SO2 Lecture 05 - Symmetric Multi-Processing

View slides

Lecture objectives:

- Kernel Concurrency
- Atomic operations
- Spin locks
- Cache thrashing
- Optimized spin locks
- Process and Interrupt Context Synchronization
- Mutexes
- Per CPU data
- Memory Ordering and Barriers
- Read-Copy Update

Synchronization basics

Because the Linux kernel supports symmetric multi-processing (SMP) it must use a set of synchronization mechanisms to achieve predictable results, free of race conditions.

> **Note**
>
> We will use the terms core, CPU and processor as interchangeable for the purpose of this lecture.

Race conditions can occur when the following two conditions happen simultaneously:

- there are at least two execution contexts that run in "parallel":
  - truly run in parallel (e.g. two system calls running on different processors)
  - one of the contexts can arbitrary preempt the other (e.g. an interrupt preempts a system call)
- the execution contexts perform read-write accesses to shared memory

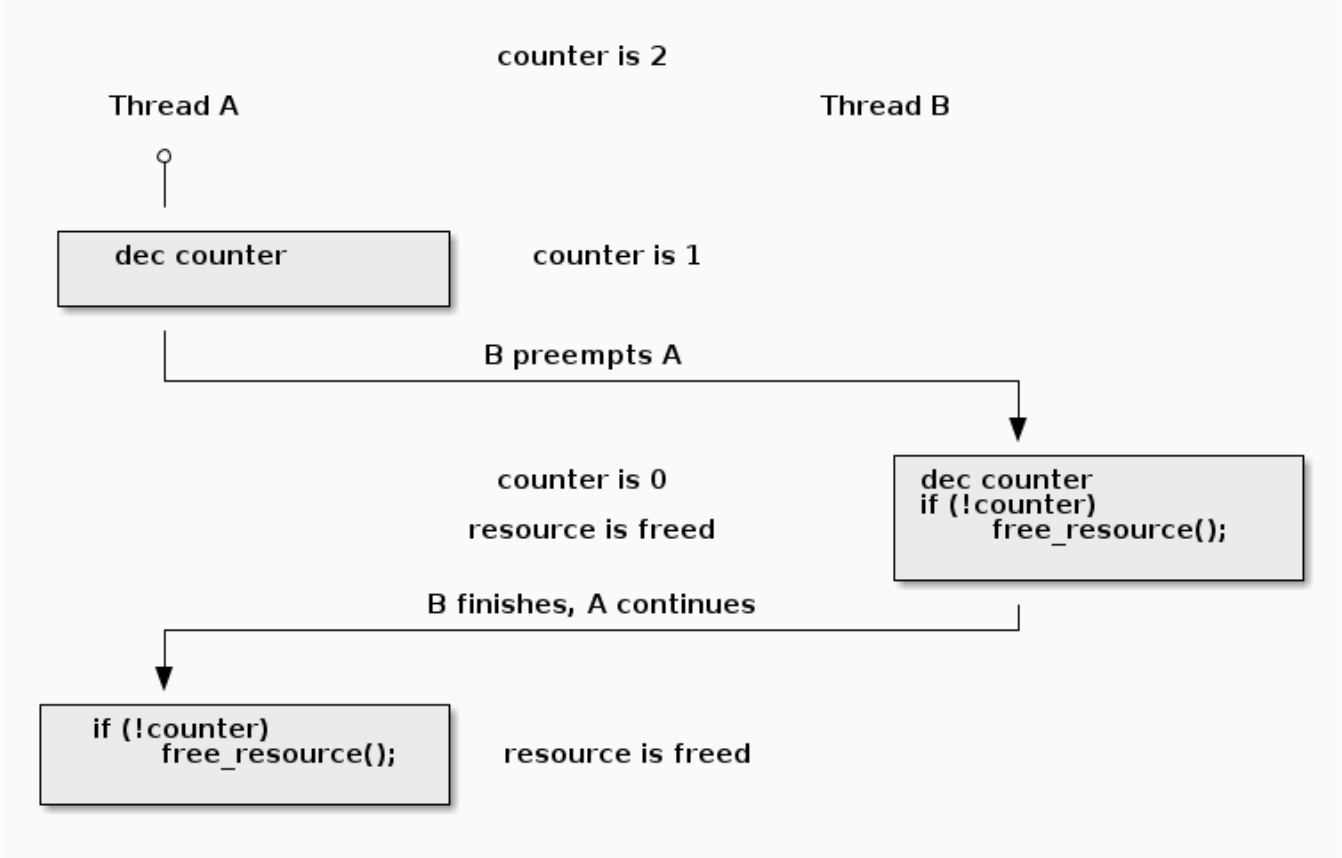
Race conditions can lead to erroneous results that are hard to debug, because they manifest only when the execution contexts are scheduled on the CPU cores in a very specific order.

A classical race condition example is an incorrect implementation for a release operation of a resource counter:

```
void release_resource()
{
     counter--;

     if (!counter)
         free_resource();
}
```

A resource counter is used to keep a shared resource available until the last user releases it but the above implementation has a race condition that can cause freeing the resource twice:



In most cases the *release_resource()* function will only free the resource once. However, in the scenario above, if thread A is preempted right after decrementing *counter* and thread B calls *release_resource()* it will cause the resource to be freed. When resumed, thread A will also free the resource since the counter value is 0.

To avoid race conditions the programmer must first identify the critical section that can generate a race condition. The critical section is the part of the code that reads and writes shared memory from multiple parallel contexts.

In the example above, the minimal critical section is starting with the counter decrement and ending with checking the counter's value.

Once the critical section has been identified race conditions can be avoided by using one of the following approaches:

- make the critical section **atomic** (e.g. use atomic instructions)
- **disable preemption** during the critical section (e.g. disable interrupts, bottom-half handlers, or thread preemption)
- **serialize the access** to the critical section (e.g. use spin locks or mutexes to allow only one context or thread in the critical section)

Linux kernel concurrency sources

There are multiple source of concurrency in the Linux kernel that depend on the kernel configuration as well as the type of system it runs on:

- **single core systems**, **non-preemptive kernel**: the current process can be preempted by interrupts
- **single core systems**, **preemptive kernel**: above + the current process can be preempted by other processes
- **multi-core systems**: above + the current process can run in parallel with another process or with an interrupt running on another processor

> **Note**
>
> We only discuss kernel concurrency and that is why a non-preemptive kernel running on an single core system has interrupts as the only source of concurrency.

Atomic operations

In certain circumstances we can avoid race conditions by using atomic operations that are provided by hardware. Linux provides a unified API to access atomic operations:

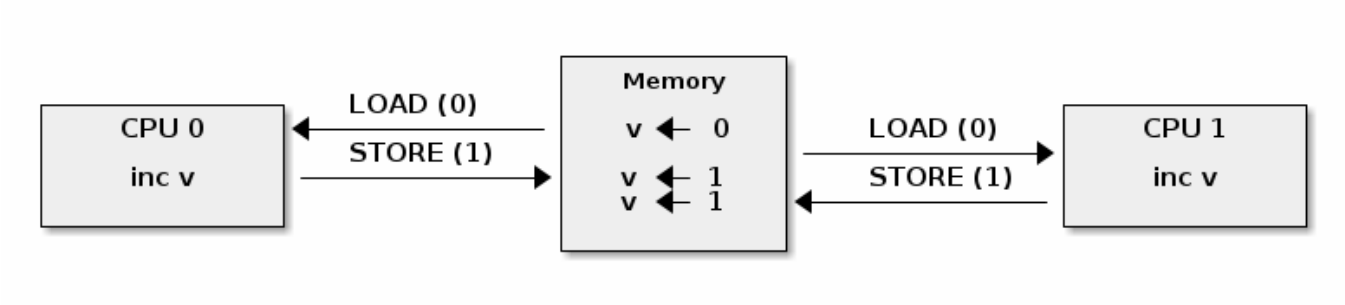
- integer based:
  - simple: `atomic_inc()`, `atomic_dec()`, `atomic_add()`, `atomic_sub()`
  - conditional: `atomic_dec_and_test()`, `atomic_sub_and_test()`
- bit based:
  - simple: `test_bit()`, `set_bit()`, `change_bit()`
  - conditional: `test_and_set_bit()`, `test_and_clear_bit()`, `test_and_change_bit()`

For example, we could use `atomic_dec_and_test()` to implement the resource counter decrement and value checking atomic:

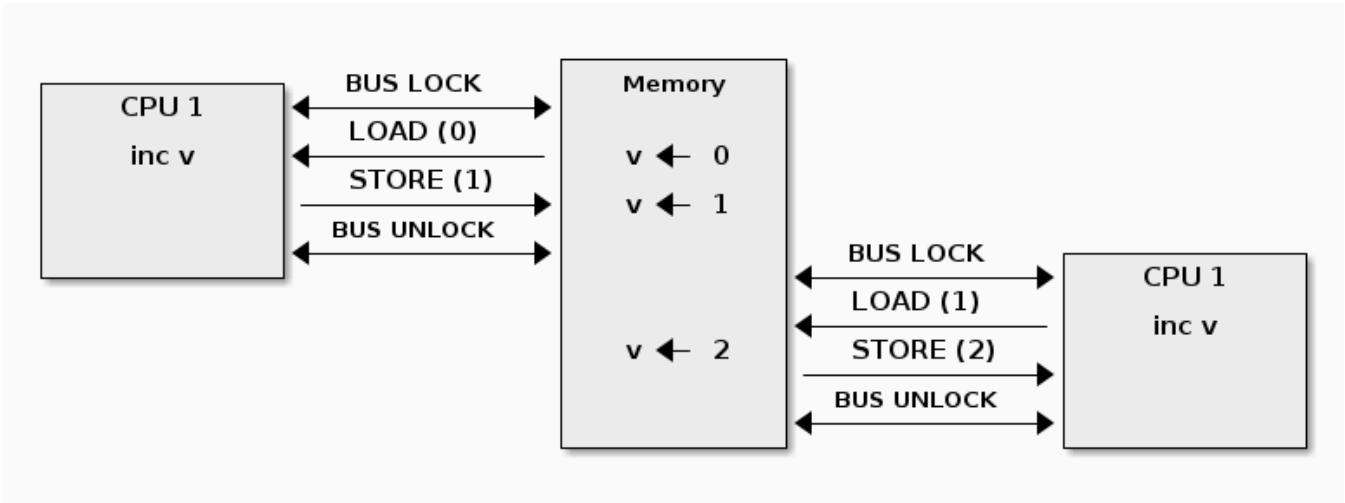
```
void release_resource()
{
    if (atomic_dec_and_test(&counter))
         free_resource();
}
```

One complication with atomic operations is encountered in multi-core systems, where an atomic operation is not longer atomic at the system level (but still atomic at the core level).

To understand why, we need to decompose the atomic operation in memory loads and stores. Then we can construct race condition scenarios where the load and store operations are interleaved across CPUs, like in the example below where incrementing a value from two processors will produce an unexpected result:



In order to provide atomic operations on SMP systems different architectures use different techniques. For example, on x86 a LOCK prefix is used to lock the system bus while executing the prefixed operation:



On ARM the LDREX and STREX instructions are used together to guarantee atomic access: LDREX loads a value and signals the exclusive monitor that an atomic operation is in progress. The STREX attempts to store a new value but only succeeds if the exclusive monitor has not detected other exclusive operations. So, to implement atomic operations the programmer must retry the operation (both LDREX and STREX) until the exclusive monitor signals a success.

Although they are often interpreted as "light" or "efficient" synchronization mechanisms (because they "don't require spinning or context switches", or because they "are implemented in hardware so they must be more efficient", or because they "are just instructions so they must have similar efficiency as other instructions"), as seen from the implementation details, atomic operations are actually expensive.

Disabling preemption (interrupts)

On single core systems and non preemptive kernels the only source of concurrency is the preemption of the current thread by an interrupt. To prevent concurrency is thus sufficient to disable interrupts.

This is done with architecture specific instructions, but Linux offers architecture independent APIs to disable and enable interrupts:

```
 #define local_irq_disable() \
     asm volatile („cli” : : : „memory”)

#define local_irq_enable() \
    asm volatile („sti” : : : „memory”)

#define local_irq_save(flags) \
    asm volatile ("pushf ; pop %0" :"=g" (flags)
                  : /* no input */: "memory") \
    asm volatile("cli": : :"memory")

#define local_irq_restore(flags) \
    asm volatile ("push %0 ; popf"
                  : /* no output */
                  : "g" (flags) :"memory", "cc");
```

Although the interrupts can be explicitly disabled and enable with `local_irq_disable()` and `local_irq_enable()` these APIs should only be used when the current state and interrupts is known. They are usually used in core kernel code (like interrupt handling).

For typical cases where we want to avoid interrupts due to concurrency issues it is recommended to use the `local_irq_save()` and `local_irq_restore()` variants. They take care of saving and restoring the interrupts states so they can be freely called from overlapping critical sections without the risk of accidentally enabling interrupts while still in a critical section, as long as the calls are balanced.

Spin Locks

Spin locks are used to serialize access to a critical section. They are necessary on multi-core systems where we can have true execution parallelism. This is a typical spin lock implementation:

```
spin_lock:
    lock bts [my_lock], 0
    jc spin_lock

/* critical section */

spin_unlock:
    mov [my_lock], 0
```

bts dts, src - bit test and set; it copies the src bit from the dts memory address to the carry flag and then sets it:

```
CF <- dts[src]
dts[src] <- 1
```

As it can be seen, the spin lock uses an atomic instruction to make sure that only one core can enter the critical section. If there are multiple cores trying to enter they will continuously "spin" until the lock is released.

While the spin lock avoids race conditions, it can have a significant impact on the system's performance due to "lock contention":

- There is lock contention when at least one core spins trying to enter the critical section lock
- Lock contention grows with the critical section size, time spent in the critical section and the number of cores in the system

Another negative side effect of spin locks is cache thrashing.

