Linux, Locking and Lots of Processors

Peter Chubb
Senior Consultant
peter.chubb@unsw.edu.au
21 June, 2022
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
— Basic concepts well established
  ○ User model
  ○ Process model
  ○ File system model
  ○ IPC — pipes, MERT
— Basic concepts well established
  ○ User model
  ○ Process model
  ○ File system model
  ○ IPC — pipes, MERT

— Additions:
  ○ Paged virtual memory (3BSD, 1979)
— Basic concepts well established
  ○ User model
  ○ Process model
  ○ File system model
  ○ IPC — pipes, MERT

— Additions:
  ○ Paged virtual memory (3BSD, 1979)
  ○ TCP/IP Networking (BSD 4.1, 1983)
— Basic concepts well established
  ◦ User model
  ◦ Process model
  ◦ File system model
  ◦ IPC — pipes, MERT

— 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
- 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
<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>
</tbody>
</table>

Process A
Open file descriptor
Offset
In-kernel inode
File descriptor table
Process A
File descriptor table

Open file descriptor

Offset

In-kernel inode

Process A

dup()
File descriptor table

Process A

fork()

dup()

Open file descriptor

Offset

In-kernel inode

File descriptor table

Process B
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
  ○ Alowed re-use of system calls for changing state
  ○ Fast for segment-style (not paged) MMU

— 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

— Processes are proxies for authenticated real people
— UID, GID, Other — rwx
— Mainly for File access.
— A process can signal any other process with the same UID
Permissions Model

— Processes are proxies for authenticated real people
— UID, GID, Other — rwx
— Mainly for File access.
— A process can signal any other process with the same UID
— A process with UID 0 can signal any process, operate on any file*

* Conditions apply
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.
<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>bin</td>
<td>324</td>
<td>324</td>
<td>324</td>
</tr>
<tr>
<td>boot</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>dev</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>var</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>vmlinux</td>
<td>125</td>
<td>125</td>
<td>125</td>
</tr>
<tr>
<td>etc</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>usr</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>sbin</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

inode 324

<table>
<thead>
<tr>
<th></th>
<th>324</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>..</td>
<td>300</td>
<td>300</td>
<td>300</td>
</tr>
<tr>
<td>bash</td>
<td>301</td>
<td>301</td>
<td>301</td>
</tr>
<tr>
<td>sh</td>
<td>300</td>
<td>300</td>
<td>300</td>
</tr>
<tr>
<td>ls</td>
<td>125</td>
<td>125</td>
<td>125</td>
</tr>
<tr>
<td>which</td>
<td>367</td>
<td>367</td>
<td>367</td>
</tr>
<tr>
<td>rnano</td>
<td>368</td>
<td>368</td>
<td>368</td>
</tr>
<tr>
<td>busybox</td>
<td>402</td>
<td>402</td>
<td>402</td>
</tr>
<tr>
<td>setserial</td>
<td>401</td>
<td>401</td>
<td>401</td>
</tr>
<tr>
<td>bzcmp</td>
<td>265</td>
<td>265</td>
<td>265</td>
</tr>
</tbody>
</table>
— 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 *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

— As drawbacks and bottlenecks are found, replace with faster/more scalable alternatives
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
— Extra iterators
  ◦ type_name_foreach()
— Extra O-O accessors
  ◦ container_of()
— Macros to register Object initialisers
— 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.
— Changes from time to time.
— Currently ‘CFS’ by Ingo Molnar.
Dual Entitlement Scheduler

Running

0.5 0.7 0.1

Expired

0 0
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
(hyper)Thread
Core
<table>
<thead>
<tr>
<th>NUMA Node</th>
<th>RAM</th>
<th>Packages</th>
<th>Cores</th>
<th>HyperThreads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Threads
- Packages
- Cores
- RAM
— Best to reschedule on same processor (don’t move cache footprint, keep memory close)
— Best to reschedule on same processor (don’t move cache footprint, keep memory close)
  ◦ Otherwise schedule on a ‘nearby’ processor
— 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)
— 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’?
— 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

