Lecture 9

Synchronization

Concurrency in the Kernel

- On symmetric multiprocessing (SMP) OS kernels are pre-emptible
- True concurrency
  - Multiple processors execute instructions simultaneously
- Pseudo concurrency
  - Instructions of multiple execution sequences are interleaved

Sources of pseudo concurrency

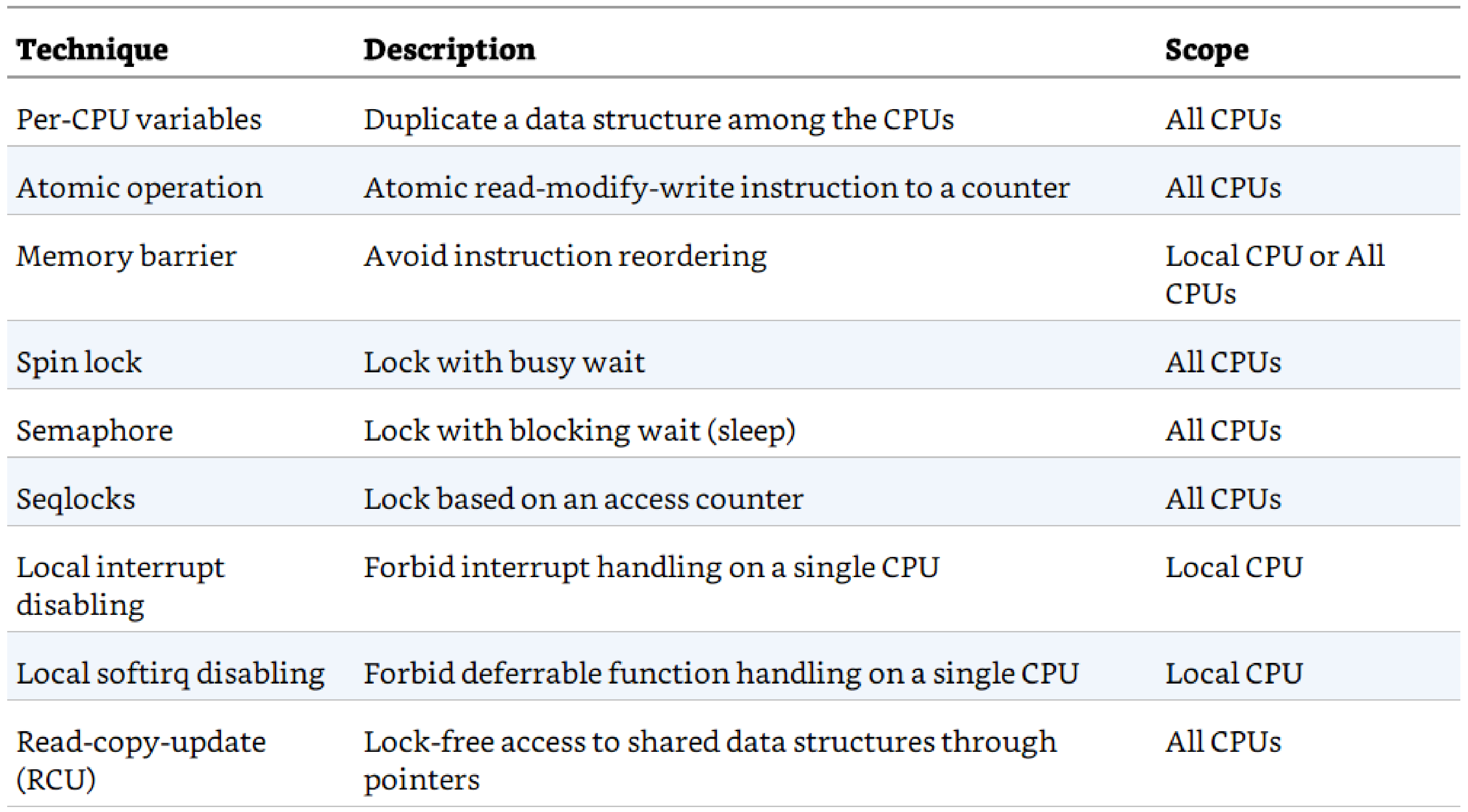
- Software-based pre-emption
  - Voluntary preemption (sleep/yield)
  - Involuntary preemption (preemptable kernel)
    - Scheduler switches threads regardless of whether they are running in user or kernel mode
  - Solutions: don’t do the former, disable preemption to prevent the latter
- Hardware pre-emption
  - Interrupt/trap/fault/exception handlers can start executing at any time
  - Solution: disable interrupts

Kernel Control Paths

- Interrupt handlers
- Exception handlers
- User-space threads in kernel(system calls)
- Kernel threads(idle, work queue, pdflush…)
- Bottom halves(soft irq, tasklet,BH...)

Note: Something like the memory allocator is not a KCP. It is called in a KCP.

Synchronization techniques in Linux



Technique	Description	Scope
Per-CPU variables	Duplicate a data structure among the CPUs	All CPUs
Atomic operation	Atomic read-modify-write instruction to a counter	All CPUs
Memory barrier	Avoid instruction reordering	Local CPU or All CPUs
Spin lock	Lock with busy wait	All CPUs
Semaphore	Lock with blocking wait (sleep)	All CPUs
Seqlocks	Lock based on an access counter	All CPUs
Local interrupt disabling	Forbid interrupt handling on a single CPU	Local CPU
Local softirq disabling	Forbid deferrable function handling on a single CPU	Local CPU
Read-copy-update (RCU)	Lock-free access to shared data structures through pointers	All CPUs

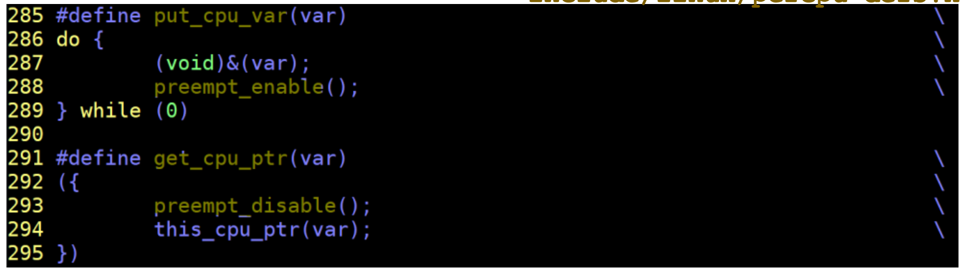
Per-CPU Variables

Per-CPU variables

- One variable per CPU
  - Reduce need to synchronize across CPUs

- Additional protection may be needed
  - Example: a kernel control path gets the address of its local copy of a per-CPU variable, and then it is preempted and moved to another CPU: the address still refers to the element of the previous CPU.
  - Control preemption before performing read or write.

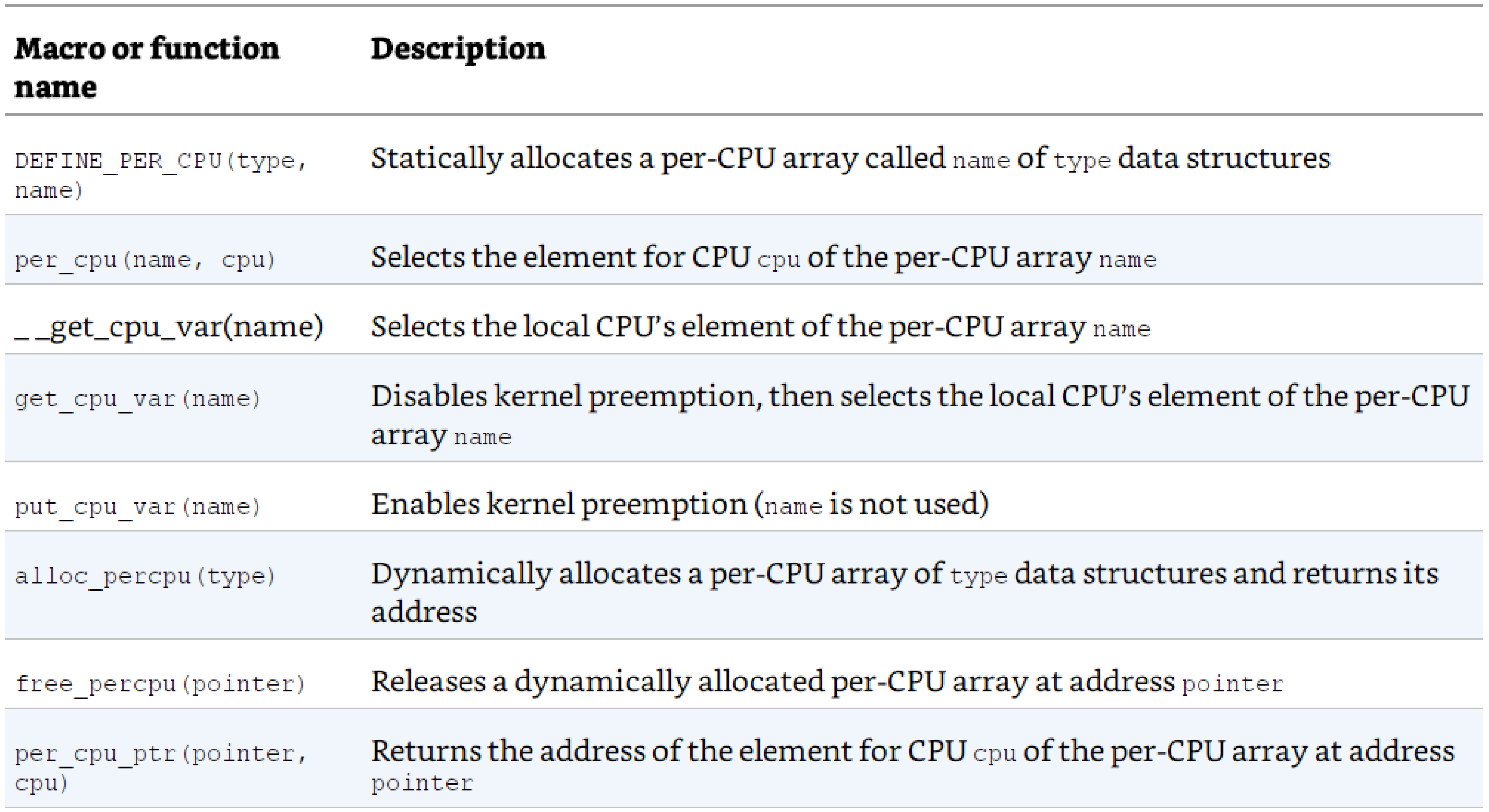
include/linux/percpu-defs.h



```
#define put_cpu_var(var)                                                \
do {                                                                    \
        (void)&(var);                                                   \
        preempt_enable();                                               \
} while (0)

#define get_cpu_ptr(var)                                                \
({                                                                      \
        preempt_disable();                                              \
        this_cpu_ptr(var);                                              \
})
```

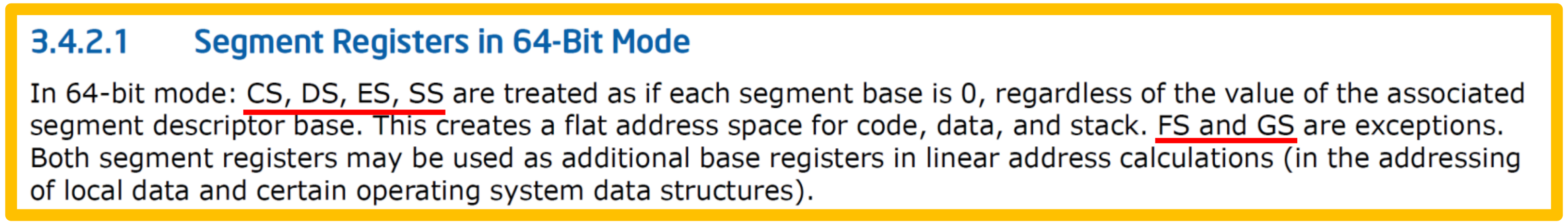
Linux per-CPU variables



Macro or function name	Description
`DEFINE_PER_CPU(type, name)`	Statically allocates a per-CPU array called `name` of `type` data structures
`per_cpu(name, cpu)`	Selects the element for CPU `cpu` of the per-CPU array `name`
__get_cpu_var(name)	Selects the local CPU's element of the per-CPU array `name`
`get_cpu_var(name)`	Disables kernel preemption, then selects the local CPU's element of the per-CPU array `name`
`put_cpu_var(name)`	Enables kernel preemption (`name` is not used)
`alloc_percpu(type)`	Dynamically allocates a per-CPU array of `type` data structures and returns its address
`free_percpu(pointer)`	Releases a dynamically allocated per-CPU array at address `pointer`
`per_cpu_ptr(pointer, cpu)`	Returns the address of the element for CPU `cpu` of the per-CPU array at address `pointer`

