Linux, Locking and Lots of Processors

Peter Chubb

peter.chubb@nicta.com.au
A LITTLE BIT OF HISTORY

- Multix in the ’60s
A LITTLE BIT OF HISTORY

- Multix in the ’60s
- Ken Thompson and Dennis Ritchie in 1967–70
A LITTLE BIT OF HISTORY

- Multix in the ’60s
- Ken Thompson and Dennis Ritchie in 1967–70
- USG and BSD
A LITTLE BIT OF HISTORY

• Multix in the ’60s
• Ken Thompson and Dennis Ritchie in 1967–70
• USG and BSD
• John Lions 1976–95
A LITTLE BIT OF HISTORY

- Multix in the ’60s
- Ken Thompson and Dennis Ritchie in 1967–70
- USG and BSD
- John Lions 1976–95
- Andrew Tanenbaum 1987
A LITTLE BIT OF HISTORY

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

- Basic concepts well established
  - Process model
  - File system model
  - IPC
A LITTLE BIT OF HISTORY

• Basic concepts well established
  – Process model
  – File system model
  – IPC
• Additions:
  – Paged virtual memory (3BSD, 1979)
A LITTLE BIT OF HISTORY

• Basic concepts well established
  – Process model
  – File system model
  – IPC
• Additions:
  – Paged virtual memory (3BSD, 1979)
  – TCP/IP Networking (BSD 4.1, 1983)
A LITTLE BIT OF HISTORY

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

Linux Kernel

Files

Thread of Control

Memory Space
Process model

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

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

Files and Processes:

File descriptor table

0
1
2
3
4
5
6
7
...

Process A
Files and Processes:

File descriptor table

Open file descriptor

Offset

In-kernel inode

Process A
Files and Processes:

```
<table>
<thead>
<tr>
<th>File descriptor table</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>·</td>
</tr>
<tr>
<td>·</td>
</tr>
</tbody>
</table>
```

Process A

Open file descriptor

Offset

In-kernel inode

\( \text{dup()} \)
**FORK() AND EXEC()**

**Files and Processes:**

```
0 1 2 3 4 5 6 7
..  
```

File descriptor table

```
File descriptor table
Process A
fork()
```

```
Open file descriptor
Offset
In-kernel inode
```

```
File descriptor table
Process B
dup()
```

NICTA Copyright © 2014 From Imagination to Impact
FORK() AND EXEC()

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

0 Standard Input
1 Standard Output
2 Standard Error

➔ Inherited from parent
➔ On login, all are set to controlling tty
FILE MODEL

- Separation of names from content.
- ‘regular’ files ‘just bytes’ → structure/meaning supplied by userspace
- Devices represented by files.
- Directories map names to index node indices (inums)
- Simple permissions model
NAMEI

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

**KISS:**

→ Simplest possible algorithm used at first
EVOLUTION

KISS:

→ Simplest possible algorithm used at first
→ Easy to show correctness
→ Fast to implement
KISS:

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

• Extra keywords:
  – Section IDs: __init, __exit, __percpu etc
  – Info Taint annotation __user, __rcu, __kernel, __iomem
  – Locking annotations __acquires(X), __releases(x)
  – extra typechecking (endián portability) __bitwise
C DIALECT

- Extra iterators
  - type_name_foreach()

- Extra accessors
  - container_of()
C DIALECT

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

Goals:

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

Implementation:

• Changes from time to time.

• Currently ‘CFS’ by Ingo Molnar.
Dual Entitlement Scheduler

Running

Expired

0.5 0.7 0.1

0 0

NICTA Copyright © 2014 From Imagination to Impact
SCHEDULING

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

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

Devil’s in the details:

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

(hyper)Thread
SCHEDULING
SCHEDULING

Packages

Cores

(hyper)Threads
SCHEDULING
SCHEDULING

Locality Issues:

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

Locality Issues:

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

Locality Issues:

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

Locality Issues:

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

- Try to keep whole sockets idle

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

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

Memory in zones

Physical
- Highmem
- Normal
- DMA

Virtual
- Normal
- DMA
- Linux kernel
- User VM

Identity Mapped with offset

Physical address 0
900M
16M

Virtual address 0
3G

NICTA Copyright © 2014
From Imagination to Impact
MEMORY MANAGEMENT

- Direct mapped pages become *logical addresses*
  - \texttt{pa()} and \texttt{va()} convert physical to virtual for these
MEMORY MANAGEMENT

- Direct mapped pages become \textit{logical addresses}
  - \texttt{\_\_pa()} and \texttt{\_\_va()} convert physical to virtual for these
- Small memory systems have all memory as logical
MEMORY MANAGEMENT

- Direct mapped pages become *logical addresses*
  - \( \text{__pa()} \) and \( \text{__va()} \) convert physical to virtual for these

- Small memory systems have all memory as logical

- More memory \( \rightarrow \) \( \Delta \) kernel refer to memory by \text{struct page}
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)
MEMORY MANAGEMENT

