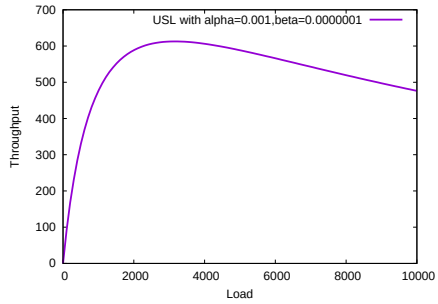
Scalability



$\alpha = 0, \beta = 0$



$\alpha > 0, \beta = 0$

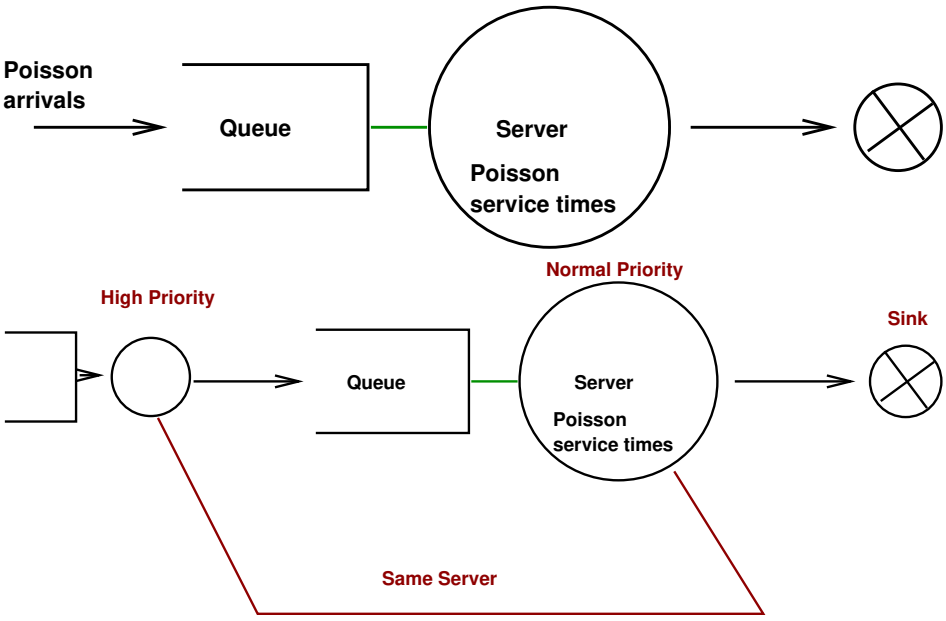


$\alpha > 0, \beta > 0$

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.

Scalability

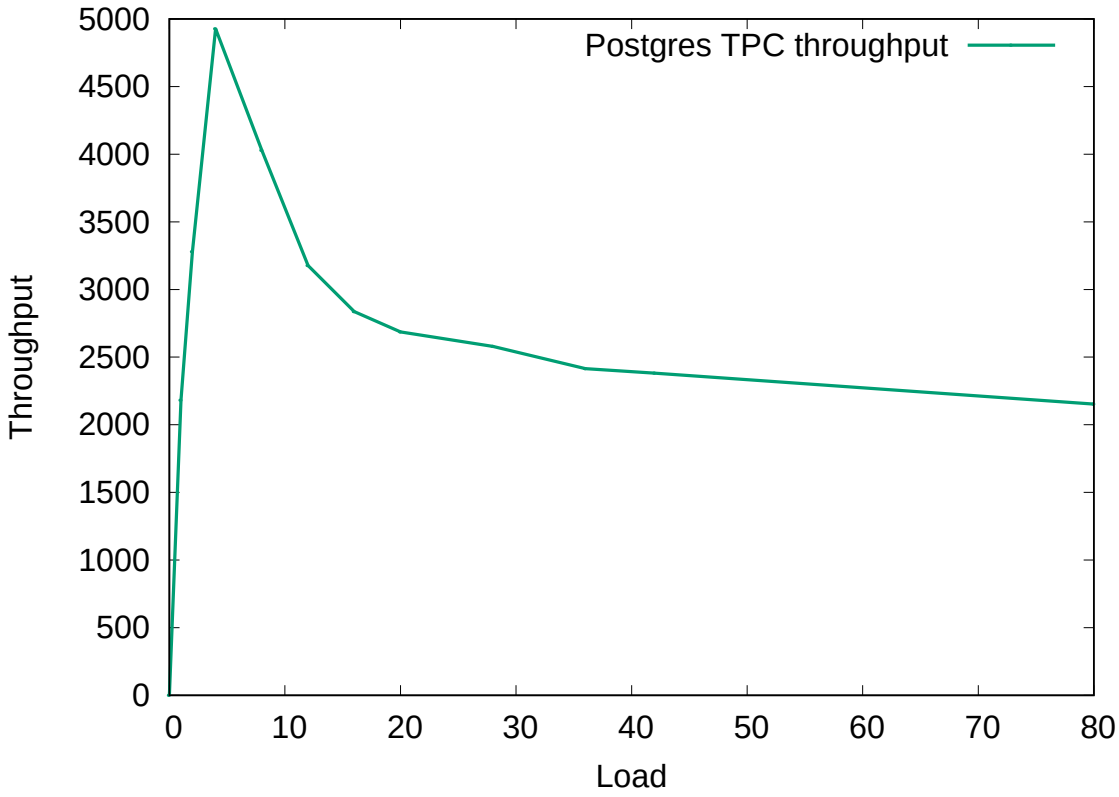
Queueing Models:



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.

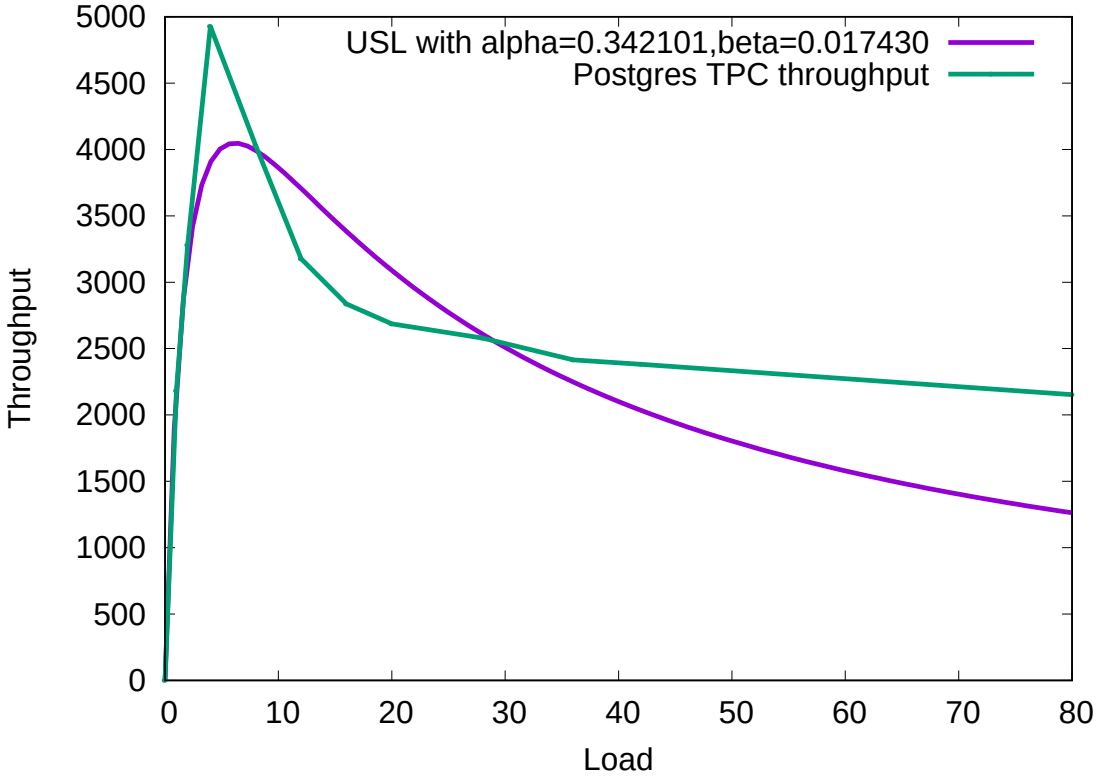
Scalability

Real examples:



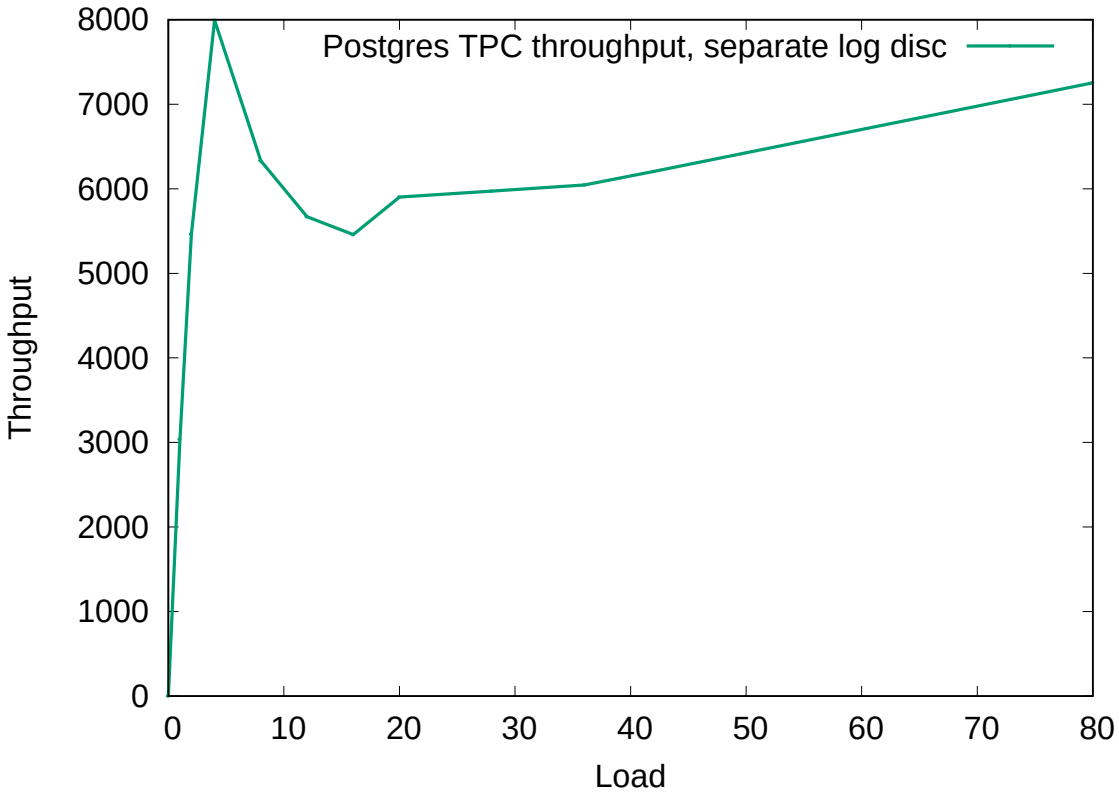
These graphs are courtesy of Etienne Le Sueur, Adrian Danis, and the Rapilog team. This is a throughput graph for TPC-C on an 8-way multiprocessor using the ext3 filesystem with a single disk spindle. As you can see, $\beta > 0$, indicating coherency delay as a major performance issue.

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.

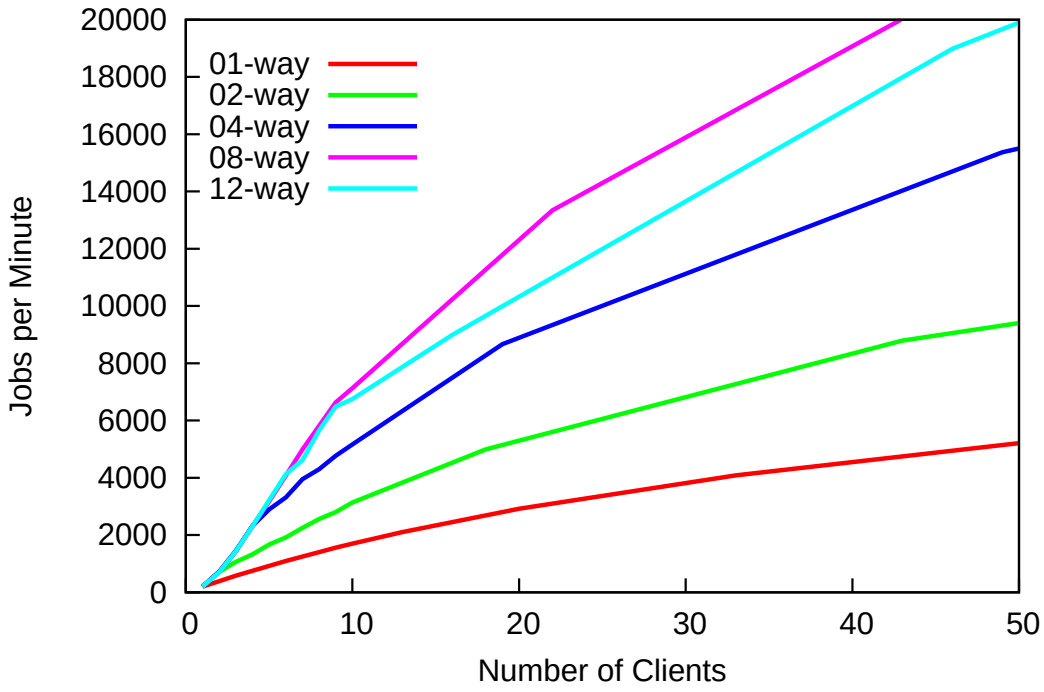
Scalability



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.

Scalability

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 queuing 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:

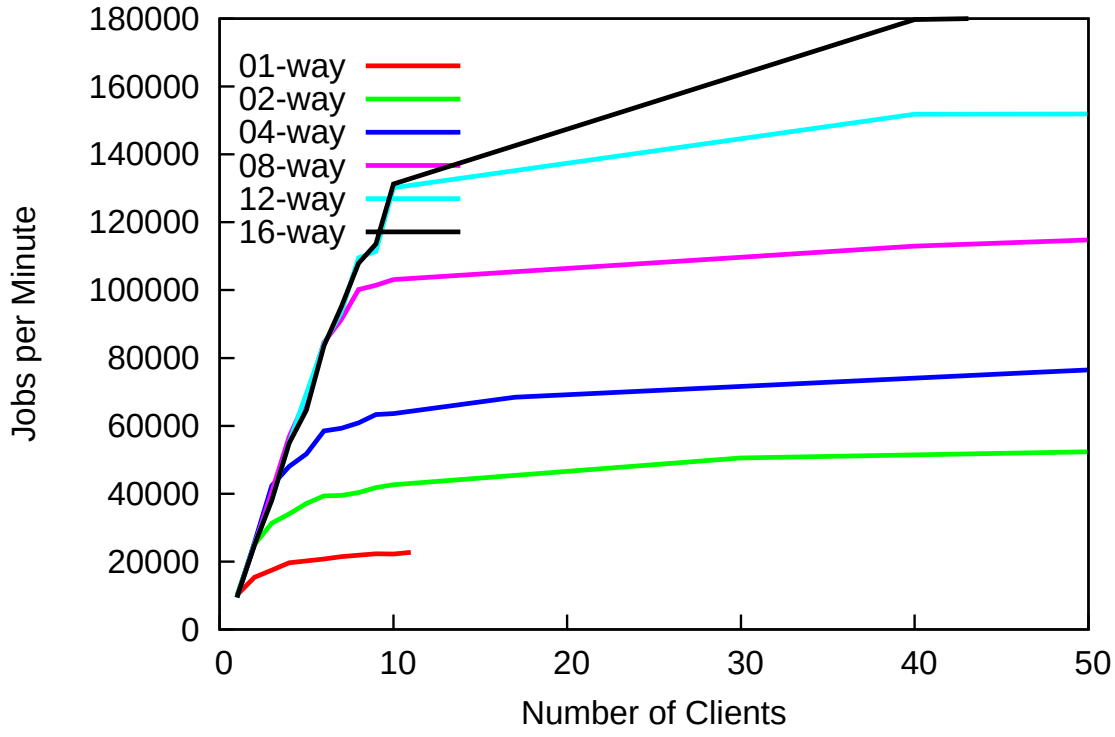
Scalability

```
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. Not only can we now extend to 16 processors, the raw performance is an order-of-magnitude better even on single core.

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

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 than algorithms 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

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 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. The idea here is to speculate that a race doesn’t occur, detect it, and retry if it does.

Doing without locks

Reader:

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

Writer:

```
spinlock(&lck);
seq++; wmb()
writer body ...
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.

