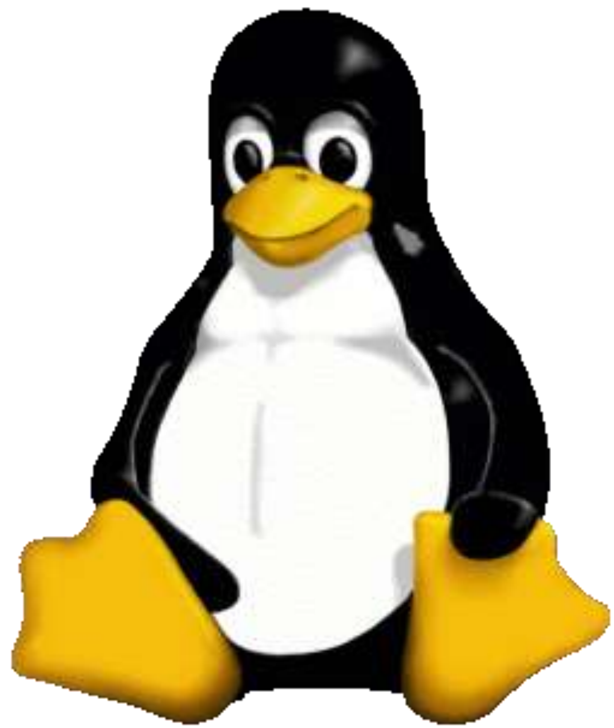
CS353 Linux Kernel

Chentao Wu吴晨涛
Associate Professor
Dept. of CSE, SJTU
wuct@cs.sjtu.edu.cn

4. Kernel Synchronization

Chentao Wu
Associate Professor
Dept. of CSE, SJTU
wuct@cs.sjtu.edu.cn



Outline

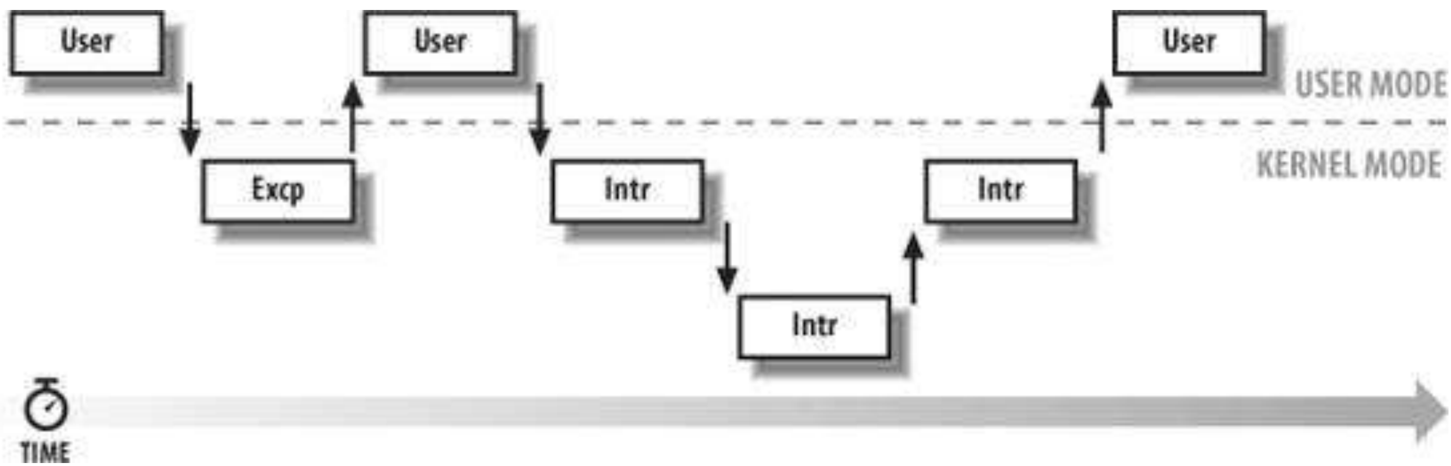
- Kernel Control Paths
- When Synchronization is not Necessary
- Synchronization Primitives
- Synchronizing Accesses to Kernel Data Structures
- Examples of Race Condition Prevention