File (or swap) -> struct address_space

Page Table (hardware defined) -> struct mm_struct

struct mm_struct -> struct vm_area_struct -> struct vm_area_struct -> struct vm_area_struct

In virtual address order....

struct task_struct
Memory Management

Address Space:

- Misnamed: means collection of pages mapped from the same object
- Tracks inode mapped from, radix tree of pages in mapping
- Has ops (from file system or swap manager) to:
  - **dirty** mark a page as dirty
  - **readpages** populate frames from backing store
  - **writepages** Clean pages — make backing store the same as in-memory copy
MEMORY MANAGEMENT

migratepage Move pages between NUMA nodes

Others... And other housekeeping
PAGE FAULT TIME

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

handle_pTE_fault(): Depending on PTE status, can

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

and if necessary flushes the TLB.
**DRIVER INTERFACE**

Three kinds of device:

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

Enumerable buses:

```c
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),
    { },
    
};

MODULE_DEVICE_TABLE(pci, cp_pci_tbl);
```
Driver Interface

Driver interface:

init called to register driver

exit called to deregister driver, at module unload time

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

open, close, etc as necessary for driver class
**Driver Interface**

Platform Devices:

```c
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,
};
```
non-enumerable buses: Treat like platform devices
SUMMARY

- I’ve told you status today
SUMMARY

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

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

- I’ve simplified a lot. There are many hairy details
I’m assuming:

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

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

- NOR Flash
- NAND Flash
FLASH MEMORY

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

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

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

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

NAND Flash Chip

Erase Block

Page

Interface Circuitry
FLASH UPDATE
FLASH UPDATE

[Diagram with arrows and colored boxes]
FLASH UPDATE
FLASH UPDATE
FLASH UPDATE

NICTA
FLASH UPDATE

RAM

NOR flash or EEPROM

Processor

NAND Flash Chip

Erase Block

Page

Interface Circuitry
THE CONTROLLER:

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

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

Mostly documented in Korean patents referred to by US patents!
Wear Management

Two ways:

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

Two ways:

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

Two ways:

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

In practice both are used.
Wear Management

Two ways:

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

• Spread writes over all erase blocks (wear levelling)

In practice both are used.

Also:

• Count reads and schedule garbage collection after some threshold
• Typically use FAT32 (or exFAT for sdxc cards)

• Always do cluster-size I/O (64k)

• First partition segment-aligned
Typically use FAT32 (or exFAT for sdxc cards)

Always do cluster-size I/O (64k)

First partition segment-aligned

Conjecture Flash controller optimises for the preformatted FAT fs
Conjecture The controller has some number of buffers it treats specially, to allow more than one write locus.
TESTING SDHC CARDS
# SD Card Characteristics

<table>
<thead>
<tr>
<th>Card</th>
<th>Price/G</th>
<th>#AU</th>
<th>Page size</th>
<th>Erase Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kingston Class 10</td>
<td>$0.80</td>
<td>2</td>
<td>128k</td>
<td>4M</td>
</tr>
<tr>
<td>Toshiba Class 10</td>
<td>$1.20</td>
<td>2</td>
<td>64k</td>
<td>8M</td>
</tr>
<tr>
<td>SanDisk Extreme UHS-1</td>
<td>$5.00</td>
<td>9</td>
<td>64k</td>
<td>8M</td>
</tr>
<tr>
<td>SanDisk Extreme Pro</td>
<td>$6.50</td>
<td>9</td>
<td>16k</td>
<td>4M</td>
</tr>
</tbody>
</table>
WRITE PATTERNS: FILE CREATE

(On Toshiba Exceria card)
WRITE PATTERNS: FILE CREATE

"~/iozone.dat" using 1:2
F2FS

- By Samsung
F2FS

• By Samsung

• ‘Use on-card FTL, rather than work against it’
F2FS

- By Samsung
- ‘Use on-card FTL, rather than work against it’
- Cooperate with garbage collection
F2FS

- By Samsung
- ‘Use on-card FTL, rather than work against it’
- Cooperate with garbage collection
- Use FAT32 optimisations
F2FS

- 2M Segments written as whole chunks
F2FS

- 2M Segments written as whole chunks — always writes at log head
F2FS

- 2M Segments written as whole chunks — always writes at log head — aligned with FLASH allocation units
F2FS

- 2M Segments written as whole chunks — always writes at log head — aligned with FLASH allocation units
- Log is the only data structure on-disk
F2FS

- 2M Segments written as whole chunks — always writes at log head — aligned with FLASH allocation units
- Log is the only data structure on-disk
- Metadata (e.g., head of log) written to FAT area in single-block writes
F2FS

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

• Log is the only data structure on-disk

• Metadata (e.g., head of log) written to FAT area in single-block writes

• Splits Hot and Cold data and Inodes.
BENCHMARKS: POSTMARK 32k READ

Kingston
Toshiba
Sandisk Extreme
SanDisk Extreme Pro

Filesystem

MB/s

FAT32
EXT4
F2FS

0
1
2
3
4
5
USING NON-F2FS

- Observation: XFS and ext4 already understand RAID
Observation: XFS and ext4 already understand RAID
RAID has multiple chunks, and a fixed stride, so...
USING non-f2fs