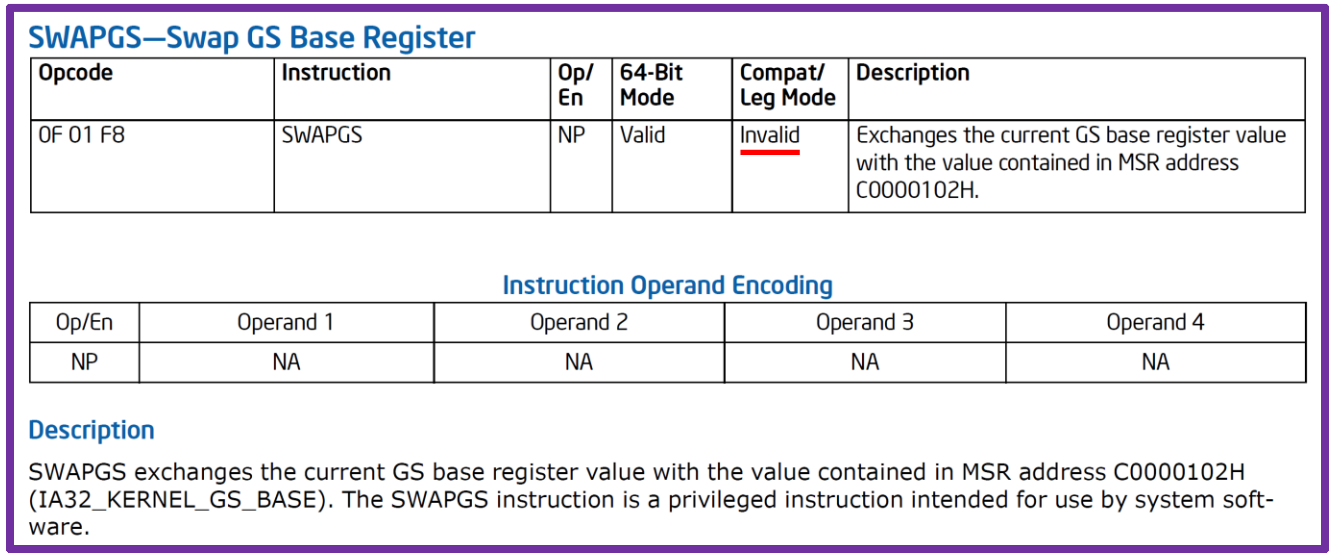
Per-CPU variables implementation in x86-64

- From the Intel manual:



3.4.2.1 Segment Registers in 64-Bit Mode

In 64-bit mode: CS, DS, ES, SS are treated as if each segment base is 0, regardless of the value of the associated segment descriptor base. This creates a flat address space for code, data, and stack. FS and GS are exceptions. Both segment registers may be used as additional base registers in linear address calculations (in the addressing of local data and certain operating system data structures).



SWAPGS—Swap GS Base Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 01 F8	SWAPGS	NP	Valid	Invalid	Exchanges the current GS base register value with the value contained in MSR address C0000102H.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

SWAPGS exchanges the current GS base register value with the value contained in MSR address C0000102H (IA32_KERNEL_GS_BASE). The SWAPGS instruction is a privileged instruction intended for use by system software.