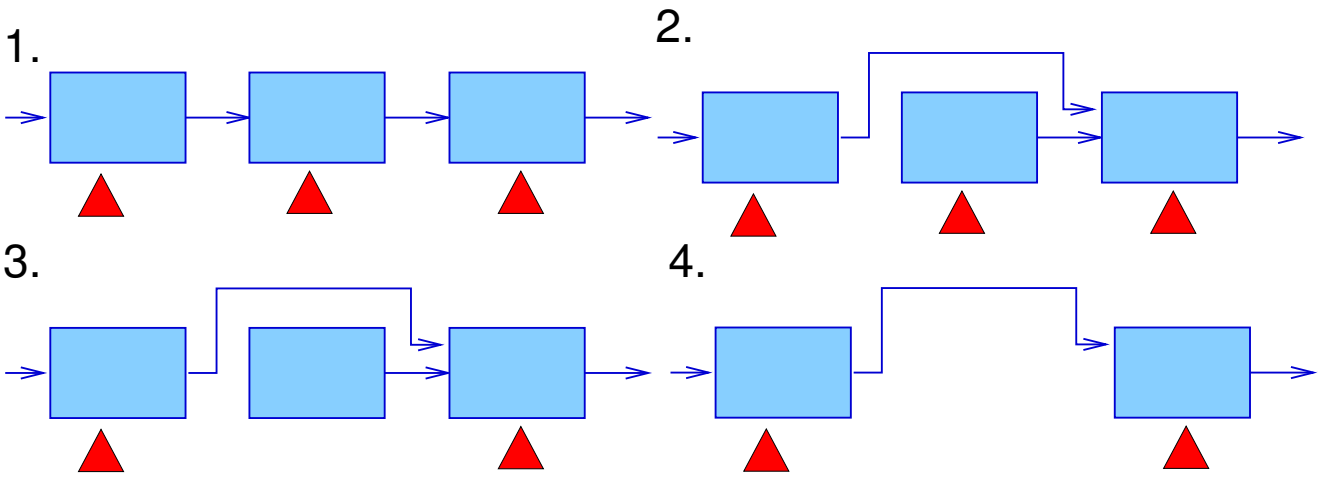
The reader grabs a copy of the sequence at the start of its section, spinning if the result is odd. At the end of the section, it rereads the sequence, if it is different from the first read value, the section is repeated.

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 variables containing 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.

References

Baumann, A., Appavoo, J., Krieger, O. & Roscoe, T. (2019), 'A `fork()` in the road', *Workshop on Hot Topics in Operating Systems (HotOs '19)* .

Lepers, B., Quema, V. & Fedorova, A. (2015), Thread and memory placement on NUMA systems: Asymmetry matters, *in* '2015 USENIX Annual Technical Conference (USENIX ATC 15)', USENIX Association, Santa Clara, CA, pp. 277–289.
URL: *https://www.usenix.org/conference/atc15/technical-sessio*

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/RCUdissertation.2*

McKenney, P. E. (2010), 'Memory barriers: A hardware view for software hackers', Online article, retrieved July 2019.
URL: *http://www.rdrop.com/users/paulmck/scalability/paper/why*

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/rcu.2002.07.08*

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

Doing without locks

References

Baumann, A., Appavoo, J., Krieger, O. & Roscoe, T. (2019), 'A `fork()` in the road', *Workshop on Hot Topics in Operating Systems (HotOs '19)* .

Lepers, B., Quema, V. & Fedorova, A. (2015), Thread and memory placement on NUMA systems: Asymmetry matters, *in* '2015 USENIX Annual Technical Conference (USENIX ATC 15)', USENIX Association, Santa Clara, CA, pp. 277–289.
URL:
https://www.usenix.org/conference/atc15/technical-ses

McKenney, P. E. (2004), Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels

Doing without locks

PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University.
URL:
http://www.rdrop.com/users/paulmck/RCU/RCUdissertatic

McKenney, P. E. (2010), 'Memory barriers: A hardware view for software hackers', Online article, retrieved July 2019.
URL:
http://www.rdrop.com/users/paulmck/scalability/paper/

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/rcu.2002.07

Ritchie, D. M. (1984), 'The evolution of the UNIX time-sharing system'

Doing without locks

AT&T Bell Laboratories Technical Journal **63**(8), 1577–1593.
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.