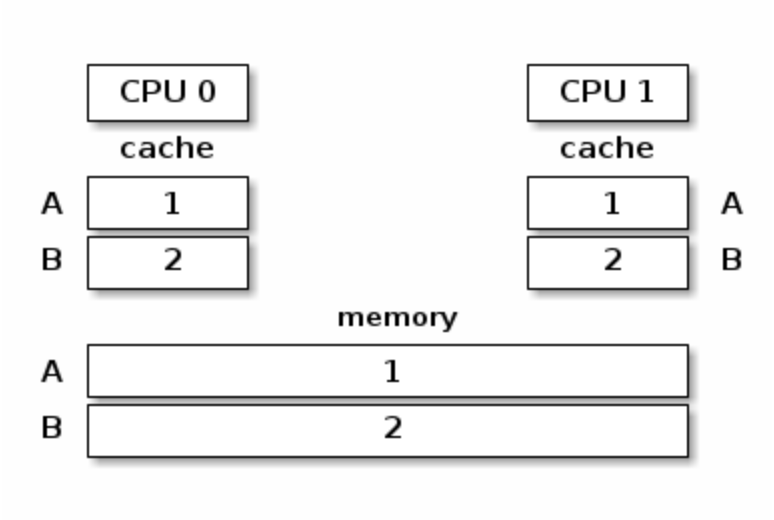
Cache thrashing occurs when multiple cores are trying to read and write to the same memory resulting in excessive cache misses.

Since spin locks continuously access memory during lock contention, cache thrashing is a common occurrence due to the way cache coherency is implemented.

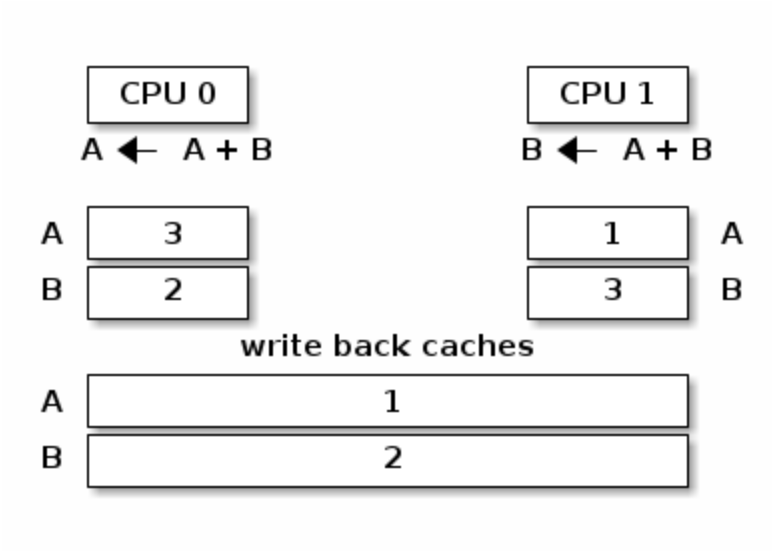
Cache coherency in multi-processor systems

The memory hierarchy in multi-processor systems is composed of local CPU caches (L1 caches), shared CPU caches (L2 caches) and the main memory. To explain cache coherency we will ignore the L2 cache and only consider the L1 caches and main memory.

In the figure below we present a view of the memory hierarchy with two variables A and B that fall into different cache lines and where caches and the main memory are synchronized:



In the absence of a synchronization mechanism between the caches and main memory, when CPU 0 executes $A = A + B$ and CPU 1 executes $B = A + B$ we will have the following memory view:



In order to avoid the situation above multi-processor systems use cache coherency protocols. There are two main types of cache coherency protocols:

- Bus snooping (sniffing) based: memory bus transactions are monitored by caches and they take actions to preserve coherency
- Directory based: there is a separate entity (directory) that maintains the state of caches; caches interact with directory to preserve coherency

Bus snooping is simpler but it performs poorly when the number of cores goes beyond 32-64.

Directory based cache coherence protocols scale much better (up to thousands of cores) and are usually used in NUMA systems.

A simple cache coherency protocol that is commonly used in practice is MESI (named after the acronym of the cache line states names: **Modified**, **Exclusive**, **Shared** and **Invalid**). It's main characteristics are:

- Caching policy: write back
- Cache line states
  - Modified: owned by a single core and dirty
  - Exclusive: owned by a single core and clean
  - Shared: shared between multiple cores and clean
  - Invalid : the line is not cached

