• Generic locks implementations
  • Weaker memory model
  • C11 atomics (__atomic vs __sync)
    • Spinlock /RW lock
    • Ticket lock
    • MCS queued spinlock
• AArch64 specific lock implementations
  • WFE for Locks
    • Spinlock
    • RW Lock / Ticket Lock / MCS queued spinlock
• Key Takeaways
**Weaker Memory Model**

- Observed ordering of memory accesses may be influenced by:
  - HW / Compiler
- AArch64 has a weaker hardware memory model
  - Allows the processor to re-order, repeat, and merge accesses.
  - One CPU core can see values change in shared memory in a different order than another core wrote them.
- This memory reordering is transparent to programmers most of time
- Still there are cases where the observed memory ordering needs to be formalized, especially for multiple threads.
  - Memory fences help govern observed memory ordering.
- Memory fences degrade performance
  - Weaker memory models typically suffer the most.
  - Compilers support the relaxed C11 memory model in atomic built-ins.
C11 Atomics (__atomic vs __sync)

- Legacy __sync built-ins considered a full barrier

(gdb) disassemble /s rte_spinlock_lock
Dump of assembler code for function rte_spinlock_lock:

C11 Atomics (__atomic vs __sync)

- __atomic built-ins are memory model aware atomic operations
- __atomic allows one to specify less restrictive barriers
  - Full memory barrier → one-way barrier.

- The DMB ISH instruction splits memory accesses in program order into two groups
- Observers will observe group 1 memory accesses before group 2.

Load x3
Store x4

Store x1
Load x5

Load x1

MemBar Acquire

Load x2
Store x2
Load x3
Store x5

one-way barrier
Generic Spinlock

- `__atomic` instead of `__sync` built-ins
  - Full memory barrier \(\rightarrow\) one-way barrier
- Calling `__atomic` built-ins in spinlock
  - with RTE_FORCE_INTRINSICS enabled
  - Apply patches:
    - Spinlock test
    - Spinlock atomic one-way barrier

- arm SoC#1 @ 3GHz vs arm SoC#2 @ 2GHz
- 12-cores vs 27-cores
  - Performance degrades as more cores are involved in lock contention.
  - `__atomic` API implementation scales up much better with more cores contention

```
$ sudo ./test/test/test -c 0xfff00 -n 4 --socket-mem=1024,1024 --file-prefix=~ -- -I ...
RTE>>spinlock_autotest
```
Generic RW Lock

- `__atomic` instead of `__sync` built-ins
  - Full memory barrier → one-way barrier
- Calling `__atomic` built-ins in RW lock
  - with RTE_FORCE_INTRINSICS enabled
  - Apply patches:
    - RW lock atomic one-way barrier
    - RW lock test
- arm SoC#1 @ 3GHz vs arm SoC#2 @ 2GHz

```
$ sudo ./test/test/test -c 0xff00 -n 4 --socket-mem=1024,1024 --file-prefix=-- -I ...
RTE>>spinlock_autotest
```
Benchmarking on x86

- The default spinlock is a x86 architecture specified inline assembly lock.
  - Apply patch:
    - Spinlock test
      [http://patchwork.dpdk.org/patch/49822](http://patchwork.dpdk.org/patch/49822)
- Calling __sync built-ins on x86
  - with RTE_FORCE_INTRINSICS enabled
    - Apply patch:
      - Spinlock test
        [http://patchwork.dpdk.org/patch/49822](http://patchwork.dpdk.org/patch/49822)
- Calling __atomic built-ins on x86
  - with RTE_FORCE_INTRINSICS enabled
    - Apply patches:
      - Spinlock test
        [http://patchwork.dpdk.org/patch/49822](http://patchwork.dpdk.org/patch/49822)
      - Spinlock atomic one-way barrier
        [http://patchwork.dpdk.org/patch/49824](http://patchwork.dpdk.org/patch/49824)
- The relaxed memory ordering atomics will not degrade spinlock’s performance on strong memory model hardware.

```
$ sudo ./test/test/test -c 0xf0 -n 4 --socket-mem=1024,1024 --file-prefix=~ -- -I ...
```

RTE>>spinlock_autotest
Shortcomings of Current Spinlock implementation

Issues of Spinlock

Unfairness

• When PE releases the lock, will invalidate the other PE’s private caches.
• The PE who owns the local cache is more likely to get the lock again, starving other PEs.