Kernel Control Paths

- Linux kernel: like a server that answers requests
  - Parts of the kernel are run in interleaved way
- A *kernel control path*: a sequence of instructions executed in kernel mode on behalf of current process
  - Interrupts or exceptions
  - Lighter than a process (less context)

Example Kernel Control Paths

- Three CPU states are considered
  - Running a process in User Mode (User)
  - Running an exception or a system call handler (Excp)
  - Running an interrupt handler (Intr)



Kernel Preemption

- *Preemptive kernel*: a process running in kernel mode can be replaced by another process while in the middle of a kernel function
- The main motivation for making a kernel preemptive is to reduce the *dispatch latency* of the user mode processes
  - Delay between the time they become runnable and the time they actually begin running
- The kernel can be preempted only when it is executing an exception handler (in particular a system call) and the kernel preemption has not been explicitly disabled

When Synchronization in Necessary

- A *race condition* can occur when the outcome of a computation depends on how two or more interleaved kernel control paths are nested
- To identify and protect the *critical regions* in exception handlers, interrupt handlers, deferrable functions, and kernel threads
  - On single CPU, critical region can be implemented by disabling interrupts while accessing shared data
  - If the same data is shared only by the service routines of system calls, critical region can be implemented by disabling kernel preemption while accessing shared data
- Things are more complicated on multiprocessor systems
  - Different synchronization techniques are necessary

When Synchronization is not Necessary

- The same interrupt cannot occur until the handler terminates
- Interrupt handlers and softirqs are non-preemptable, non-blocking
- A kernel control path performing interrupt handling cannot be interrupted by a kernel control path executing a deferrable function or a system call service routine
- Softirqs cannot be interleaved

Synchronization Primitives

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

Per-CPU Variables

- The simplest and most efficient synchronization technique consists of declaring kernel variables as *per-CPU variables*
  - an array of data structures, one element per each CPU in the system
  - A CPU should not access the elements of the array corresponding to the other CPUs
- While per-CPU variables provide protection against concurrent accesses from several CPUs, they do not provide protection against accesses from asynchronous functions (interrupt handlers and deferrable functions)
- Per-CPU variables are prone to race conditions caused by kernel preemption, both in uniprocessor and multiprocessor systems

Functions and Macros for the Per-CPU Variables

Macro/ function name	Description
DEFINE_PER_CPU(type, name)	Statically allocates a per-CPU array
per_cpu(name, cpu)	Selects the element for CPU of the per-CPU array
__get_cpu_var(name)	Selects the local CPU's element of the per-CPU array
get_cpu_var(name)	Disables kernel preemption, then selects the local CPU's element of the per-CPU array
put_cpu_var(name)	Enables kernel preemption
alloc_percpu(type)	Dynamically allocates a per-CPU array
free_percpu(pointer)	Releases a dynamically allocated per-CPU array
per_cpu_ptr(pointer, cpu)	Returns the address of the element for CPU of the per-CPU array

Atomic Operations (1)

- Atomic 80x86 instructions
  - Instructions that make zero or one aligned memory access
  - Read-modify-write instructions (inc or dec)
  - Read-modify-write instructions whose opcode is prefixed by the lock byte (0xf0)
  - Assembly instructions whose opcode is prefixed by a rep byte (0xf2, 0xf3) are not atmoic

Atomic Operations (2)

- Atomic_t type: 24-bit atomic counter
- Atomic operations in Linux:

Function	Description
atomic_read(v)	Return *v
atomic_set(v,i)	set *v to i
atomic_add(i,v)	add i to *v
atomic_sub(i,v)	subtract i from *v
atomic_sub_and_test(i,v)	subtract i from *v and return 1 if result is 0
atomic_inc(v)	add 1 to *v
atomic_dec(v)	subtract 1 from *v
atomic_dec_and_test(v)	subtract 1 from *v and return 1 if result is 0
atomic_inc_and_test(v)	add 1 to *v and return 1 if result is 0
atomic_add_negative(i,v)	add i to *v and return 1 if result is negative

Atomic Bit Handling Functions

Function	Description
test_bit(nr, addr)	return the nrth bit of *addr
set_bit(nr, addr)	set the nrth bit of *addr
clear_bit(nr, addr)	clear the nrth bit of *addr
change_bit(nr, addr)	invert the nrth bit of *addr
test_and_set_bit(nr, addr)	set nrth bit of *addr and return old value
test_and_clear_bit(nr, addr)	clear nrth bit of *addr and return old value
test_and_change_bit(nr, addr)	invert nrth bit of *addr and return old value
atomic_clear_mask(mask, addr)	clear all bits of addr specified by mask
atomic_set_mask(mask, addr)	set all bits of addr specified by mask

Memory Barriers

- When dealing with synchronization, instruction reordering must be avoided
- A memory barrier primitive ensures that the operations before the primitive are finished before starting the operations after the primitive
  - All instructions that operate on I/O ports
  - All instructions prefixed by lock byte
  - All instructions that write into control registers, system registers, or debug registers
  - A few special instructions, e.g. iret
  - lfence, sfence, and mfence instructions for Pentium 4

Memory Barriers in Linux

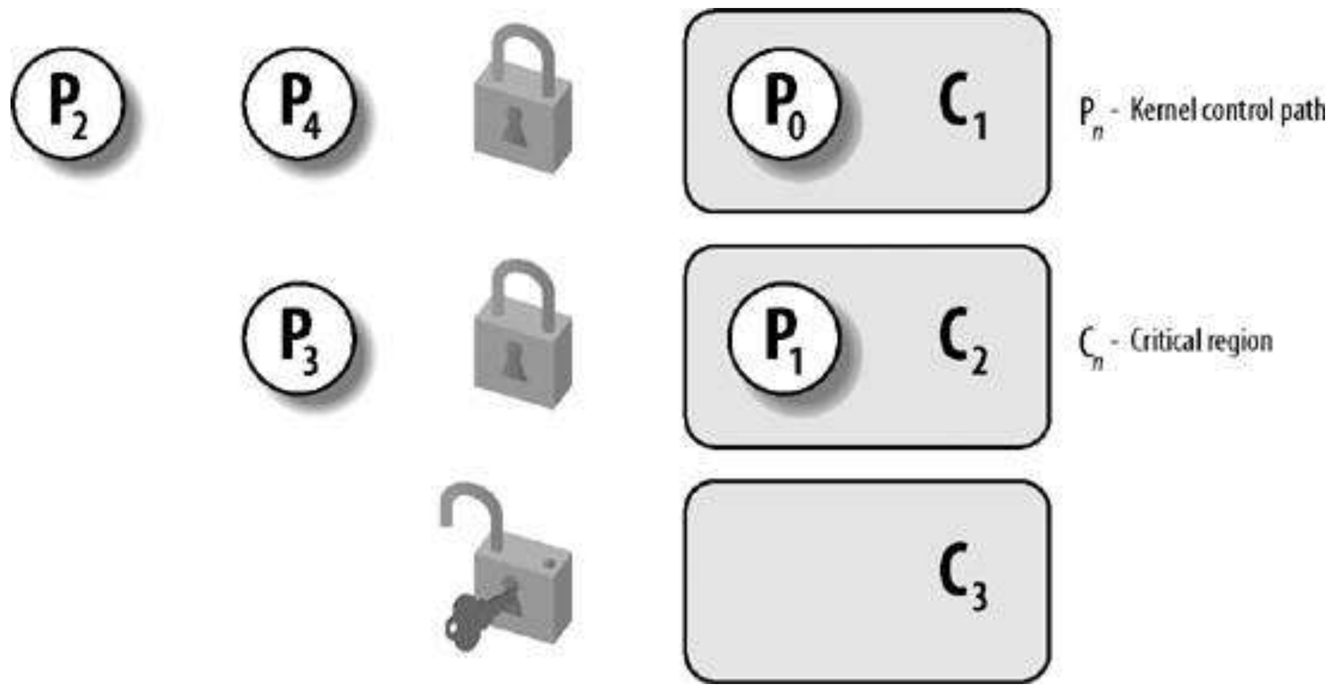
Macro	Description
mb()	Memory barrier for MP and UP
rmb()	Read memory barrier for MP, UP
wmb()	Write memory barrier for MP, UP
smp_mb()	Memory barrier for MP only
smp_rmb()	Read memory barrier for MP only
smp_wmb()	Write memory barrier for MP only

上海交通大學

Spin Locks

- Spin locks are a special kind of lock designed to work in a multiprocessor environment
  - Busy waiting
  - Very convenient
  - Represented by spinlock_t structure
    - slock: 1 – unlocked, <=0 - locked
    - break_lock: flag

Protecting Critical Regions with Several Locks



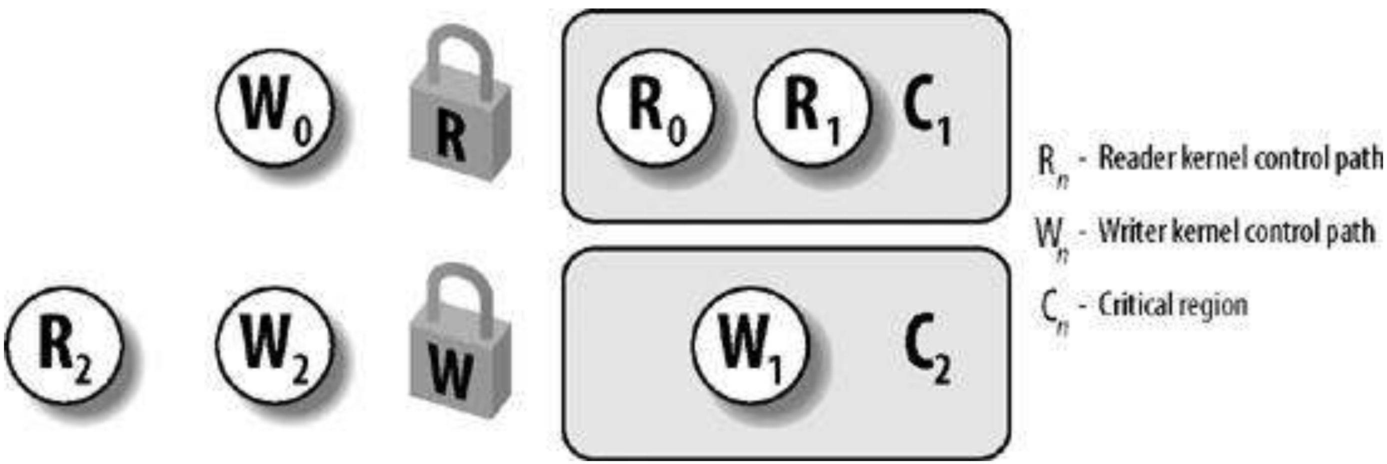
Spin Lock Macros

Macro	Description
spin_lock_init()	set the spinlock to 1 (unlocked)
spin_lock()	cycle until spin lock becomes 1, then set to 0
spin_unlock()	set the spin lock to 1
spin_unlock_wait()	wait until the spin lock becomes 1
spin_is_locked()	return 0 if the spin lock is set to 1
spin_trylock()	set the spin lock to 0 (locked), and return 1 if the lock is obtained

Read/Write Spin Locks

- To increase the amount of concurrency in the kernel
  - Multiple reads, one write
- rwlock_t structure
  - lock field: 32-bit
    - 24-bit counter: (bit 0-23) # of kernel control paths currently reading the protected data (in two's complement)
    - An unlock flag: (bit 24)
- Macros
  - read_lock()
  - read_unlock()
  - write_lock()
  - write_unlock()