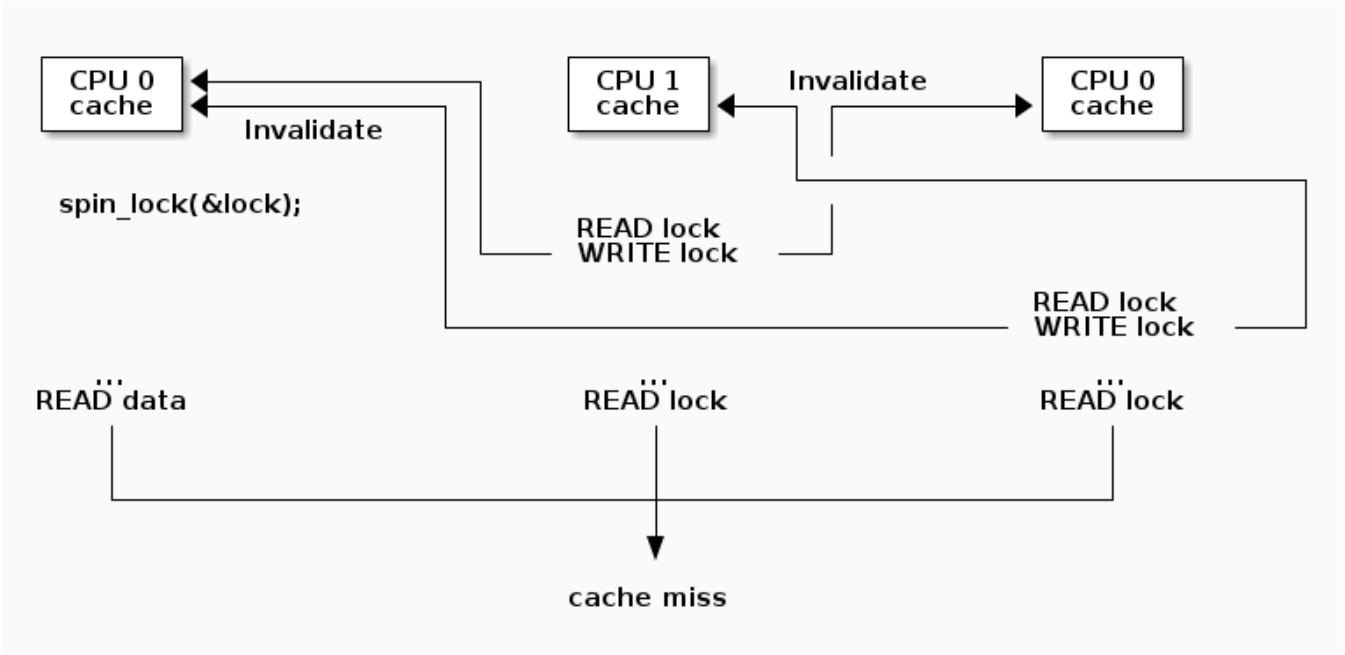
Issuing read or write requests from CPU cores will trigger state transitions, as exemplified below:

- Invalid -> Exclusive: read request, all other cores have the line in Invalid; line loaded from memory
- Invalid -> Shared: read request, at least one core has the line in Shared or Exclusive; line loaded from sibling cache
- Invalid/Shared/Exclusive -> Modified: write request; **all other** cores **invalidate** the line
- Modified -> Invalid: write request from other core; line is flushed to memory

> **Note**
>
> The most important characteristic of the MESI protocol is that it is a write-invalidate cache protocol. When writing to a shared location all other caches are invalidated.

This has important performance impact in certain access patterns, and one such pattern is contention for a simple spin lock implementation like we discussed above.

To exemplify this issue lets consider a system with three CPU cores, where the first has acquired the spin lock and it is running the critical section while the other two are spinning waiting to enter the critical section:



As it can be seen from the figure above due to the writes issued by the cores spinning on the lock we see frequent cache line invalidate operations which means that basically the two waiting cores will flush and load the cache line while waiting for the lock, creating unnecessary traffic on the memory bus and slowing down memory accesses for the first core.

Another issue is that most likely data accessed by the first CPU during the critical section is stored in the same cache line with the lock (common optimization to have the data ready in the cache after the lock is acquired). Which means that the cache invalidation triggered by the two other spinning cores will slow down the execution of the critical section which in turn triggers more cache invalidate actions.

Optimized spin locks

As we have seen simple spin lock implementations can have poor performance issues due to cache thrashing, especially as the number of cores increase. To avoid this issue there are two possible strategies:

- reduce the number of writes and thus reduce the number of cache invalidate operations
- avoid the other processors spinning on the same cache line, and thus avoid the cache invalidate operations

An optimized spin lock implementation that uses the first approach is presented below:

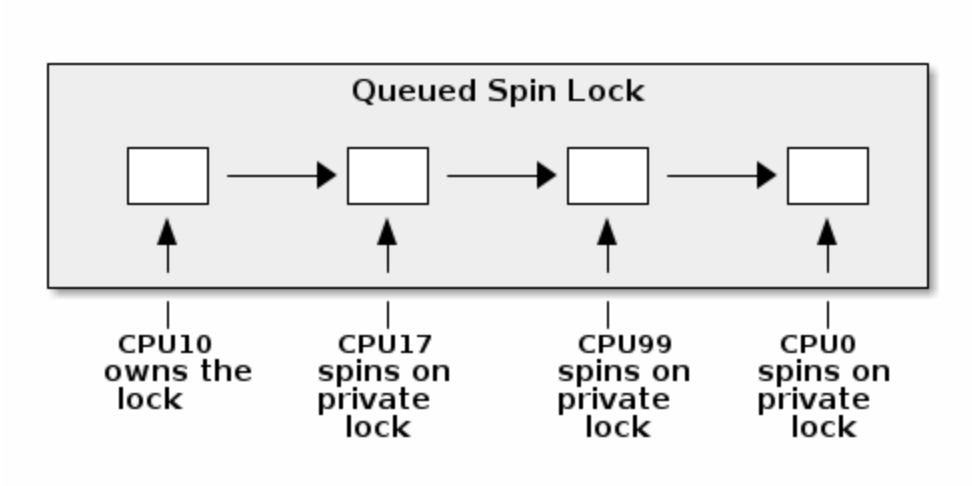
```
spin_lock:
    rep ; nop
    test lock_addr, 1
    jnz spin_lock
    lock bts lock_addr
    jc spin_lock
```