3G

Physical address 0

16M

900M

Identity Mapped with offset

Identity Mapped with offset
— Direct mapped pages become *logical addresses*
  ◦ \texttt{pa()} and \texttt{va()} convert physical to virtual for these
— 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
— 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*
— 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

struct mm_struct

Page Table (hardware defined)

struct task_struct

struct mm_struct

In virtual address order....

struct vm_area_struct

struct vm_area_struct

struct vm_area_struct

owner
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

def 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:
A  enumerable-bus device
B  Non-enumerable-bus device
Driver Interface: Device Discovery
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);
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
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
uart_A: serial@84c0 {
    compatible = "amlogic,meson6-uart", "amlogic,meson-uart";
    reg = <0x84c0 0x18>;
    interrupts = <GIC_SPI 26 IRQ_TYPE_EDGE_RISING>;
    status = "ok";
};
Debugging device discovery

Add `debug_initcalls` to Linux boot args
— traces all calls to `init()` functions at boot time.
(See `Documentation/admin-guide/kernel-parameters.txt` in the linux kernel source for other useful boot args)
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
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 vs. Applied load for 1 processor.
Scalability

Throughput

Applied load

1 processor
Scalability

Throughput

Applied load

1 processor
Scalability

Throughput

Applied load

1 processor

2 processors

3 processors
Scalability

Throughput

Applied load

1 processor

2 processors

3 processors
Scalability

Throughput

Applied load

1 processor

2 processors

3 processors
Scalability

Throughput

Latency

Applied load

3 processors

2 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

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

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

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

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

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

Poisson arrivals → Queue → Server with Poisson service times → Sink

Same Server

High Priority

Normal Priority
Scalability

Queueing Models

Poisson arrivals

Queue

Server
Poisson service times

Sink

High Priority

Queue

Normal Priority

Server
Poisson service times

Same Server

Sink
Scalability

Real examples

Postgres TPC throughput
Scalability

USL with $\alpha=0.342101, \beta=0.017430$

Postgres TPC throughput
Scalability

Throughput

Postgres TPC throughput, separate log disc
Scalability

Another example

reAIM-7 on HP 16-way Itanium:
Scalability

Another example

reAIM-7 on HP
16-way Itanium: $\alpha$
huge; 12-way
curve below 8 way.
<table>
<thead>
<tr>
<th>SPINLOCKS</th>
<th>HOLD</th>
<th>WAIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>UTIL CON</td>
<td>MEAN( MAX )</td>
<td>MEAN( MAX )</td>
</tr>
<tr>
<td>72.3% 13.1% 0.5us(9.5us)</td>
<td>29us(20ms)(42.5%)</td>
<td>50542055</td>
</tr>
<tr>
<td>0.01% 85.3% 1.7us(6.2us)</td>
<td>46us(4016us)(0.01%)</td>
<td>1113</td>
</tr>
</tbody>
</table>
struct page *find_lock_page(struct address_space *mapping, 
    unsigned long offset)
{
    struct page *page;
    spin.lockIRQ(&mapping->tree.lock);
    repeat:
        page = radix_tree_lookup(&mapping->page_tree, offset);
        if (page) {
            page.cache_get(page);
            if (TestSetPageLocked(page)) {
                spin.unlockIRQ(&mapping->tree.lock);
                lock.page(page);
                spin.lockIRQ(&mapping->tree.lock);
            }
        }
    ...
}
Spin lock

RWlock

Note Scales!
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
  ○ 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
— Each solved problem uncovers another
— Fixing performance for one workload can worsen another
— 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)
— Readers don’t lock
— Writers serialised.
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);
McKenney (2004), McKenney et al. (2002)
McKenney (2004), McKenney et al. (2002)
McKenney (2004), McKenney et al. (2002)
RCU

McKenney (2004), McKenney et al. (2002)
Background Reading I


**URL:** http://www.rdrop.com/users/paulmck/rclock/rcu.2002.07.08.pdf