- Observation: XFS and ext4 already understand RAID
- RAID has multiple chunks, and a fixed stride, so...
- Configure FS as if for RAID
Using non-f2fs

Still running benchmarks, see LCA talk next January for results!
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
Amdahl’s law:
If a process can be split such that \( \sigma \) of the running time cannot be sped up, but the rest is sped up by running on \( p \) processors, then overall speedup is

\[
\frac{p}{1 + \sigma(p - 1)}
\]
SCALABILITY

Throughput vs. Applied load

1 processor
SCALABILITY

Throughput

1 processor

Applied load
SCALABILITY

Throughput

1 processor

Applied load
SCALABILITY

Throughput

Applied load

1 processor
**SCALABILITY**

![Graph showing scalability with applied load and throughput. The graph illustrates the relationship between the number of processors and the throughput. For 1, 2, and 3 processors, the graph shows a linear increase in throughput with an increase in applied load.](image)
SCALABILITY

Throughput

Applied load

1 processor

2 processors

3 processors
SCALABILITY

Throughput

Applied load

1 processor

2 processors

3 processors
**SCALABILITY**

![Graph showing throughput and latency with applied load for 2 and 3 processors.]
Gunther’s law:

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

where:

- \( N \) is demand
- \( \alpha \) is the amount of serialisation: represents Amdahl’s law
- \( \beta \) is the coherency delay in the system.
- \( C \) is Capacity or Throughput
SCALABILITY

\[ \alpha = 0, \beta = 0 \]
**Scalability**

\[\alpha = 0, \beta = 0\]

\[\alpha > 0, \beta = 0\]
**SCALABILITY**

\[ \alpha = 0, \beta = 0 \]

\[ \alpha > 0, \beta = 0 \]

\[ \alpha > 0, \beta > 0 \]
Queueing Models:

Poisson arrivals → Queue → Server

Poisson service times
SCALABILITY

Real examples:

Throughput vs. Load for Postgres TPC throughput.
SCALABILITY

Throughput vs Load graph showing USL with alpha=0.342101, beta=0.017430 and Postgres TPC throughput.

NICTA Copyright © 2014 From Imagination to Impact
SCALABILITY

Postgres TPC throughput, separate log disc
SCALABILITY

Another example:

![Graph showing scalability with number of clients and jobs per minute.](image)

- 01-way
- 02-way
- 04-way
- 08-way
- 12-way

Jobs per Minute vs. Number of Clients
## Scalability

<table>
<thead>
<tr>
<th>Spinlocks</th>
<th>Hold</th>
<th>Wait</th>
</tr>
</thead>
<tbody>
<tr>
<td>Util</td>
<td>Con</td>
<td>Mean( Max )</td>
</tr>
<tr>
<td>72.3%</td>
<td>13.1%</td>
<td>0.5us(9.5us)</td>
</tr>
<tr>
<td>0.01%</td>
<td>85.3%</td>
<td>1.7us(6.2us)</td>
</tr>
</tbody>
</table>
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);
            }
        }
}

NICTA Copyright © 2014 From Imagination to Impact
SCALABILITY

Number of Clients

Jobs per Minute

01-way
02-way
04-way
08-way
12-way
16-way

NICTA Copyright © 2014 From Imagination to Impact
TACKLING SCALABILITY PROBLEMS

- Find the bottleneck
TACKLING SCALABILITY PROBLEMS

- Find the bottleneck
  - not always easy
TACKLING SCALABILITY PROBLEMS

- Find the bottleneck
- fix or work around it
TACKLING SCALABILITY PROBLEMS

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

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

• Find the bottleneck
• fix or work around it
• check performance doesn’t suffer too much on the low end.
• Experiment with different algorithms, parameters
TACKLING SCALABILITY PROBLEMS

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

- Each solved problem uncovers another
- Fixing performance for one workload can worsen another
- Performance problems can make you cry
DOING WITHOUT LOCKS

Avoiding Serialisation:

- *Lock-free* algorithms

- Allow safe concurrent access *without excessive* serialisation
DOING WITHOUT LOCKS

Avoiding Serialisation:

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

Sequence locks:

- Readers don’t lock
- Writers serialised.
DOING WITHOUT LOCKS

Reader:

volatile seq;
do {
    do {
        lastseq = seq;
    } while (lastseq & 1);
rmb();
....
} while (lastseq != seq);
DOING WITHOUT LOCKS

Writer:

spinlock(&lck);
seq++; wmb()

... 

wmb(); seq++;
spinunlock(&lck);
DOING WITHOUT LOCKS

RCU: ??

1.
DOING WITHOUT LOCKS

RCU: ??

1. 

2. 

NICTA Copyright © 2014 From Imagination to Impact
DOING WITHOUT LOCKS

RCU: ??

1. 

2. 

3.
DOING WITHOUT LOCKS

RCU: ??

1. 

2. 

3. 

4.
References


BACKGROUND READING

‘Ottawa Linux Symp.’.

URL:

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