- we first test the lock read only, using a non atomic instructions, to avoid writes and thus invalidate operations while we spin
- only when the lock *might* be free, we try to acquire it

The implementation also use the **PAUSE** instruction to avoid pipeline flushes due to (false positive) memory order violations and to add a small delay (proportional with the memory bus frequency) to reduce power consumption.

A similar implementation with support for fairness (the CPU cores are allowed in the critical section based on the time of arrival) is used in the Linux kernel (the ticket spin lock) for many architectures.

However, for the x86 architecture, the current spin lock implementation uses a queued spin lock where the CPU cores spin on different locks (hopefully distributed in different cache lines) to avoid cache invalidation operations:



Conceptually, when a new CPU core tries to acquire the lock and it fails it will add its private lock to the list of waiting CPU cores. When the lock owner exits the critical section it unlocks the next lock in the list, if any.

While a read spin optimized spin lock reduces most of the cache invalidation operations, the lock owner can still generate cache invalidate operations due to writes to data structures close to the lock and thus part of the same cache line. This in turn generates memory traffic on subsequent reads on the spinning cores.

Hence, queued spin locks scale much better for large number of cores as is the case for NUMA systems. And since they have similar fairness properties as the ticket lock it is the preferred implementation on the x86 architecture.

Process and Interrupt Context Synchronization

Accessing shared data from both process and interrupt context is a relatively common scenario. On single core systems we can do this by disabling interrupts, but that won't work on multi-core systems, as we can have the process running on one CPU core and the interrupt context running on a different CPU core.

Using a spin lock, which was designed for multi-processor systems, seems like the right solution, but doing so can cause common deadlock conditions, as detailed by the following scenario:

- In the process context we take the spin lock
- An interrupt occurs and it is scheduled on the same CPU core
- The interrupt handler runs and tries to take the spin lock
- The current CPU will deadlock

To avoid this issue a two fold approach is used:

- In process context: disable interrupts and acquire a spin lock; this will protect both against interrupt or other CPU cores race conditions (`spin_lock_irqsave()` and `spin_lock_restore()` combine the two operations)
- In interrupt context: take a spin lock; this will will protect against race conditions with other interrupt handlers or process context running on different processors

We have the same issue for other interrupt context handlers such as softirqs, tasklets or timers and while disabling interrupts might work, it is recommended to use dedicated APIs:

- In process context use `spin_lock_bh()` (which combines `local_bh_disable()` and `spin_lock()`) and `spin_unlock_bh()` (which combines `spin_unlock()` and `local_bh_enable()`)
- In bottom half context use: `spin_lock()` and `spin_unlock()` (or `spin_lock_irqsave()` and `spin_lock_irqrestore()` if sharing data with interrupt handlers)

As mentioned before, another source of concurrency in the Linux kernel can be other processes, due to preemption.

Preemption is configurable: when active it provides better latency and response time, while when deactivated it provides better throughput.

Preemption is disabled by spin locks and mutexes but it can be manually disabled as well (by core kernel code).

As for local interrupt enabling and disabling APIs, the bottom half and preemption APIs allows them to be used in overlapping critical sections. A counter is used to track the state of bottom half and preemption. In fact the same counter is used, with different increment values:

```
#define PREEMPT_BITS      8
#define SOFTIRQ_BITS      8
#define HARDIRQ_BITS      4
#define NMI_BITS          1

#define preempt_disable() preempt_count_inc()

#define local_bh_disable() add_preempt_count(SOFTIRQ_OFFSET)

#define local_bh_enable() sub_preempt_count(SOFTIRQ_OFFSET)

#define irq_count() (preempt_count() & (HARDIRQ_MASK | SOFTIRQ_MASK))

#define in_interrupt() irq_count()

asmlinkage void do_softirq(void)
{
    if (in_interrupt()) return;
    ...
```

Mutexes

Mutexes are used to protect against race conditions from other CPU cores but they can only be used in **process context**. As opposed to spin locks, while a thread is waiting to enter the critical section it will not use CPU time, but instead it will be added to a waiting queue until the critical section is vacated.