Read/Write Spin Locks



Seqlock

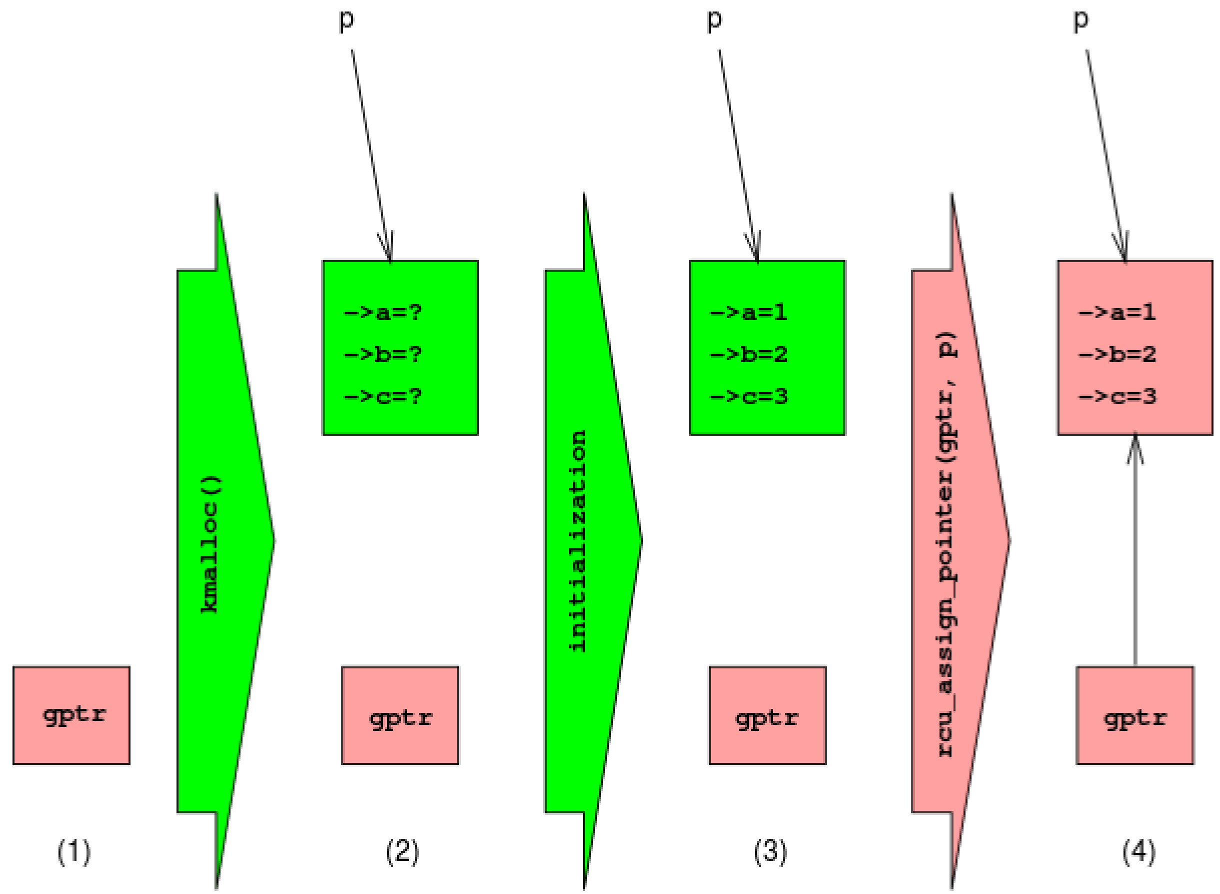
- *Seqlocks* introduced in Linux 2.6 are similar to read/write spin locks
  - except that they give a much higher priority to writers
  - a writer is allowed to proceed even when readers are active

Read-Copy Update (1)

- *Read-copy update* (*RCU*): another synchronization technique designed to protect data structures that are mostly accessed for reading by several CPUs
  - RCU allows many readers and many writers to proceed concurrently
  - RCU is lock-free
- Key ideas
  - Only data structures that are dynamically allocated and referenced via pointers can be protected by RCU
  - No kernel control path can sleep inside a critical section protected by RCU