Per-CPU on x86-64

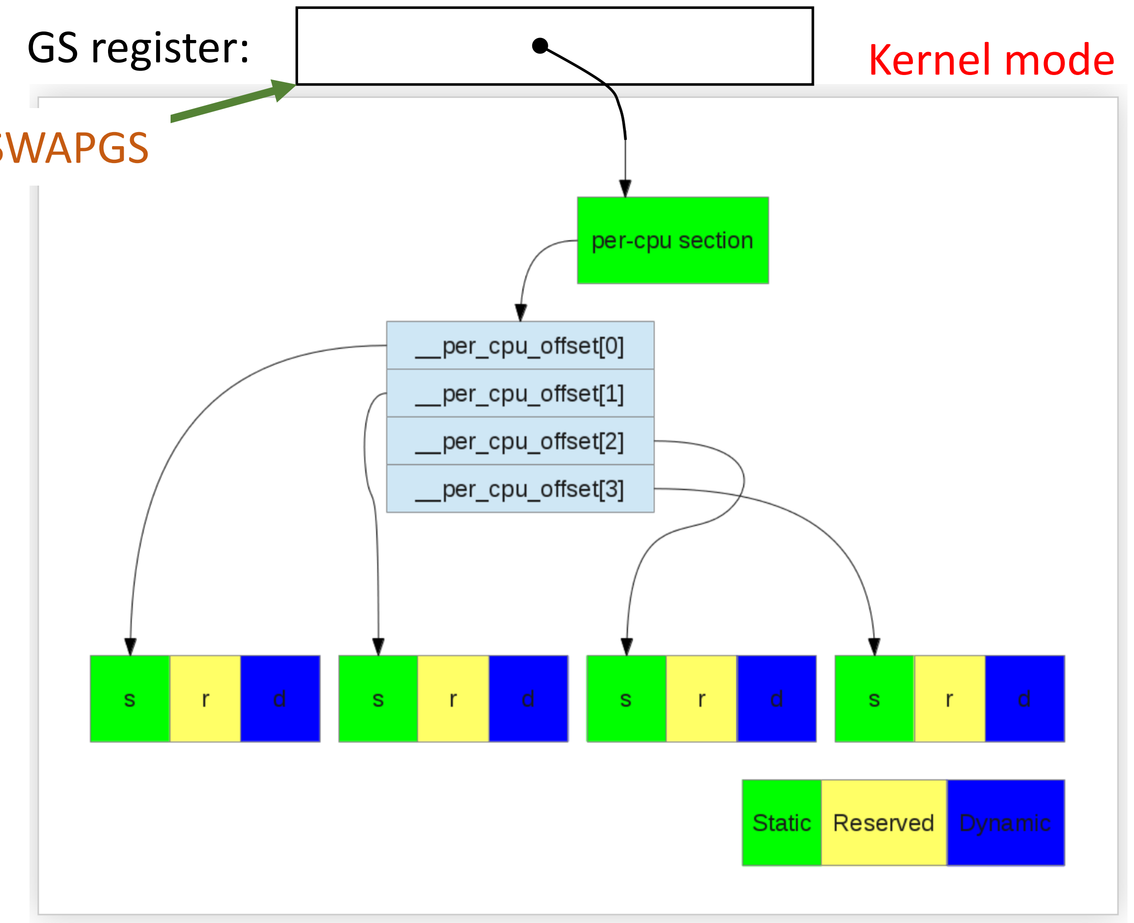
User mode

GS register: 0

SWAPGS

GS register:

Kernel mode



Translating a per-cpu variable to its real body (NR_CPUS = 4)

Source: http://thinkiii.blogspot.com/2014/05/a-brief-introduction-to-per-cpu.html

Atomic Operations

Atomic operators

- Simplest synchronization primitives
  - Primitive operations that are indivisible
- Two types
  - methods that operate on integers
  - methods that operate on bits
- Implementation
  - Assembly language sequences that use the atomic read-modify-write instructions of the underlying CPU architecture

Atomic integer operators

```
atomic_t v;
atomic_set(&v, 5);                     /* v = 5 (atomically) */
atomic_add(3, &v);                     /* v = v + 3 (atomically) */
atomic_dec(&v);                /* v = v - 1 (atomically) */
printf("This will print 7: %d\n", atomic_read(&v));
```

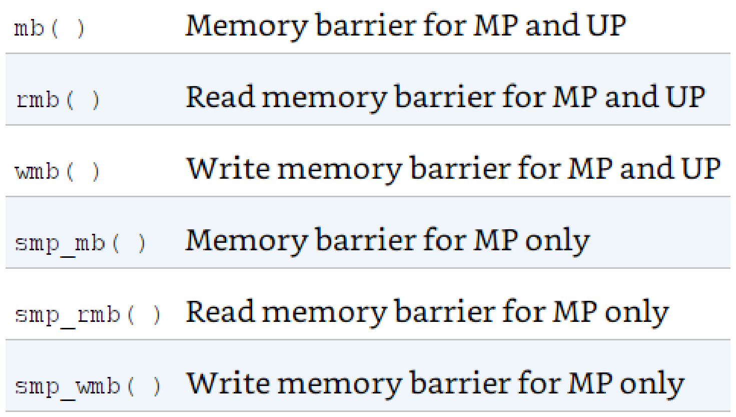
Beware:

- Can only pass atomic_t to an atomic operator
- atomic_add(3,&v); and
  {
  atomic_add(1,&v);
  atomic_add1(2,&v);
  }
  are not the same! … Why?

Memory Barriers

Barriers

- Compilers can reorder the sequence of instructions
- Superscalar processors may reorder reads and writes
- Optimization barrier
    - `barrier()`
    - Instruction to compiler: C statements before and after barrier are not to be mixed
- Memory barrier
    - Machine instructions issued to processor to explicitly prevent memory read-write reordering



`mb( )`	Memory barrier for MP and UP
`rmb( )`	Read memory barrier for MP and UP
`wmb( )`	Write memory barrier for MP and UP
`smp_mb( )`	Memory barrier for MP only
`smp_rmb( )`	Read memory barrier for MP only
`smp_wmb( )`	Write memory barrier for MP only

Spin Locks

Spin locks

- Mutual exclusion for larger (than one operator) critical sections requires additional support
- Spin locks are one possibility
  - Single holder locks
  - When lock is unavailable, the acquiring process keeps trying

Basic use of spin locks

```
spinlock_t mr_lock = SPIN_LOCK_UNLOCKED;
spin_lock(&mr_lock);          /* critical section ... */
spin_unlock(&mr_lock);
```

spin_lock()

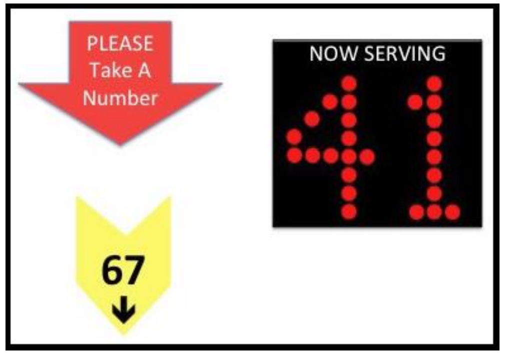
- Acquires the spinlock using atomic instructions required for SMP