Since mutexes and spin locks usage intersect, it is useful to compare the two:

- They don't "waste" CPU cycles; system throughput is better than spin locks if context switch overhead is lower than medium spinning time
- They can't be used in interrupt context
- They have a higher latency than spin locks

Conceptually, the `mutex_lock()` operation is relatively simple: if the mutex is not acquired we can take the fast path via an atomic exchange operation:

```
void __sched mutex_lock(struct mutex *lock)
{
  might_sleep();

  if (!__mutex_trylock_fast(lock))
    __mutex_lock_slowpath(lock);
}

static __always_inline bool __mutex_trylock_fast(struct mutex *lock)
{
  unsigned long curr = (unsigned long)current;

  if (!atomic_long_cmpxchg_acquire(&lock->owner, 0UL, curr))
    return true;

  return false;
}
```

otherwise we take the slow path where we add ourselves to the mutex waiting list and put ourselves to sleep:

```
...
  spin_lock(&lock->wait_lock);
...
  /* add waiting tasks to the end of the waitqueue (FIFO): */
  list_add_tail(&waiter.list, &lock->wait_list);
...
  waiter.task = current;
...
  for (;;) {
    if (__mutex_trylock(lock))
      goto acquired;
  ...
    spin_unlock(&lock->wait_lock);
  ...
    set_current_state(state);
    spin_lock(&lock->wait_lock);
  }
  spin_lock(&lock->wait_lock);
acquired:
  __set_current_state(TASK_RUNNING);
  mutex_remove_waiter(lock, &waiter, current);
  spin_lock(&lock->wait_lock);
...
```

The full implementation is a bit more complex: instead of going to sleep immediately it optimistic spinning if it detects that the lock owner is currently running on a different CPU as chances are the owner will release the lock soon. It also checks for signals and handles mutex debugging for locking dependency engine debug feature.

The `mutex_unlock()` operation is symmetric: if there are no waiters on the mutex then we can take the fast path via an atomic exchange operation:

```
void __sched mutex_unlock(struct mutex *lock)
{
  if (__mutex_unlock_fast(lock))
    return;
  __mutex_unlock_slowpath(lock, _RET_IP_);
}

static __always_inline bool __mutex_unlock_fast(struct mutex *lock)
{
  unsigned long curr = (unsigned long)current;

  if (atomic_long_cmpxchg_release(&lock->owner, curr, 0UL) == curr)
    return true;

  return false;
}

void __mutex_lock_slowpath(struct mutex *lock)
{
...
  if (__mutex_waiter_is_first(lock, &waiter))
          __mutex_set_flag(lock, MUTEX_FLAG_WAITERS);
...
```

> **Note**
>
> Because `struct task_struct` is cached aligned the 7 lower bits of the owner field can be used for various flags, such as `MUTEX_FLAG_WAITERS`.

Otherwise we take the slow path where we pick up first waiter from the list and wake it up:

```
...
spin_lock(&lock->wait_lock);
if (!list_empty(&lock->wait_list)) {
  /* get the first entry from the wait-list: */
  struct mutex_waiter *waiter;
  waiter = list_first_entry(&lock->wait_list, struct mutex_waiter,
                            list);
  next = waiter->task;
  wake_q_add(&wake_q, next);
}
...
spin_unlock(&lock->wait_lock);
...
wake_up_q(&wake_q);
```

Per CPU data

Per CPU data avoids race conditions by avoiding to use shared data. Instead, an array sized to the maximum possible CPU cores is used and each core will use its own array entry to read and write data. This approach certainly has advantages:

- No need to synchronize to access the data
- No contention, no performance impact
- Well suited for distributed processing where aggregation is only seldom necessary (e.g. statistics counters)

Memory Ordering and Barriers

Modern processors and compilers employ out-of-order execution to improve performance. For example, processors can execute "future" instructions while waiting for current instruction data to be fetched from memory.

Here is an example of out of order compiler generated code:

C code	Compiler generated code

`a = 1;`
`b = 2;`
	`MOV R10, 1`
	`MOV R11, 2`
	`STORE R11, b`
	`STORE R10, a`

> **Note**
>
> When executing instructions out of order the processor makes sure that data dependency is observed, i.e. it won't execute instructions whose input depend on the output of a previous instruction that has not been executed.

In most cases out of order execution is not an issue. However, in certain situations (e.g. communicating via shared memory between processors or between processors and hardware) we must issue some instructions before others even without data dependency between them.