Unpredictability

• Starvation causes unpredictable waiting time
• Starvation may cause throughput loss or more latency.

Cache bouncing

• Once the lock acquired and released. The PE will invalidate the other PE’s private caches.
• If the lock struct shared the cache line with other shared memory, the shared data modification will also cause cache bouncing

Spinlock UT shows the unfairness

RTE>>spinlock_autotest
...
Test with lock on 12 cores...
Core [20] count = 14541
Core [21] count = 15573
Core [22] count = 180639
Core [23] count = 180372
Core [24] count = 1
Core [25] count = 1
Core [26] count = 17
Core [27] count = 28
Core [28] count = 6
Core [29] count = 7
Core [30] count = 1
Core [31] count = 1
Total count = 391187

Ref: https://lwn.net/Articles/267968/
Ticket Lock

The problems to address

- Unfairness
- Unpredictability

How to address

- **Ticket based**
  - Each request increases the next ticket by 1.
  - Each release increases the current ticket by 1.
  - Whose current matches next takes the lock.
- **FIFO service**

```c
typedef struct {
    uint16_t current;
    uint16_t next;
} rte_ticketlock_t;
```

Ticket lock

[http://patchwork.dpdk.org/cover/50359/](http://patchwork.dpdk.org/cover/50359/)
MCS Queued Lock

The MCS lock (proposed by Mellor-Crummey and Scott) is a simple spin-lock with each CPU trying to acquire the lock spinning on its own variable.

Advantages

- Guarantees FIFO lock services
- Remove cache line bouncing
  - Spins on the core own local variables only
- Requires a small constant amount of space per lock
- Requiring only $O(1)$ cache coherency transactions per lock acquisition

typedef struct rte_mcs_spinlock {
  struct rte_mcs_spinlock *next;
  int locked; /* 1 if locked, 0 otherwise */
} rte_mcs_spinlock_t;

https://lwn.net/Articles/590243/
MCS Queued Lock cont.

- Pointer in the "main" lock is the tail of the queue of waiting CPUs.

```c
typedef struct rte_mcs_spinlock {
    struct rte_mcs_spinlock *next;
    int locked; /* 1 if locked, 0 otherwise */
} rte_mcs_spinlock_t;

void rte_mcs_spinlock_lock(rte_mcs_spinlock_t **msl, rte_mcs_spinlock_t *me);
void rte_mcs_spinlock_unlock(rte_mcs_spinlock_t **msl, rte_mcs_spinlock_t *me);
```
Aarch64 specific Spinlock implementation

- **WFE**
  - Tight loop → suspend execution
  - Less stress to memory subsystem
  - Less power

- **Benchmarking**
  - WFE spinlock is on par with __sync spinlock when testing on 12 lcores.
  - With increasing contentions (12 → 96 lcores contending for the lock), more contention cause less total number of locking and unlocking.
  - WFE scales much better than __sync.
How does WFE work?

- **WFE (wait for event)**
  - Suspend execution when the lock was held by other PEs, instead of continuous polling
  - Get wake up events if the other PE releases the lock
- **Use cases: busy-polling waiting**
  - Spinlock
  - RW lock
  - Ticket lock
  - MCS spinlock
Work in progress

- Generic implementations
  - Spinlock (under community review arming at 19.05)
  - RW lock (under community review arming at 19.05)
  - Ticket lock (under community review arming at 19.05)
  - MCS queued spinlock (under internal review arming at 19.08)

- Aarch64 specific implementations
  - WFE Spinlock (under internal review, aiming at 19.08)
  - Planning to use it in RW lock, ticket lock and MCS lock
Key Takeaways

- More performant, scalable RW lock, spinlock
- New ticket lock, MCS queued spinlock
- Integration of Arm specific features in locks
Thanks!

Gavin Hu
gavin.hu@arm.com
Phil Yang
phil.yang@arm.com