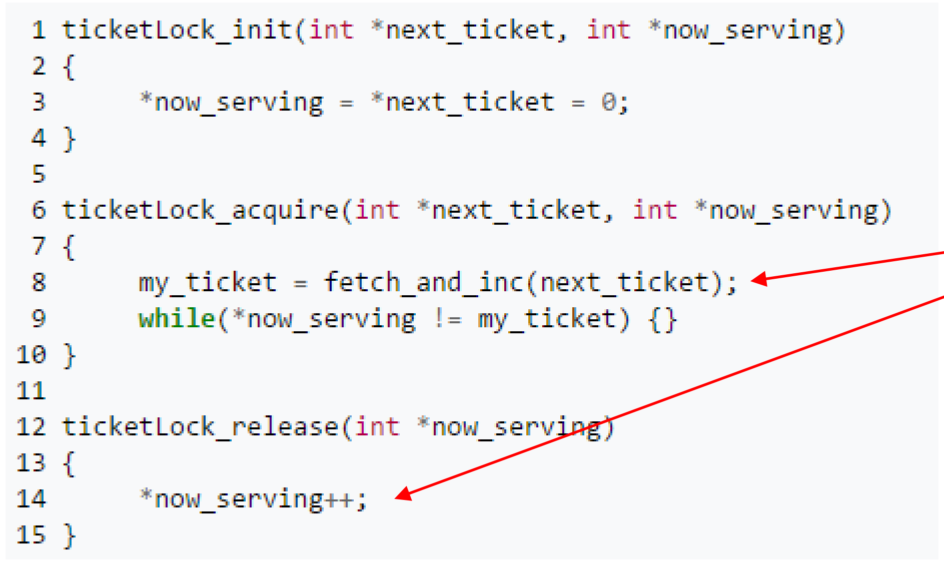
spin_unlock()

- Releases the spinlock

Ticket locks

- Spin locks in Linux implemented by ticket locks
- FIFO fairness





```
ticketLock_init(int *next_ticket, int *now_serving)
{
      *now_serving = *next_ticket = 0;
}

ticketLock_acquire(int *next_ticket, int *now_serving)
{
      my_ticket = fetch_and_inc(next_ticket);
      while(*now_serving != my_ticket) {}
}

ticketLock_release(int *now_serving)
{
      *now_serving++;
}
```

Atomic operations!

An Issue

- Interrupting a spin lock holder may cause several problems:
  - Spin lock holder is delayed, so is every thread spin waiting for the spin lock
    - Not a big problem if interrupt handlers are short
  - Interrupt handler may access the data protected by the spin-lock
    - Should the interrupt handler use the lock?
    - Can it be delayed trying to acquire a spin lock?
    - What if the lock is already held by the thread it interrupted?

Solutions

- If data is only accessed in interrupt context and is local to one specific CPU we can use interrupt disabling to synchronize

- If data is accessed from other CPUs we need additional synchronization
  - Spin locks
  - Spin locks can not be acquired in interrupt context because this might deadlock

- Normal code (kernel context) must disable interrupts and acquire spin lock
  - interrupt context code need not acquire spin lock
  - assumes data is not accessed by interrupt handlers on different CPUs, i.e., interrupts are CPU-local and this is CPU-local data

Combining spin locks and interrupt disabling

- Non-interrupt code acquires spin lock to synchronize with other non-interrupt code and disables interrupts to synchronize with local invocations of the interrupt handler

Combining spin locks and interrupt disabling

```
spinlock_t mr_lock = SPIN_LOCK_UNLOCKED;
unsigned long flags;
spin_lock_irqsave(&mr_lock, flags);  /* critical section ... */
spin_unlock_irqrestore(&mr_lock, flags);
```

`spin_lock_irqsave()`

- disables interrupts locally
- acquires the spinlock using instructions required for SMP

`spin_unlock_irqrestore()`

- Restores interrupts to the state they were in when the lock was acquired

Bottom halves and softirqs

- Softirqs, tasklets and BHs are deferrable functions
  - delayed interrupt handling work that is scheduled
  - they can wait for a spin lock without holding up devices
  - they can access non-CPU local data
- Softirqs – the basic building block
  - statically allocated and non-preemptively scheduled
  - can not be interrupted by another softirq on the same CPU
  - can run concurrently on different CPUs, and synchronize with each other using spin-locks
- Bottom Halves
  - built on softirqs
  - can not run concurrently on different CPUs

Spin locks and deferred functions

- `spin_lock_bh()`
  - implements the standard spinlock
  - disables softirqs
  - needed for code outside a softirq that manipulates data also used inside a softirq
  - Allows the softirq to use non-preemption only
- `spin_unlock_bh()`
  - Releases the spinlock
  - Enables softirqs

Spin lock rules

- Do not try to re-acquire a spinlock you already hold!
    - It leads to self deadlock!
- Spinlocks should not be held for a long time
    - Excessive spinning wastes CPU cycles!
    - What is “a long time”?
- Do not sleep while holding a spinlock!
    - Someone spinning waiting for you will waste a lot of CPU
    - never call any function that touches user memory, allocates memory, calls a semaphore function or any of the schedule functions while holding a spinlock! All these can block.

Semaphores

Semaphores

- Semaphores are locks that are safe to hold for longer periods of time
  - Contention for semaphores causes blocking not spinning
  - Should not be used for short duration critical sections
  - Semaphores are safe to sleep with!
    - Can be used to synchronize with user contexts that might block or be preempted

- Semaphores can allow concurrency for more than one process at a time, if necessary
  - i.e., initialize to a value greater than 1

Semaphore implementation

- Implemented as a wait queue and a usage count
  - Wait queue: list of processes blocking on the semaphore
  - Usage count: number of concurrently allowed holders
    - if negative, the semaphore is unavailable, and
    - absolute value of usage count is the number of processes currently on the wait queue
    - initialize to 1 to use the semaphore as a mutex lock

Semaphore operations

- Down()
  - attempts to acquire the semaphore by decrementing the usage count and testing if its negative
    - blocks if usage count is negative
- Up()
  - releases the semaphore by incrementing the usage count and waking up one or more tasks blocked on it

Can you be interrupted when blocked?

