Linux, Locking and Lots of Processors

Peter Chubb

peter.chubb@data61.csiro.au

July 15, 2019
A little bit of history

- MULTICS in the ’60s
A little bit of history

- MULTICS in the ’60s
- Ken Thompson and Dennis Ritchie in 1967–70
A little bit of history

- MULTICS in the ’60s
- Ken Thompson and Dennis Ritchie in 1967–70
- USG and BSD
A little bit of history

• MULTICS in the ’60s
• Ken Thompson and Dennis Ritchie in 1967–70
• USG and BSD
• John Lions 1976–95
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
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
A little bit of history

- Basic concepts well established
  - Process model
  - File system model
  - IPC
A little bit of history

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

- Additions:
  - Paged virtual memory (3BSD, 1979)
A little bit of history

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

- Additions:
  - Paged virtual memory (3BSD, 1979)
  - TCP/IP Networking (BSD 4.1, 1983)
A little bit of history

• Basic concepts well established
  – Process model
  – File system model
  – IPC

• Additions:
  – Paged virtual memory (3BSD, 1979)
  – TCP/IP Networking (BSD 4.1, 1983)
  – Multiprocessing (Vendor Unices such as Sequent’s ‘Balance’, 1984)
Abstractions
Process model

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

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

→ 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

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Process A
fork() and exec()

Files and Processes:

File descriptor table

Open file descriptor

In-kernel inode

Process A

Copyright © 2019 Linux, Locking and Lots of Processors 7
fork() and exec()

Files and Processes:

File descriptor table

0 1 2 3 4 5 6 7 ...

Open file descriptor

Offset

In-kernel inode

dup()
fork() and exec()
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
The problem with `fork()`

- Almost perfect in original system
  - Implemented in a few lines of assembly
  - Allowed re-use of system calls for changing state
  - Fast for segment-style (not paged) MMU
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
Permissions Model

- Based on logged-in-users
- UID, GID, Other — rwx
- Mainly for File access.
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 ($\text{inums}$)
- Simple permissions model based on who you are.
## File model

The file system is represented as a tree structure. Each directory is a node that contains a list of files and subdirectories. The `ls` command is used to list the contents of a directory.

### inode 324

- **.`**: 324
- `..`: 2
- `bash`: 300
- `sh`: 300
- `ls`: 301
- `which`: 367
- `rnano`: 368
- `busybox`: 402
- `setserial`: 401
- `bzcmp`: 265

### Node: `bin` (inode 324)

- **.`**: 2
- `..`: 2
- `bin`: 324
- `boot`: 3
- `dev`: 4

### Directory Contents:

- `var`: 5
- `vmlinux`: 125
- `etc`: 6
- `usr`: 7
- `sbin`: 8

Copyright © 2019 Linux, Locking and Lots of Processors
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
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
Evolution

KISS:

→ Simplest possible algorithm used at first
Evolution

KISS:

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

**KISS:**

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

→ As drawbacks and bottlenecks are found, replace with faster/more scalable alternatives
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
Linux C Dialect

• Extra iterators
  – \texttt{type\_name\_foreach()}

• Extra O-O accessors
  – \texttt{container\_of()}

• Macros to register Object initialisers
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.
Internal Abstractions

→ MMU
→ Memory consistency model
→ Device model
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.
Scheduling

Implementation:

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

Dual Entitlement Scheduler

Running

0.5 — 0.7 — 0.1

Expired

0 — 0
Scheduling

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

(hyper)Thread
Scheduling

Core
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 (can power them off)
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’?
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

3G

User VM

Physical address 0

900M

16M

Identity Mapped with offset

Copyright © 2019 Linux, Locking and Lots of Processors
Memory Management

- Direct mapped pages become *logical addresses*
  - \( \_\_p\text{a}(\_\_\_\_\_\_\_) \) and \( \_\_v\text{a}(\_\_\_\_\_\_) \) convert physical to virtual for these
Memory Management

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

- small memory systems have all memory as logical
Memory Management

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

**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

- struct address_space
- struct vm_area_struct
- struct mm_struct
- struct task_struct
- File (or swap)
- Page Table (hardware defined)

In virtual address order....

Copyright © 2019 Linux, Locking and Lots of Processors
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
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_IDREALTEK_8139), },
    { PCI DEVICE (PCI_VENDOR_ID_TTTECH,
        PCI_DEVICE_ID_TTTECH_MC322), },
    { },
};
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 (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,
};
Driver Interface

non-enumerable buses: Treat like platform devices
Device Tree

- Describe board+peripherals
Device Tree

- Describe board+peripherals
  - replaces ACPI on embedded systems
Device Tree

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

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

- *Namespace* isolation
Containers

- *Namespace* isolation
- Plus Memory and CPU isolation
Containers

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

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

_In hierarchy of control groups_
Containers

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

*In hierarchy of control groups*

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

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

1 processor

Applied load
Scalability

Throughput

1 processor

Applied load
Scalability

Throughput

Applied load

1 processor
Scalability

Throughput

Applied load

1 processor
Scalability

Throughput

Applied load

1 processor

2 processors

3 processors
Scalability

Throughput vs. Applied load graph showing:
- 1 processor
- 2 processors
- 3 processors
Scalability

Throughput

Applied load

1 processor

2 processors

3 processors
Scalability

Applied load vs. Throughput and Latency for 2 and 3 processors.
Scalability

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

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

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

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

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

Queueing Models:

Poisson arrivals → Queue → Server

Poisson service times
Scalability

Queueing Models:

- Poisson arrivals
- Queue
- Server with Poisson service times
- Sink
- High Priority
- Normal Priority
- Same Server
Scalability

Real examples:

![Graph showing Postgres TPC throughput vs load]

Throughput

Load
Scalability

Postgres TPC throughput, separate log disc

Copyright © 2019 Linux, Locking and Lots of Processors
Scalability

Another example:

![Graph showing scalability](image-url)

- Jobs per Minute vs. Number of Clients for 01-way, 02-way, 04-way, 08-way, and 12-way configurations.
### 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>
Scalability

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

The graph illustrates the scalability of different types of systems (01-way, 02-way, 04-way, 08-way, 12-way, 16-way) in terms of jobs per minute as the number of clients increases. The x-axis represents the number of clients, while the y-axis shows the jobs per minute. The graph shows how the performance of these systems scales with the number of clients, indicating varying levels of scalability and efficiency.
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);

Writer:

spinlock(&lck);
seq++; wmb()
writer body ...
wmb(); seq++;
spinunlock(&lck);
Doing without locks


1.
Doing without locks

Doing without locks

Doing without locks


1. 

2. 

3. 

4.
Doing without locks

References


URL:


URL: