Multiprocessing in the Linux Kernel

Steve Muckle
October 15th, 2009

Sources/References

- ARM v7 ARM (Architecture Reference Manual)
  - A3.5: Memory Types and Attributes and the Memory Order Model
  - A3.8: Memory Access Order (covers barriers)
- Multiprocessing Overview by Thomas Sartorius
- Linux Device Drivers
  - Chapter 5: Concurrency and Race Conditions
- QC Multicore Multiprocessing Guide
  Agile 80-VR848-1

Overview

- Why We're Talking About This Now
- Overview of UP Systems and Issues
- Introduction to Multiprocessing
- MP Safe Programming
- Multiprocessing on ARM
- Designing for SMP in the Linux Kernel

Why We're Talking About This Now

- Presilicon development is essential
- Concept to volume ramp up has been around ~30months avg
- Want this to be ~24months
- 7x30 had full Android stack up less than 2 weeks after CoD
- We have to deal with SMP, which introduces a new class of issues and makes others worse
- Still hitting issues in 8x50

Overview of Uniprocessor Systems and Issues

Uniprocessor (UP) Systems

- Operating system instance runs on a single CPU
- At any given time, only one thread is executing
- There is a single linear CPU cache hierarchy to deal with
- This is easy, right?
- Wrong

Raptor2 (8x50)

- HLOS controls a single Scorpion core (UP)
- The modem processor (ARM9) and QDSP6 are not included as the HLOS does not schedule threads on these cores

Raptor2 (8x50)

- HLOS controls a single Scorpion core (UP)
- The modem processor (ARM9) and QDSP6 are not included as the HLOS does not schedule threads on these cores
- We are still fixing caching, synchronization issues

Concurrency Issues in UP Systems

- Preemption in both user and kernel space, sharing with interrupt context necessitates locking

thread 1		thread 2
	x = 3	
x = x + 1;		x = x + 1;
	x = ?	

- Outcome depends on compilation and interleaving of instructions between two threads

Data Coherency Issues in UP Systems

- Interaction with other bus masters may cause data coherency issues

CPU	DMA Engine
read DMA'd buffer;	
prepare new DMA op;	
initiate new DMA op to read data;	
wait;	perform DMA
read DMA'd buffer;	

Data Coherency Issues in UP Systems

- Interaction with other bus masters may cause data coherency issues

```
CPU                                   DMA Engine
read DMA'd buffer;
prepare new DMA op;
initiate new DMA op to read data;
wait;                                 perform DMA
read DMA'd buffer; /* stale data in cache */
```

Data Coherency Issues in UP Systems

- Interaction with other bus masters may cause data coherency issues

CPU	DMA Engine
read DMA'd buffer;	
use DMA'd buffer for scratch space;	
prepare new DMA op;	
initiate new DMA op to read data;	
wait;	perform DMA
read DMA'd buffer;	

Data Coherency Issues in UP Systems

- Interaction with other bus masters may cause data coherency issues

CPU DMA Engine
read DMA'd buffer;
use DMA'd buffer for scratch space;
prepare new DMA op;
initiate new DMA op to read data;
wait; perform DMA
during wait, dirty cache line is evicted!
read DMA'd buffer; /* corrupted buffer */

Data Coherency Issues in UP Systems

- Interaction with other bus masters may cause data coherency issues

CPU	DMA Engine
read DMA'd buffer;	
use DMA'd buffer for scratch space;	
cache invalidate DMA'd buffer;	
prepare new DMA op;	
initiate new DMA op to read data;	
wait;	perform DMA
read DMA'd buffer;	

Data Coherency Issues in UP Systems

- Scorpion may speculatively fetch into the data cache
- This means cache lines may be filled without any code executing that touches that data
- What does this mean?

Speculative Fetch Example

CPU DMA Engine

read DMA'd buffer;

cache invalidate DMA'd buffer;

prepare new DMA op;

initiate new DMA op to read data;

wait; perform DMA

read DMA'd buffer;

Speculative Fetch Example

CPU DMA Engine
read DMA'd buffer;
cache invalidate DMA'd buffer;
prepare new DMA op;
initiate new DMA op to read data;
wait; perform DMA
during wait, speculative fetch loads
cache lines while DMA is in progress!
read DMA'd buffer;/* stale/corrupt cache */

Speculative Fetch Example

CPU DMA Engine
read DMA'd buffer;
cache invalidate DMA'd buffer;
prepare new DMA op;
initiate new DMA op to read data;
wait; perform DMA
cache invalidate DMA'd buffer;
read DMA'd buffer;

Observation of Memory Ops in UP Systems

- My driver maps HW registers via a normal memory mapping and does the following:

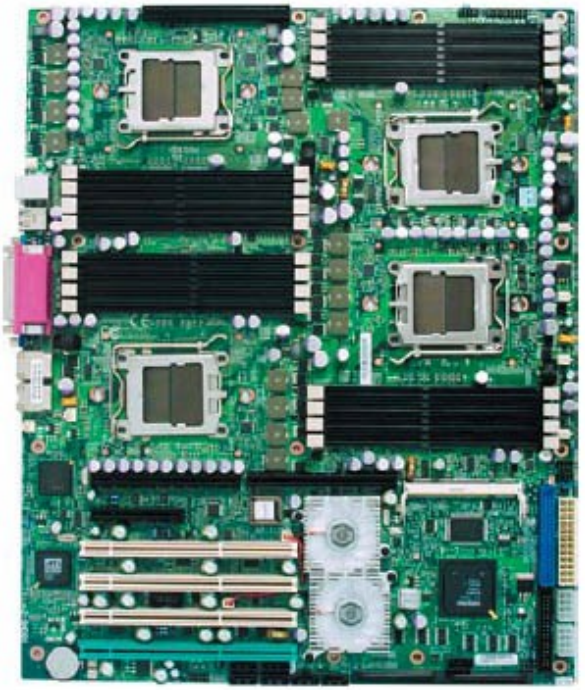
```
writeb(0x13, CONFIG_XACTION_REG);
writeb(0x77, CONFIG_BUFLEN_REG);
writeb(0x1, INITIATE_OP_REG);
```

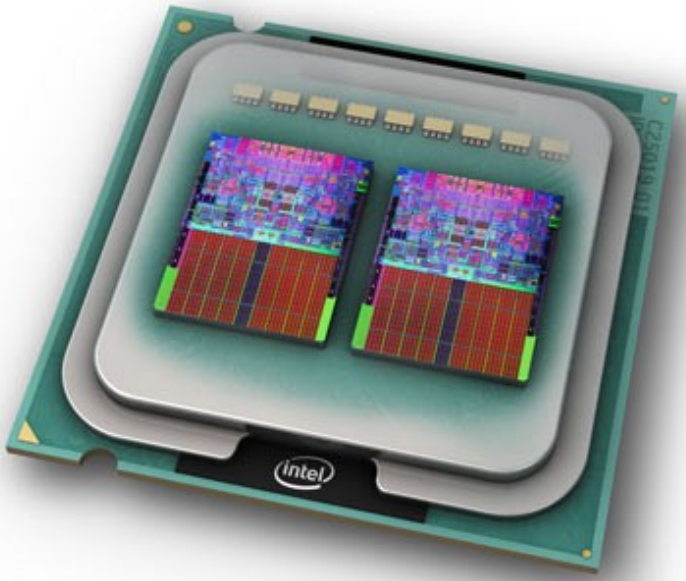
- The device is not architecturally obligated to observe the first two writes prior to the third. What does this mean?

Observation of Memory Ops in UP Systems

- Possible ordering of execution of memory ops:
  writeb(0x1, INITIATE_OP_REG);
  writeb(0x13, CONFIG_XACTION_REG);
  writeb(0x77, CONFIG_BUFLEN_REG);
- Transaction is initiated before it is configured!
- Solution: map this using a device mapped memory
- Covered later in more detail

Introduction to Multiprocessing





Let's Clear Up Some Terminology

What is the difference between multiprocessing (MP), SMP, ASMP, and “multicore?”

Multiprocessing (MP)

- Wikipedia:

  “The use of two or more CPUs within a single computer system.”

Symmetric Multiprocessing (SMP)

- All CPUs treated (roughly) equally by OS
- All CPUs connected to a single shared main memory
- One HLOS
- This is the *software* definition of SMP

Non-Uniform Memory Access (NUMA)

- Memory access time depends on memory location relative to CPU performing access
- For example, each CPU may have local memory, but be able to access memory near other CPUs as well
- Upscaling of SMP

Asymmetric Multiprocessing (ASMP)

- An MP system that assigns certain tasks only to certain processors
- Possibly different operating systems, instruction sets, etc
- Again, this is the *software* definition of ASMP

“Multicore” (processor)

- Clock speeds are hitting a wall
  - power consumption, heat dissipation
- A multicore processor is composed of two or more CPU cores and implements multiprocessing in a single package.
- Most commonly, multicore implies SMP

Multiprocessing is Not New

- Originated in mid 1950s, available since early 1960s (Burroughs D825 introduced in 1962)
- Intel 430NX Pentium chipset (March 1994) supported SMP
- Linux kernel 2.0 (June 1996) introduced SMP support
- Intel Pentium D Smithfield (dual die) launched in April 2005
- Intel Core Duo (single die) launched in June 2006

Blackbird (8x60) and Phantom (8x72)

- Blackbird will have a single high-level operating system (HLOS) controlling two 1.2Ghz Scorpions (homogenous SMP)
- Phantom will have a single HLOS controlling a 1.2Ghz Scorpion and a 1.5Ghz Scorpion (heterogeneous SMP)
- HLOS does not schedule threads on the modem processor (ARM11), DSP, RPM, or SPS

MP Safe Programming

General Guidelines

1. More than One Thing Is Going On!

- Just because the CPU is doing one thing, doesn't mean nothing else is happening
- N highest priority threads and ISRs are executing concurrently
- Threads could execute on any core
- In general, set affinity APIs should not be used (contact linux.multicore if you think this is needed)

2. Identifying Concurrency Issues

- Identify all mutable shared state
- Identify all critical sections
- Identify all threads/ISRs involved
- Use appropriate locking/sychronization
- Do not assume mundane operators are atomic

3. Deadlocks

- What are the ingredients of a deadlock?
  - mutual exclusion
  - hold and wait
  - no preemption
  - circular wait
- General methods for avoiding deadlocks?
  - strictly ordering locks
  - only hold one resource at a time

4. Reentrancy

- A function that can be entered multiple times concurrently or pseudo-concurrently is *reentrant*
- Must adhere to the following rules:
  - does not contain global mutable state or return the address of global mutable state
  - does not lock singleton resources, or takes steps to avoid deadlocks
  - does not call non-reentrant functions
- Example: system calls in drivers

5. Thread Priorities

- Do not rely on thread priorities as a method of mutual exclusion!
- Use correct OS primitives instead

6. Hardware Accesses

- Check for concurrent accesses to hardware

7. Beware the Weak Memory Model!

- This does not work!

Core 1	Core 2
update_data();	while (writeDone != 1);
writeDone = 1;	use_data();

- These accesses can be performed out of order
- Need barrier mechanisms, will cover in next section

8. Must DESIGN MP Safety

- MP safety/correctness is only possible by design
- Testing cannot guarantee MP correctness
- Most MP bugs are inherently racy and can be extremely difficult to reproduce
- Design and code reviews are of paramount importance

Multiprocessing on ARM

What architectural details of ARM should you be aware of with respect to multiprocessing?

Caveat: This is not exhaustive!

What is an observer?

What is reading or writing memory in an SoC?

Examples of Observers in 8x60/8x72

- Apps processor(s)
- Modem processor
- DSP(s)
- Datamover
- RPM
- SPS
- Also referred to as a master (as in bus master)

Coherency

- Consistency in the observation order of values in a *single* memory location
- All observers need not observe all values, but there must be a strict ordering to all values observed

Incoherent Memory Access Sequence

- X = 1
- P1 reads X, loading cache line (X = 1)
- P2 reads X, sees X = 1
- P2 writes X = 2, goes into memory
- P2 reads X again from memory, X = 2
- P3 reads X, sees X = 2
- P1 writes an adjacent value to cache
- P1 cleans cache line, X = 1
- P3 reads X, sees X = 1

Incoherent Memory Access Sequence

Where's the problem in this sequence?

Incoherent Memory Access Sequence

- X = 1
- P1 reads X, loading cache line (X = 1)
- P2 reads X, sees X = 1
- P2 writes X = 2, goes into memory
- P2 reads X again from memory, X = 2
- P3 reads X, sees X = 2
- P1 writes an adjacent value to cache
- P1 cleans cache line, X = 1
- P3 reads X, sees X = 1

Incoherent Memory Access Sequence

- Coherency is not being managed between the different masters
- This can be addressed either in SW or HW
- One possible solution – make this region noncacheable

Incoherent Memory Access Sequence

- X = 1
- P1 reads X (X = 1)
- P2 reads X, sees X = 1
- P2 writes X = 2, goes into memory
- P2 reads X again from memory, X = 2
- P3 reads X, sees X = 2
- P1 writes an adjacent value
- P1's adjacent access has no effect on X
- P3 reads X, sees X = 2

Cache Coherency Strategies

- Do Nothing
  - SW must enforce coherency itself
- Shareable → Noncacheable
  Scorpion UP does this, provides coherency to all masters
- Shareable → Writethrough
  - Must now do snoop-kills – i.e. CPU0 modifies a value in L1, it goes to L2, but must also invalidate value in CPU1 L1
  - No need to snoop on reads

Cache Coherency Strategies

- Full MESI snoop protocol
  - Modified, Exclusive, Shared, Invalid – four possible states of cache line
  - allows shared data to be copy-back cacheable
- ScorpionMP supports first three

Shareability Domains

- ARMv7 defines three s*hareability realms*
  - nonshareable: hardware coherency for uniprocessor only
  - inner shareable: hardware coherency for members of inner shareable realm
  - outer shareable: hardware coherency for inner and outer shareable realms
- Normal memory may be marked as belonging to one of these different realms

ScorpionMP Coherency

(this covers expected common case usage)

- Single shared L2
- Scorpion CPUs in inner shareability realm
- L1s are forced writethrough, snoop kills to maintain coherency (unless nonshareable)
- Normal application memory will be normal, cacheable, write back, write allocate, inner-shareable

Sequential Execution Model (SEM)

- Each instruction appears to be

  - fetched
  - executed
  - completed

  to all observers before the next instruction is fetched
- This is unfortunately not what you get on ARM

Weakly Ordered Memory Access Model

- Order of

  - memory access generation
  - observation by other masters
  - completion of access at memory/device

  may all be different

- SEM semantics maintained within CPU core

Types of Memory Mappings in ARMv6+

- Weakly ordered + device driver = bad
- The way memory is mapped influences the ordering of access to it
- There are three main types of memory mappings:

  - Normal Memory
  - Device Memory
  - Strongly-Ordered Memory

Normal Memory

- Used for memory which is idempotent:
  - repeated reads/writes have no side effects
  - accesses can be merged
  - prefetching allowed, etc.
- Accesses to normal memory areas are weakly ordered
- Most memory in the system is mapped this way
- Don't map devices using normal memory!

Normal Memory

- May be nonshareable, inner shareable, or outer shareable
  - this affects coherency management
- Also takes caching parameter:
  - write through cacheable
  - write back, write allocate cacheable
  - write back, no write allocate cacheable
  - non-cacheable

Device Memory and Strongly Ordered Memory

- Used to map areas which cause side effects when accessed
  - interrupt controller clearing an interrupt
  - DMA engine kicking off a transfer
- All accesses to DEV/SO memory occur at program size and are not repeated or combined.
- DEV/SO memory is not cacheable.

Device Memory and Strongly Ordered Memory

- DEV/SO accesses within the same 1KB aligned region follow program order (SEM)
- DEV/SO accesses to different 1KB aligned regions are weakly ordered relative to each other
- DEV/SO accesses are weakly ordered relative to normal memory accesses
- There is no architectural difference between device and strongly ordered

Ordering Weakly Ordered Accesses

- You may need to guarantee the order in which two memory accesses are observed
- Example:
  writel(0x1, EBI1_TURN_ON_REG);
  *(EBI1_ADDR) = 0xfeadbead;
- The second access to EBI1 won't work unless the first access has completed
- If the first access is device, and second is normal, these accesses are not guaranteed to be ordered properly

Ordering Weakly Ordered Accesses

- What gets printed?
- Initial conditions: x = 0; y = 0; z = 0;
- P1: x = 1
- P2: while (x != 1);
  y = 1;
- P3: while (y != 1);
  printf("x = %d\n", x);

Ordering Weakly Ordered Accesses

- Barriers deal with ordering of accesses between multiple memory locations.
- ARMv7 supports three barrier operations: DMB, DSB, and ISB
- CP15 ops on ARMv6, native instructions on ARMv7
- These shouldn't be used directly (will be wrapped in OS primitives) but you should know what they do

DMB, DSB

- Can operate on particular sharability domains or affect writes, or reads and writes
- We'll discuss the full system, read and write variant here

DMB

- Data Memory Barrier enforces global ordering
- Divides memory accesses into two groups.
- Once an observer sees a group 2 access, it is required to see all group 1 accesses
- Other observers may continue to not see any group 1 accesses (unless they have seen a group 2 access)
- Only affects ordering of memory accesses.

DMB

- Group 1:

  - Accesses by barrier-executing CPU before the barrier in program order

  - Accesses by other observers observed by the barrier-executing CPU before the barrier (directly or recursively)

DMB

- Group 2:

  - Accesses by barrier-executing CPU after the barrier in program order

  - Accesses by other observers which occurred after they observed a post-barrier access by the barrier-executing CPU

DMB Example

```
writel(0x1, EBI1_TURN_ON_REG);
*(EBI1_ADDR) = 0xfeadbead;
```

DMB Example

```
writel(0x1, EBI1_TURN_ON_REG);
dmb();
*(EBI1_ADDR) = 0xfeadbead;
```

DMB Example

- What gets printed?
- Initial conditions: x = 0; y = 0; z = 0;
- P1: x = 1
- P2: while (x != 1);
  y = 1;
- P3: while (y != 1);
  printf("x = %d\n", x);

DMB Example

- What gets printed?
- Initial conditions: x = 0; y = 0; z = 0;
- P1: x = 1
- P2: while (x != 1);
  dmb();
  y = 1;
- P3: while (y != 1);
  printf("x = %d\n", x);

DMB Example

- What gets printed?
- Initial conditions: x = 0; y = 0; z = 0;
- P1: x = 1
- P2: while (x != 1);
  dmb();
  y = 1;
- P3: while (y != 1);
  dmb();
  printf("x = %d\n", x);

DMB Example

- P1: STR R5, [R1]
  STR R0, [R2]
- P2:
  loop
  LDR R12, [R2]
  CMP R12, #1
  BNE loop
  AND R12, R12, #0
  LDR R5, [R1, R12]

DMB Example

- P1: STR R5, [R1]
  DMB
  STR R0, [R2]
- P2:
  loop
  LDR R12, [R2]
  CMP R12, #1
  BNE loop
  AND R12, R12, #0
  LDR R5, [R1, R12]

DMB Example

- P1: STR R5, [R1]
  DMB
  STR R0, [R2]
- P2:
  loop
  LDR R12, [R2]
  CMP R12, #1
  BNE loop
  AND R12, R12, #0
  LDR R5, [R1, R12]
- DMB not needed in P2 due to dependency

DSB

- Data synchronization barrier enforces global observation
- Divides accesses into two groups, like DMB

DSB

- DSB does not complete until all preceding cache, branch predictor, and TLB ops complete
- DSB does not complete until observed memory accesses before the DSB are observed by everyone
- Once any observer sees a post-DSB access, all observers must see all pre-DSB accesses

DSB Example

- DMA of cacheable memory:

```
fill_buffer(dma_buf);
cache_clean(dma_buf);
dsb();
dma_go(dma_buf);
```

ISB

- Flushes the pipeline of the CPU
- All subsequent instructions are fetched from cache/memory only when ISB has completed
- Ensures that *completed* cache, TLB, branch predictor, or CP14/CP15 ops are seen by instructions after the ISB
- You do not need an ISB between a cache op and an instruction that directly accesses a target address of that op on the same CPU

ISB Example

- mcr p15, 0, r2, c1, c0, 0   /* MMU on */
- isb
- msm_pm_mapped_pa:
- /* Switch to virtual */
- adr   r2, msm_pm_pa_to_va
- ldr   r0, =msm_pm_pa_to_va
- mov   pc, r0

Compiler Reordering

- The preceding issues are with ordering by the CPU at runtime
- This is a separate problem than compiler reordering
- We'll touch on the proper Linux kernel APIs for both of these a little later

SWP, SWPB

- Swaps a word/byte between registers and memory
- No accesses from other observers may occur between the load and store halves of the instruction
- Support basic busy/free mechanisms, but not mechanisms requiring calculation
- Locks the bus during transaction
- SWP and SWPB are deprecated, not supported on Scorpion

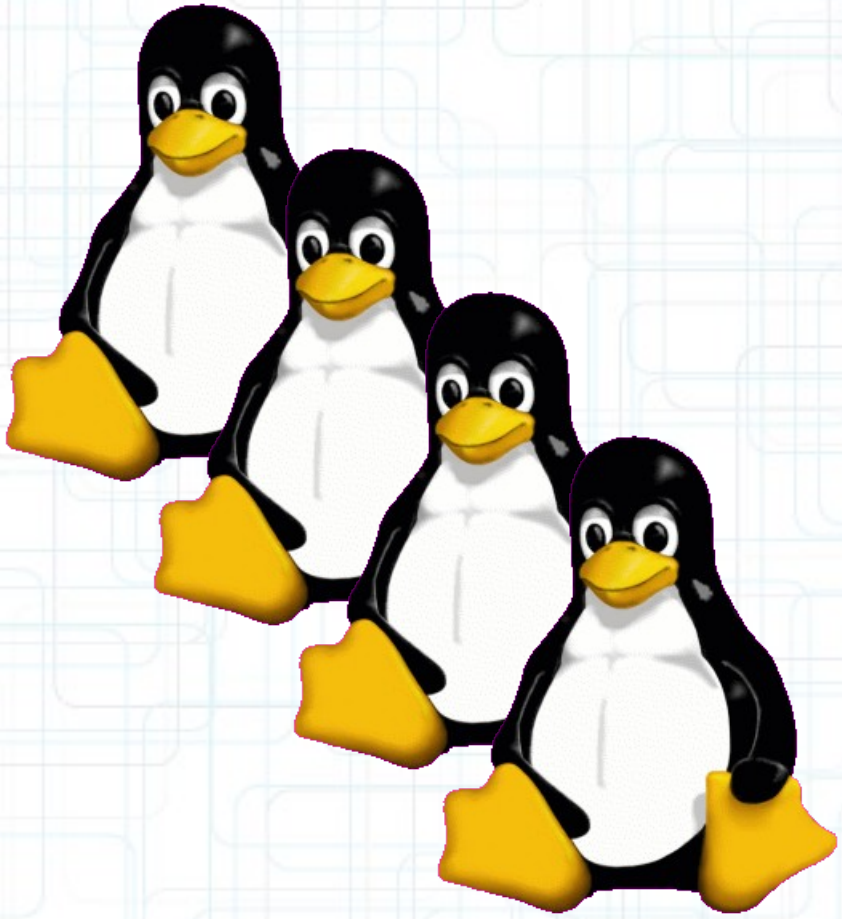
LDREX, STREX

- Load register exclusive, store register exclusive
- LDREX tags a memory location for exclusive access
- STREX only succeeds if the location is tagged by the CPU for exclusive access
- Modification to an address by another processor removes the exclusive access tag

WFE/SEV

- WFE causes the CPU to go into a lower power state
- It wakes up when:
  - another CPU sends an event using SEV
  - an IRQ or FIQ occurs, unless masked
  - an asynchronous abort occurs, unless masked
  - a debug event event occurs
- Used to improve power efficiency and reduce bus contention in SMP spinlocks

Designing for SMP in the Linux Kernel



Designing for SMP in the Linux Kernel - Agenda

- The “Good” News
- CONFIG_PREEMPT and CONFIG_SMP
- Concurrency primitives and APIs
- SMP scheduling
- Interrupt handling and Timers
- Testing

The “Good” News

- Your code *should* already be correct w.r.t SMP
- Code in the Linux kernel is not written specifically for UP or SMP
- It must be able to deal with CONFIG_PREEMPT or CONFIG_SMP
- Linux kernel code is expected to have proper concurrency management
- Linux Kernel has supported SMP since 1996

CONFIG_PREEMPT

- Introduced in 2.6 kernel (released late 2003)
- A thread can be descheduled while executing kernel code
- Kernel preemption reduces the dispatch latency of processes – i.e. how long it takes from runnable → running

CONFIG_PREEMPT=n CONFIG_SMP=n

- Isn't this easy
- Almost no concurrency issues in the kernel
- Just shared state with interrupts to worry about
- Note that CONFIG_PREEMPT refers to *kernel* preemption, not preemption in general

CONFIG_PREEMPT=y
CONFIG_SMP=n

- We now have pseudo-concurrency in the kernel, and the locking concerns that come with it
- This is the current configuration used on all Android and LE targets

CONFIG_PREEMPT=n CONFIG_SMP=y

- We now have multiple threads potentially executing in the kernel concurrently
- The fact that those threads can't be preempted in the kernel doesn't really help us much
- Only one thing is still safe – per-CPU variables (assuming they're not accessed from interrupt handlers)

CONFIG_PREEMPT=y
CONFIG_SMP=y

- Preemptive kernel on multiple CPUs
- What we will be running
- Nothing is safe

Concurrency Primitives and APIs

- Lockfree algorithms, operations
- Spinlocks
- Mutexes and Semaphores
- Per-CPU variables
- Memory and Compiler Barriers

Lockfree Algorithms

- Locks are tricky – can we just get away without them?
- Circular buffer (one writer, one reader)
- Read-copy-update
- Seqlocks
- Atomic variables, atomic bit operations

Atomic Operations

- i++ is not necessarily atomic!
- #include <asm/atomic.h>
- Eliminates the need for a separate lock if the shared resource is just a single integer value
- These rely on ldrex and strex for ARMv6 and beyond
  - SMP not supported for < ARMv6, these disable interrupts

Atomic Operations Example

```
atomic_t bufcount = ATOMIC_INIT(0);

void my_interrupt_handler(void)
{
   ...
  /*add buffers we processed to the total*/
  atomic_add(local_bufcount, &bufcount);
  ...
}
```

Atomic Bitops

- #include <linux/bitops.h>
- No data type, just takes a pointer to an unsigned long
- i.e.
  /* set 4th bit in 32-bit word */
  set_bit(4, &some_mask);

Spinlocks

- Can be held by at most one thread of execution
- Intended for SMP safety, although these are used in UP preemptive kernels as well
- Do different things depending on CONFIG_PREEMPT and CONFIG_SMP
- As a rule of thumb, use when you'll hold the lock for less than the cost of 2 context switches (or can't block)
- Not recursive!