- down_interruptible()
  - Returns –EINTR if signal received while blocked
  - Returns 0 on success
- down_trylock()
  - attempts to acquire the semaphore
  - on failure it returns nonzero instead of blocking

Reader-writer Spinlocks

Reader/writer Locks

- No need to synchronize concurrent readers unless a writer is present
    - reader/writer locks allow multiple concurrent readers but only a single writer (with no concurrent readers)

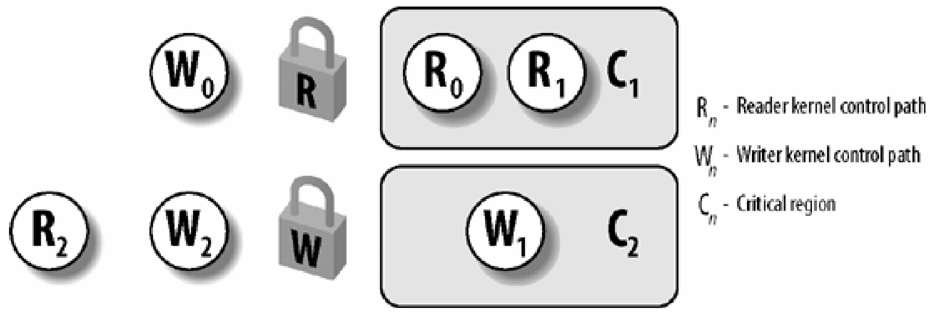
- Both spin locks and semaphores have reader/writer variants

Reader/writer spin locks (rwlock)

```
rwlock_t mr_rwlock = RW_LOCK_UNLOCKED;

read_lock(&mr_rwlock);
/* critical section (read only) ... */
read_unlock(&mr_rwlock);

write_lock(&mr_rwlock);
/* critical section (read and write) ... */
write_unlock(&mr_rwlock);
```


W_0
R
R_0
R_1
C_1
R_2
W_2
W
W_1
C_2
R_n - Reader kernel control path
W_n - Writer kernel control path
C_n - Critical region

Reader/writer lock warnings

- Reader locks cannot be automatically upgraded to the writer variant
  - Attempting to acquire exclusive access while holding reader access will deadlock!
  - If you know you will need to write eventually
    - Obtain the writer variant of the lock from the beginning
    - Or, release the reader lock and re-acquire the lock as a writer
      - But bear in mind that memory may have changed when you get in!

Reader/writer semaphores (rw_semaphore)

```
struct rw_semaphore mr_rwsem;
init_rwsem(&mr_rwsem);

down_read(&mr_rwsem);  /* critical region (read only) ... */
up_read(&mr_rwsem);

down_write(&mr_rwsem); /* critical region (read and write) ... */
up_write(&mr_rwsem);
```

Big reader locks (br_lock)

- Specialized form of reader/writer lock
  - Very fast to acquire for reading
  - Very slow to acquire for writing
  - Good for read-mostly scenarios

- Implemented using per-CPU locks
  - Readers acquire their own CPU’s lock
  - Writers must acquire all CPUs’ locks

Seqlocks

Seqlock

- Introduced in Linux 2.6
- Similar to Read-Write Locks but writers get much higher priority
- Pros:
    - Writer almost never waits (unless more than one writer)
- Cons:
    - Readers have to retry

Seqlock Data Structure

- Two fields:
    - A spin lock lock
    - An integer **`sequence`** field
- Reader must read the sequence field twice (before and after reading the data)
    - Spinlock not used!

```
do {
    seq = read_seqbegin(&seqlock);
            ...
} while (read_seqretry(&seqlock, seq));
```

Seqlock writer

- Writer must
  - Acquire spinlock and increase the sequence field (in **`write_seqlock()`**)
  - Do its write
  - Increase the sequence field once more then release spinlock (in **`write_sequnlock()`**)
  - Hence, before and after write, sequence field is even

True if sequence is odd <u>or</u>
the two sequences don't match

```
do {
    seq = read_seqbegin(&seqlock);
              ...
} while (read_seqretry(&seqlock, seq));
```

Conditions for using seqlocks

- The data structure to be protected does not include pointers that are modified by the writers and dereferenced by the readers
  - Otherwise, a writer could change the pointer under the nose of the readers
  -
- The code in the critical regions of the readers does not have side effects
  - Otherwise, multiple reads would have different effects from a single read

Big Kernel Lock

(Not in use anymore)

Big kernel lock (BKL)

- A global kernel lock - **`kernel_flag`**
  - Used to be the only SMP lock
  - Removed since 2.6.39
  - Replaced with fine-grain localized locks

- Linux distributions at or above CentOS 7, Debian 7 (Wheezy) and Ubuntu 11.10 not longer use BKL.

Preemptible kernel issues

- Have to be careful of legacy code that assumes per-CPU data is implicitly protected from preemption
    - Legacy code assumes “non-preemption in kernel mode”
    - May need to use new preempt_disable() and preempt_enable() calls
    - Calls are nestable
        - for each n preempt_disable() calls, preemption will not be re-enabled until the nth preempt_enable() call

Read-copy Update

Read-Copy Update (RCU)

- Designed for multi CPU operation
- Allows many readers and many writers to proceed concurrently
- RCU is lock free
- Only data structures that are dynamically allocated and referenced by means of pointers can be protected by RCU
- No kernel control path can sleep inside a critical region protected by RCU

RCU Read

- **`rcu_read_loc()`** is just **`preempt_disable()`**
  - Dereference a pointer and read
- **`rcu_write_lock()`** is just **`preempt_enable()`**
- Real work done by writers

RCU Writes

- Writer makes a copy of data structure by dereferencing the pointer
  - All changes made strictly to the copy, never to the source
- Once finished, writer changes the pointer to the data structure atomically
- So readers only see old or new copy of data structure but not both
- We have a new problem: when can we free the old data structure?

