NICTA

LINUX INTERNALS

Peter Chubb and Etienne Le Sueur
first.last@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

- 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. (Ritchie 1984)
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

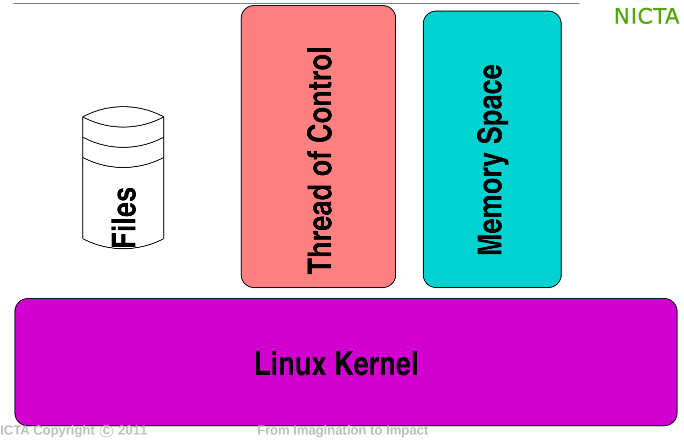
licencees were not able to use the book legally any more. UNSW obtained an exemption of some sort; but the upshot was that the Lions book was copied and copied and studied around the world. However, the licence change also meant that an alternative was needed for OS courses.
Many universities stopped 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

- 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()

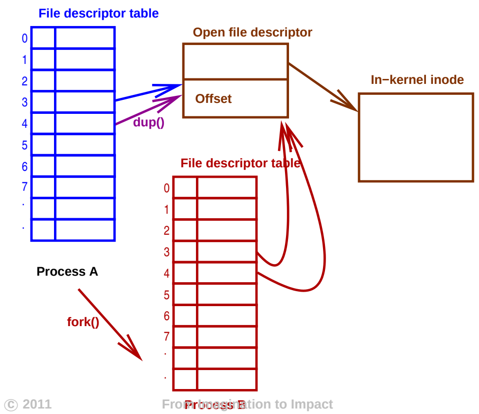
NICTA

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

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 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()

NICTA



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()

NICTA

```
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

NICTA

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

NICTA

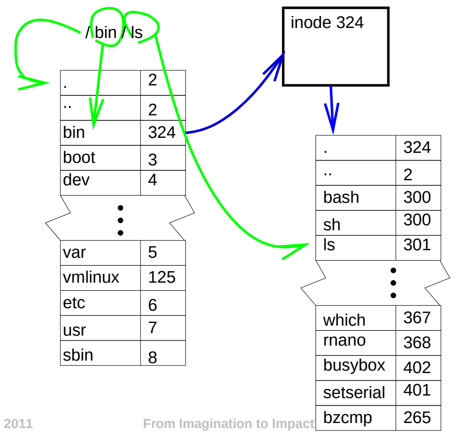
- Separation of names from content.
- 'regular' files 'just bytes' $\rightarrow$ structure/meaning supplied by userspace
- Devices represented by files.
- Directories map names to index node indices (`inums`)
- Simple permissions model

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 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

NICTA



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

NICTA

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

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]`.

C Dialect

NICTA

- 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

NICTA

- 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

Goals:

- O(1) in number of runnable processes, number of processors
- '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.
Becasue 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. 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.
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:

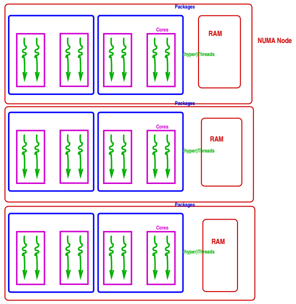
- Avoiding overflow
- Keeping recent history
- multiprocessor locality
- handling too-many threads
- Sleeping tasks

SCHEDULING

- 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 an infinitely 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.

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

NICTA

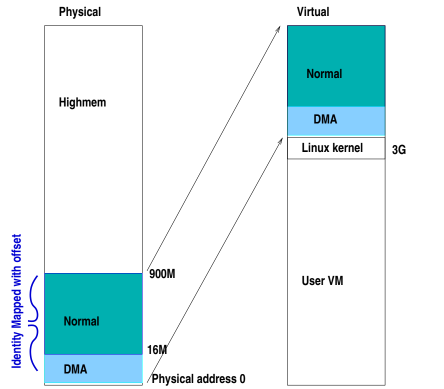
- One queue per processor (or hyperthread)
- Processors in hierarchical ‘domains’
- Load balancing per-domain, bottom up
- Aims to keep whole domains idle if possible (power savings)

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

NICTA

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 virtual address space to be allocated to userspace, but this comes with a performance hit.

MEMORY MANAGEMENT

NICTA

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

MEMORY MANAGEMENT

NICTA

`struct page`:

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

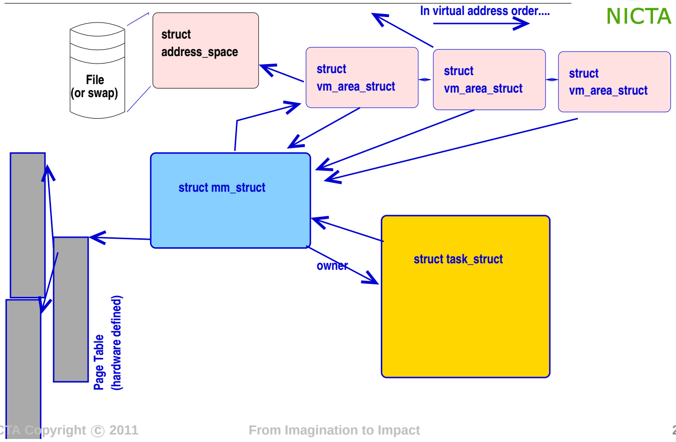
Direct mapped pages can be referred to by *logical addresses*; there are a simple pair of macros for converting between physical and logical addresses for these. Anything not mapped must be referred to by a **`struct page`** and an offset within the page. There is a **`struct page`** for every physical 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).

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

NICTA



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

  migratepage Move pages between NUMA nodes

Memory Management

NICTA

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

NICTA

- 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

NICTA

`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

NICTA

Three kinds of device:

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

Driver Interface

NICTA

Enumerable buses:

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


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

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.

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.

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

NICTA

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

NICTA

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

NICTA

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

Background Reading

NICTA

References

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

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