CONFIG_PREEMPT=n
CONFIG_SMP=n

- Regular spinlocks compile out completely.
- What would happen if a thread went into the kernel and spun on a busy spinlock?

CONFIG_PREEMPT=y CONFIG_SMP=n

- Spinlock can simply disable kernel preemption while in the critical section
- Locking mechanism itself compiles out

CONFIG_PREEMPT=y
CONFIG_SMP=y

- Must disable kernel preemption, and actually lock a spinlock
- Kernel preemption is not disabled by the spinlock while waiting for the lock

Spinlock + interrupt

- What happens if a resource guarded by a spinlock is used in interrupt context?
- spin_lock_irqsave, spin_lock_irq
  - disables interrupts in the critical section
- spin_lock_bh leaves hw interrupts enabled, disables softirqs and tasklets

Spinlocks, cont'

- Reader/writer spinlocks available
- Any number of readers allowed in, writers must receive exclusive access
- writers get priority, can cause starvation for readers if not used properly

Mutexes and Semaphores

- Use these if you will hold the lock for a long time or may block while holding the lock

  ```
  #include <linux/semaphore.h>
  void sema_init(struct semaphore *s, int v)
  DECLARE_MUTEX(mymutex);
  void down(struct semaphore *s);
  int down_interruptible(struct semaphore *s);
  int down_trylock(struct semaphore *s);
  void up(struct semaphore *s);
  ```

- reader/writer semaphores also available
  - writers get priority, can cause starvation for readers if not used properly
- Can't use these in interrupt context

Choosing the Right Synchronization

Kernel control paths accessing structure	UP protection	MP further protection
Exceptions	semaphore	semaphore
Interrupts	Local Interrupt Disabling	spin_lock_irqsave
Deferrable Functions (softirqs, tasklets)	None	spinlock
Exceptions and Interrupts	Local Interrupt Disabling	spin_lock_irqsave
Exceptions and Deferrable Functions	Local Softirq Disabling	spin_lock_bh
Interrupts and Deferrable Functions	Local Interrupt Disabling	spin_lock_irqsave
Exceptions, Interrupts, and Deferrable Functions	Local Interrupt Disabling	spin_lock_irqsave

Understanding the Linux Kernel, 3rd Ed. Pg 218

Lock Debugging

- CONFIG_DEBUG_SPINLOCK
  Catches missing spinlock init and other common errors
- CONFIG_DEBUG_MUTEXES
  ditto
- CONFIG_DEBUG_LOCK_ALLOC
  Detects a live lock being freed or reinitialized

Lock Debugging

- CONFIG_PROVE_LOCKING
  runtime checking of lock correctness
- CONFIG_LOCK_STAT
  collect lock statistics and show in /proc/lock_stat
- CONFIG_DEBUG_SPINLOCK_SLEEP
  Makes certain functions that sleep complain if you call them holding a spinlock

SMP Spinlock Bug: Example

- Locking the same lock twice is easy to avoid, right?

SMP Spinlock Bug: Example

- Locking the same lock twice is easy to avoid, right?
- Who here uses the datamover?

SMP Spinlock Bug: Example

- ADM irq occurs, ADM handler executes

SMP Spinlock Bug: Example

- ADM irq occurs, ADM handler executes
- spin_lock_irqsave(&msm_dmov_lock, irq_flags);

SMP Spinlock Bug: Example

- ADM irq occurs, ADM handler executes
- spin_lock_irqsave(&msm_dmov_lock, irq_flags);
- Calls device callback (nand, sd, etc)

SMP Spinlock Bug: Example

- ADM irq occurs, ADM handler executes
- spin_lock_irqsave(&msm_dmov_lock, irq_flags);
- Calls device callback (nand, sd, etc)
- Device callback decides there is more work to do, calls dma_enqueue

SMP Spinlock Bug: Example

- ADM irq occurs, ADM handler executes
- spin_lock_irqsave(&msm_dmov_lock, irq_flags);
- Calls device callback (nand, sd, etc)
- Device callback decides there is more work to do, calls dma_enqueue
- spin_lock_irqsave(&msm_dmov_lock, irq_flags);

SMP Spinlock Bug: Example

- ADM irq occurs, ADM handler executes
- spin_lock_irqsave(&msm_dmov_lock, irq_flags);
- Calls device callback (nand, sd, etc)
- Device callback decides there is more work to do, calls dma_enqueue
- spin_lock_irqsave(&msm_dmov_lock, irq_flags);
- Oops

SMP Spinlock Bug: Example

- The previous example works on UP
  - spin_lock_irqsave → disable interrupts
- Game over on SMP

Completions

- Communicate that something is done
- Mutexes/semaphores optimized for available case – not good for this job
- #include <linux/completion.h>
  DECLARE_COMPLETION(my_cmpl);
  init_completion(struct completion *);
  wait_for_completion(struct completion *);
  complete(struct completion *);
  …

Per-CPU Variables

- If your data is associated with each CPU in the system, these can offer a lockfree solution
- Elements are aligned to prevent false sharing in the cache
- Need to disable preemption - why?
- Additional synchronization needed if accessed in interrupt or softirq/tasklet context

Per-CPU Variables

```
#include <linux/percpu.h>

DEFINE_PER_CPU(type, varname);

/* disable preemption, get my element */
myvar = get_cpu_var(varname)
…work...
/* enable preemption */
put_cpu_var(varname)

alloc_percpu(type), free_percpu(type)
per_cpu_ptr(pointer, cpu)
```

Per-CPU Variables

```
#define get_cpu()
    ({ preempt_disable(); smp_processor_id(); })

#define put_cpu()
    preempt_enable()
```

Memory and Compiler Barriers

- The compiler may reorder seemingly unrelated instructions
- The CPU may reorder seemingly unrelated instructions
- When dealing with multiple observers this may not work
- Synchronization primitives contain the correct barrier operations

Memory Barriers

- #include <asm/system.h>
- Defines rmb(), wmb(), mb()
  - these evaluate to compiler barriers on UP
  - on SMP, each evaluates to a dmb
- Defines smp_mb(), smp_rmb(), smp_wmb()
  - these also compile to barrier() on UP
- Defines dmb(), dsb(), and isb()
- These APIs are under construction

Compiler Barrier

include/linux/compiler-gcc.h:

```
#define barrier() __asm__ __volatile__("": : :"memory")
```

- Prevents compiler from reordering instructions around the empty assembly stub
- Forces assumption that all memory locations have changed – cannot use previously loaded memory values
- Each memory barrier API in the kernel does this, in addition to any other behavior

Designing for SMP in the Linux Kernel - Agenda

- The “Good” News
- CONFIG_PREEMPT and CONFIG_SMP
- Concurrency primitives and APIs
- SMP scheduling
- Interrupt handling and Timers
- Testing

SMP Scheduling

- Separate runqueues for each CPU
  - with completely fair scheduler, separate time-ordered rb trees per CPU
- Scheduling behavior is not different per CPU for SMP than on UP
- How are CPU resources effectively managed system-wide?

Load Balancer

- CPUs are organized into scheduling domains and groups
  - this allows representation of various multiprocessor topologies (NUMA, hyperthreading, traditional SMP, etc)
- The load balancer runs occasionally and ensures runqueues are balanced

Interrupt Handling

- Linux kernel supports setting IRQ affinity to a set of CPUs
- QGIC is the new interrupt controller on 8660 and 8672
- QGIC supports configuration of an interrupt to be routed to one or more CPUs - but not round robin, random, etc., if multiple CPUs selected, individual interrupt is sent to multiple CPUs

Timers

- A couple different schemes
- There may be one timer, and an IPI used to broadcast it to other CPUs
  - possible downside to this approach?
- Each CPU may have its own local timer

Testing

- Yeah, you have to design for SMP correctness – still need to beef up testing
- Upcoming scheduler modifications to add hooks for tweaking
- Unit tests must go further than doing something once – need long stress tests in varying conditions
- Be smart about exercising concurrency boundaries
- Send ideas to linux.multicore!

Get Started Now

- Start reviewing your code for SMP safety
- Prelimenary builds for Virtio and RUMI3 coming soon
- RUMI3 availability no excuse – Virtio simulator installs on a standard desktop PC

References/Resources

- ARM v7 ARM (Architecture Reference Manual)
  - A3.5: Memory Types and Attributes and the Memory Order Model
  - A3.8: Memory Access Order (covers barriers)
- Multiprocessing Overview by Thomas Sartorius
- Linux Device Drivers
  - Chapter 5: Concurrency and Race Conditions
- QC Multicore Multiprocessing Guide
  Agile 80-VR848-1