Everyone must cooperate

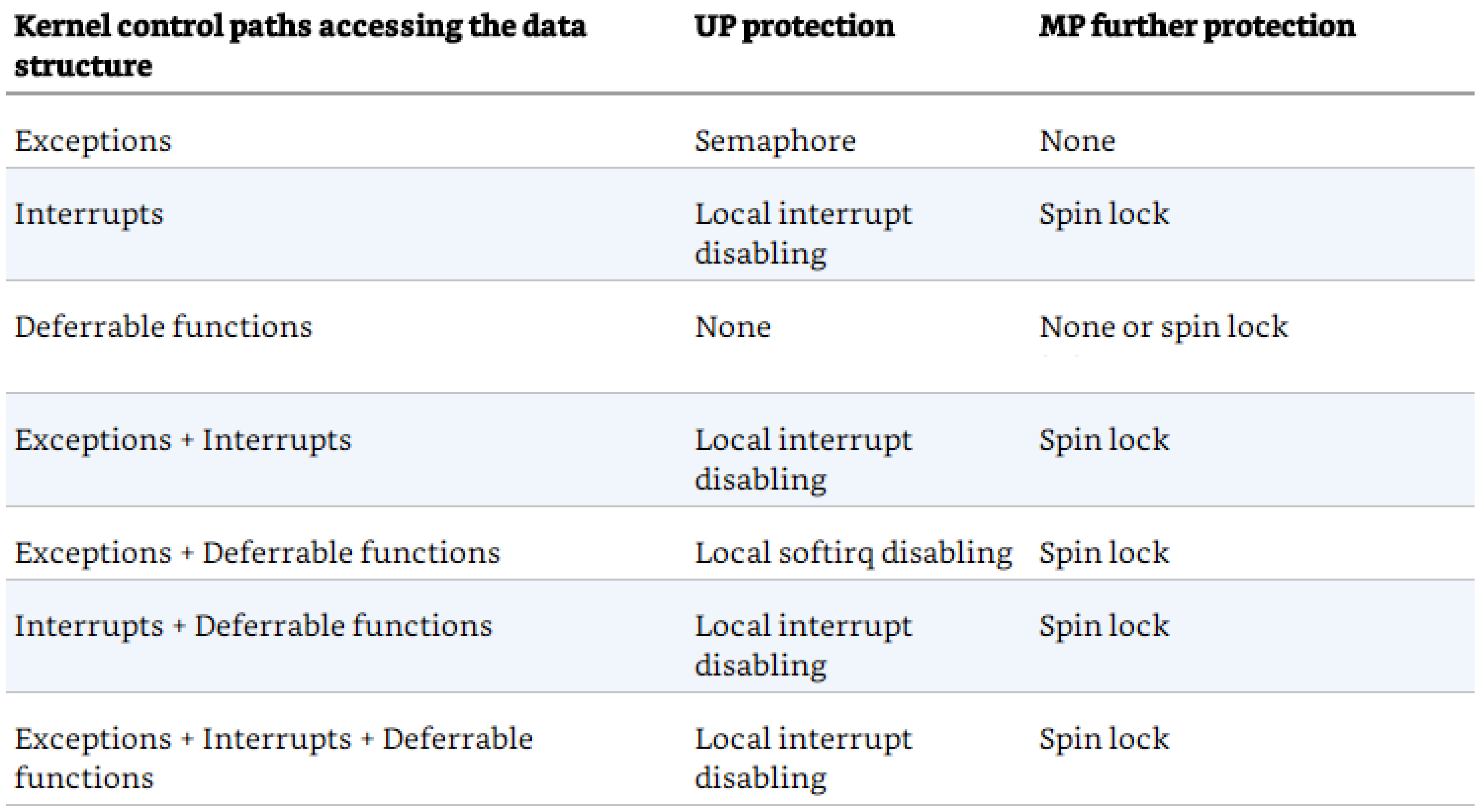
- The kernel requires every potential reader to execute `rcu_read_unlock()` before:
  - The CPU performs a process switch
  - The CPU starts executing in User Mode
  - The CPU executes the idle loop

- RCU introduced in Linux 2.6 and used in the networking layer and in the Virtual File System

Completion

- Introduced in Linux 2.6
- Solves a problem with semaphores
  - **`up()`** and **`down()`** can be done in parallel
  - A process can destroy a semaphore while some other process (say in another CPU) is in the middle of an operation
- **`complete()`** replaces **`up()`**
- **`wait_for_completion()`** replaces **`down()`**
- Spinlock is used to ensure that the two cannot be done concurrently

Choice of Protection



Kernel control paths accessing the data structure	UP protection	MP further protection
Exceptions	Semaphore	None
Interrupts	Local interrupt disabling	Spin lock
Deferrable functions	None	None or spin lock
Exceptions + Interrupts	Local interrupt disabling	Spin lock
Exceptions + Deferrable functions	Local softirq disabling	Spin lock
Interrupts + Deferrable functions	Local interrupt disabling	Spin lock
Exceptions + Interrupts + Deferrable functions	Local interrupt disabling	Spin lock

Lock-free Data Structures

Locks

- For thread synchronization
- Guards global data structures
- Ensure serializability
  - Even though there are multiple threads, the overall execution will behave like some serial sequencing of the thread’s execution
  - Critical sections (where two or more threads can interact) must be serialized

The problem with locks (1)

- Deadlock
- Priority Inversion
- Convoying
- “Async-signal-safety”
- Kill-tolerant availability
- Pre-emption tolerance
- Overall performance

The problem with locks (2)

- Deadlock
  - Infinite (circular) waits for locks
  - Absolutely no progress in computation

- Priority Inversion
  - Low priority thread gets lock required by high priority threads
  - Priority inheritance usually used to solve this

The problem with locks (3)

- Convoying
  - Slow thread gets the lock and blocks out fast ones
  - Overall performance suffers

- Signal safety
  - Signal handlers cannot use lock-based primitives
    - Suppose a thread receives a signal while holding a user level lock in the memory allocator
    - Signal handler executes, calls malloc, wants the lock

The problem with locks (4)

- Kill-tolerance
  - What happens if holder of a lock dies (crashed/killed)?
- Pre-emption tolerance
  - What happens if holder of a lock gets preempted?