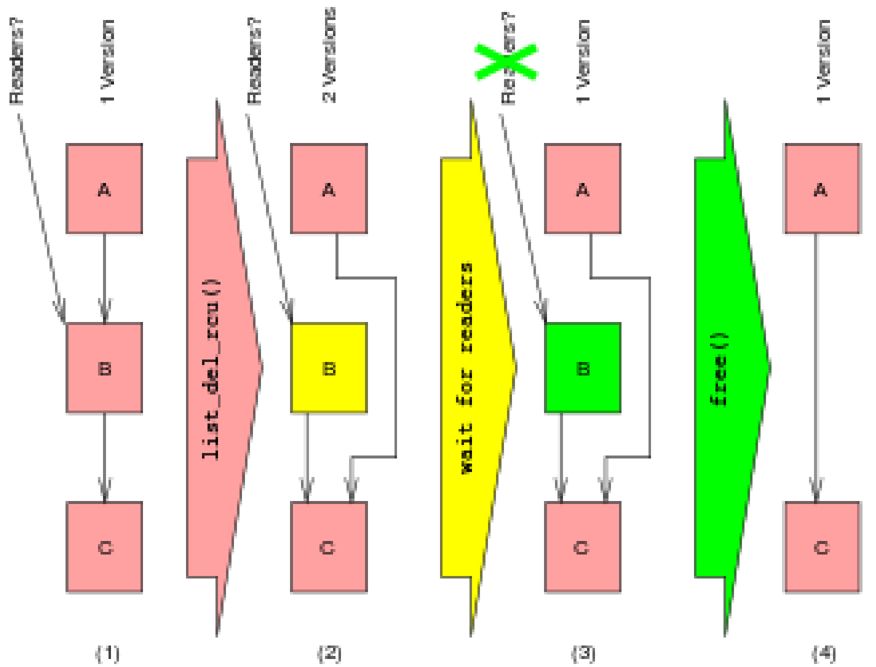
RCU (2)

- gptr: global pointer

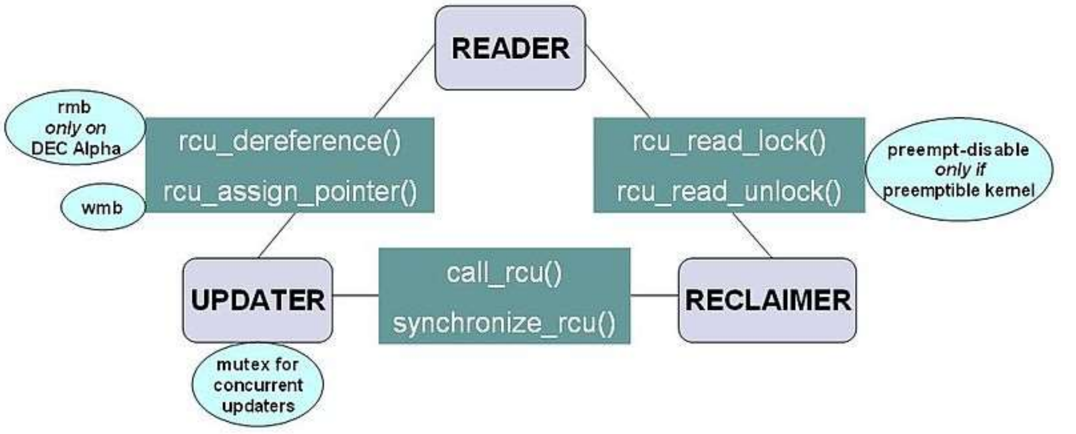


RCU (3)

- One copy



RCU (4)



- Macros
  - rcu_read_lock()
  - rcu_read_unlock()
  - call_rcu()
- RCU
  - New in Linux 2.6
  - Used in networking layer and VFS

Semaphores

- Two kinds of semaphores
  - Kernel semaphores: by kernel control paths
  - System V IPC semaphores: by user processes
- Kernel semaphores
  - struct semaphore
    - count
    - wait
    - sleepers
  - up(): to acquire a kernel semaphore (similar to signal)
  - down(): to release kernel semaphore (similar to wait)