For this purpose we can use barriers to order memory operations:

- A read barrier (`rmb()` , `smp_rmb()`) is used to make sure that no read operation crosses the barrier; that is, all read operation before the barrier are complete before executing the first instruction after the barrier
- A write barrier (`wmb()` , `smp_wmb()`) is used to make sure that no write operation crosses the barrier
- A simple barrier (`mb()` , `smp_mb()`) is used to make sure that no write or read operation crosses the barrier

Read Copy Update (RCU)

Read Copy Update is a special synchronization mechanism similar with read-write locks but with significant improvements over it (and some limitations):

- **Read-only** lock-less access at the same time with write access
- Write accesses still requires locks in order to avoid races between writers
- Requires unidirectional traversal by readers

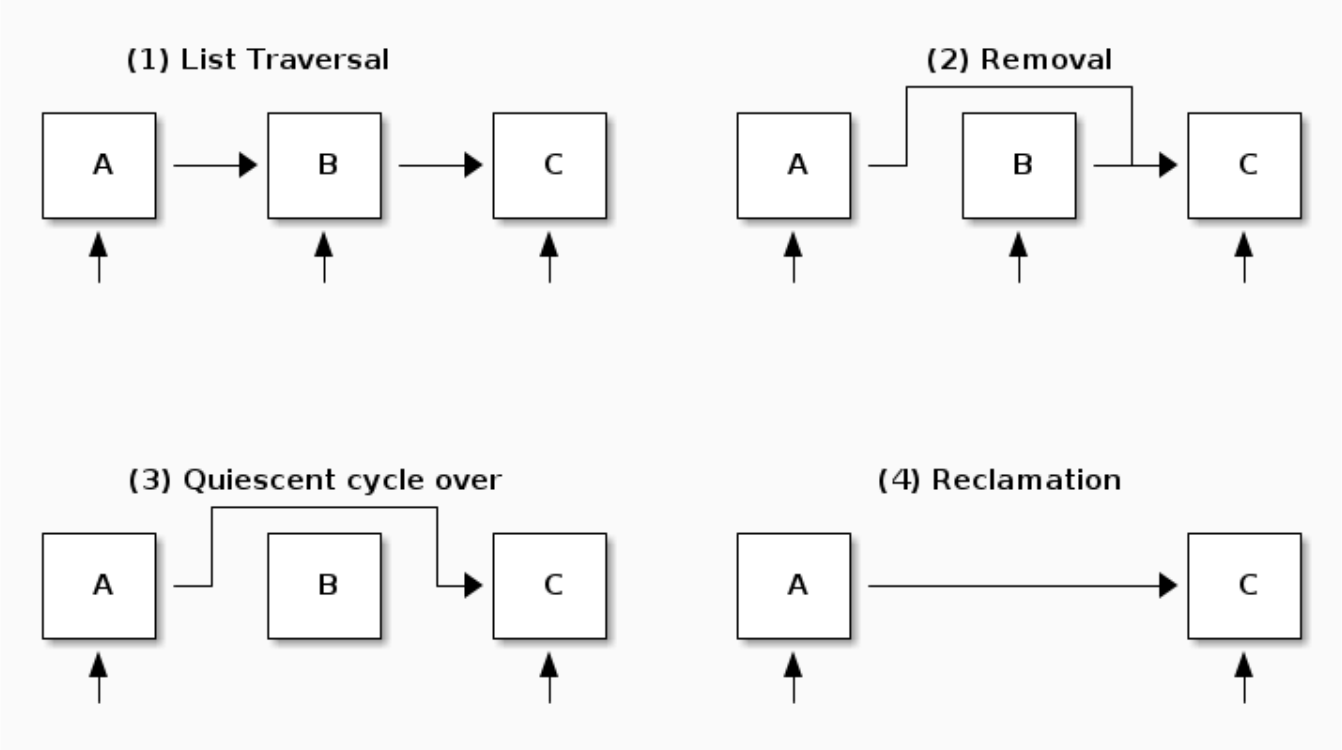
In fact, the read-write locks in the Linux kernel have been deprecated and then removed, in favor of RCU.

Implementing RCU for a new data structure is difficult, but a few common data structures (lists, queues, trees) do have RCU APIs that can be used.

RCU splits removal updates to the data structures in two phases:

- **Removal**: removes references to elements. Some old readers may still see the old reference so we can't free the element.
- **Elimination**: free the element. This action is postponed until all existing readers finish traversal (quiescent cycle). New readers won't affect the quiescent cycle.

As an example, lets take a look on how to delete an element from a list using RCU:



In the first step it can be seen that while readers traverse the list all elements are referenced. In step two a writer removes element B. Reclamation is postponed since there are still readers that hold references to it. In step three a quiescent cycle just expired and it can be noticed that there are no more references to element B. Other elements still have references from readers that started the list traversal after the element was removed. In step 4 we finally perform reclamation (free the element).

Now that we covered how RCU functions at the high level, lets looks at the APIs for traversing the list as well as adding and removing an element to the list:

```
/* list traversal */
rcu_read_lock();
list_for_each_entry_rcu(i, head) {
  /* no sleeping, blocking calls or context switch allowed */
}
rcu_read_unlock();


/* list element delete  */
spin_lock(&lock);
list_del_rcu(&node->list);
spin_unlock(&lock);
synchronize_rcu();
kfree(node);

/* list element add  */
spin_lock(&lock);
list_add_rcu(head, &node->list);
spin_unlock(&lock);
```