The problem with locks (5)

- Performance
  - Contention pattern
  - Data structure
  - Granularity

Lock free programming

- Today's processors support atomic instructions
  - How else are locks themselves implemented?
- Designing generalized lock-free algorithms is hard
- Design lock-free data structures instead
  - Buffer, list, stack, queue, map, deque, snapshot

Compare-and-swap

- An atomic operation that comes in various flavours (depending on the processors)

```
int compare_and_swap (int* reg, int oldval, int newval)
{
  int old_reg_val = *reg;
  if (old_reg_val == oldval)
     *reg = newval;
  return old_reg_val;
}
```

Variant 1

```
bool compare_and_swap (int *accum, int *dest, int newval)
{
  if ( *accum == *dest ) {
      *dest = newval;
      return true;
  } else {
      *accum = *dest;
      return false;
  }
}
```

Variant 2

On x86

- Essentially the x86 `cmpxchg` instruction
- In Visual Studio, two functions:

```
LONG __cdecl InterlockedCompareExchange(
  __inout  LONG volatile *Destination,
  __in     LONG Exchange,
  __in     LONG Comparand
);
```

```
LONGLONG __cdecl InterlockedCompareExchange64(
  __inout  LONGLONG volatile *Destination,
  __in     LONGLONG Exchange,
  __in     LONGLONG Comparand
);
```

64-bit version

A simple example – lock-free stack

```
struct Node {
     Node * next;
     int data;
};

Node * head;
```

- A LIFO linked list instead of an array
- Can generalize `data` to anything

Lock-free stack push

```
void push(int t) {

    struct Node* node = (struct Node*)(malloc(sizeof(struct Node)));

    node->data = t;

    do {
        node->next = head;
    } while (!cas(&head, node->next, node));
}
```

```
int cas(int *loc, int oldval, int newval)
{
    if (*loc == oldval) {
        *loc = newval;
        return true;
    } else
        return false;
}
```

- `cas` performs the compare and swap
- Thread-safe
  - Any number of threads can call this concurrently

Lock-free pop

```
bool pop(int& t) {
  struct Node* current = head;

  while(current) {
    if(cas(&head, current, current->next)) {
      t = current->data; // problem?
      return true;
    }
    current = head;
  }
  return false;
}
```

The ABA Problem

- Thread 1 looks at some shared variable, finds that it is ‘A’
- Thread 1 calculates some interesting thing based on the fact that the variable is ‘A’
- Thread 2 executes, changes variable to B
- If Thread 1 wakes up now and tries to compare-and-set, all is well – compare and set fails and Thread 1 retries
- Instead, Thread 2 changes variable back to A!
- OK if the variable is just a value, but what if it is a pointer?

The ABA in lock-free stack

Thread 1: pop()

read A from head

store A.next `somewhere’

Thread 2:

pop()

pops A, discards it

First element becomes B
memory manager recycles
‘A’ into new variable
Pop(): pops B
Push(head, A)

cas with A succeeds

Work-around for ABA problem

- Keep a ‘update count’ (needs ‘doubleword CAS’)
  - Increment it during a successful update
- Don’t recycle the memory ‘too soon’

The ABA in lock-free stack

Thread 1: pop()

read A and a tag from head

store A.next `somewhere’

Thread 2:

pop()

pops A, discards it

First element becomes B
memory manager recycles
‘A’ into new variable
Pop(): pops B
Push(head, A + newtag)

cas with A and tag succeeds

LL/SC

- Load-link returns the current value of a memory location.
- A subsequent store-conditional to the same memory location will store a new value only if no updates have occurred to that location since the load-link.
- Implemented in Alpha, PowerPC, MIPS, and ARM

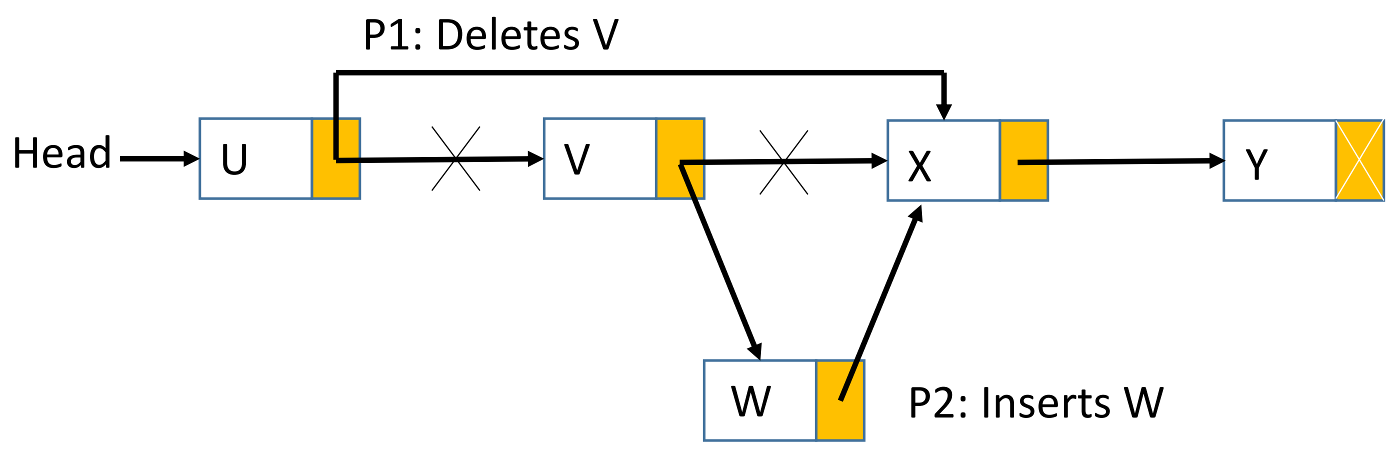
Features of lock-free data structures

- No overhead

- Common lock-free technique: atomically switching pointers
  - A small global data is guarded

- No API possible to ‘hold lock’

CAS alone is insufficient



P1: CAS(U->next, V, X) and P2: CAS(V->next, X, W)

End result: from the head, W is not seen, i.e., not inserted

Harris’ solution

- Harris, “A pragmatic implementation of non-blocking linked-lists”, 2001 (15th International Symposium on Distributed Computing)
  - Place a ‘mark’ in the next pointer of the soon-to-be deleted node
  - Easy on aligned architectures (free couple of low-order bits in most pointers)
  - Always fail if we try to CAS this (doesn’t look like a real pointer)
  - If we detect this problem, restart
  - Have to go back to the start of the list (we’ve ‘lost our place’)

Summary

- Lock-free programming can produce good performance
- Not easy to do in practice
  - Performance
  - Correctness (ABA problem)
- Well-established, tested, tuned implementations of common data structures are available
- Linux kernel uses lock-free data structures

A good reference site

http://www.rossbencina.com/code/lockfree

- Public domain lock-free data structure libraries you can use

End