Read/Write Semaphores

- Similar to read/write spin locks
  - except that waiting processes are suspended instand of spinning
- struct rw_semaphore
  - count
  - wait_list
  - wait_lock
- init_rwsem()
- down_read(), down_write(): acquire a read/write semaphore
- up_read(), up_write(): release a read/write semaphore

Completions

- To solve a subtle race condition in mutliprocessor systems
  - Similar to semaphores
- struct completion
  - done
  - wait
- complete(): corresponding to up()
- wait_for_completion(): corresponding to down()

Local Interrupt Disabling

- Interrupts can be disabled on a CPU with cli instruction
  - local_irq_disable() macro
- Interrupts can be enabled by sti instruction
  - local_irq_enable() macro

Disabling/Enabling Deferrable Functions

- "softirq"
- The kernel sometimes needs to disable deferrable functions without disabling interrupts
  - local_bh_disable() macro
  - local_bh_enable() macro

Synchronizing Accesses to Kernel Data Structures (1)

- Rule of thumb for kernel developers:
  - Always keep the concurrency level as high as possible in the system
  - Two factors:
    - The number of I/O devices that operate concurrently
    - The number of CPUs that do productive work

Synchronizing Accesses to Kernel Data Structures (2)

- A shared data structure consisting of a single integer value can be updated by declaring it as an atomic_t type and by using atomic operations
- Inserting an element into a shared linked list is never atomic since it consists of at least two pointer assignments

Choosing among Spin Locks, Semaphores, and Interrupt Disabling

Kernel control paths	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	local softirq disabling	spin lock
interrupts+deferrable	local interrupt disabling	spin lock
exceptions+interrupts+ deferrable	local interrupt disabling	spin lock

Interrupt-aware Spin Lock Macros

- spin_lock_irq(l), spin_unlcok_irq(l)
- spin_lock_bh(l), spin_unlock_bh(l)
- spin_lock_irqsave(l,f), spin_unlock_irqrestore(l,f)
- read_lock_irq(l), read_unlock_irq(l)
- read_lock_bh(l), read_unlock_bh(l)
- write_lock_irq(l), write_unlock_irq(l)
- write_lock_bh(l), write_unlock_bh(l)
- read_lock_irqsave(l,f), read_unlock_irqrestore(l,f)
- write_lock_irqsave(l,f), write_unlock_irqrestore(l,f)
- read_seqbegin_irqsave(l,f), read_seqretry_irqrestore(l,f),
- write_seqlock_irqsave(l,f), write_sequnlock_irqrestore(l,f)
- write_seqlock_irq(l), write_sequnlock_irq(l)
- write_seqlock_bh(l), write_sequnlock_bh(l)

上海交通大學

Examples of Race Condition Prevention (1)

- Reference counters: an atomic_t counter associated with a specific resource
- The global kernel lock (a.k.a big kernel lock, or BKL)
  - Lock_kernel(), unlock_kernel()
  - Mostly used in early versions, used in Linux 2.6 to protect old code (related to VFS, and several file systems)
- Memory descriptor read/write semaphore
  - mmap_sem field in mm_struct
- Slab cache list semaphore
  - cache_chain_sem semaphore
- Inode semaphore
  - i_sem field

Examples of Race Condition Prevention (2)

- When a program uses two or more semaphores, the potential for deadlock is present because two different paths could wait for each other
  - Linux has few problems with deadlocks on semaphore requests since each path usually acquire just one semaphore
  - In cases such as rmdir() and rename() system calls, two semaphore requests
  - To avoid such deadlocks, semaphore requests are performed in address order
    - Semaphore request are performed in predefined address order

Project Help:
Compile Linux Kernel

上海交通大學

Tips

- Accelerate compiling
  - http://www.freemindworld.com/blog/2010/100105_make_complie_process_faster.shtml
- Optimize the configuration of compiling
  - http://kenwublog.com/docs/linux-kernel-2-6-36-optimization.htm
- Other suggestions
  - Without graphics