

# Locking in Linux kernel

Galois @ USTC Linux Users Group

zyf11@mail.ustc.edu.cn

Slides are powered by  
OpenOffice.org+Linux+GNU+X31-150\$

Copyright © 2005 Galois Y.F. Zheng

Permissions is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license can be downloaded from GNU's home:

<http://www.gnu.org/licenses/fdl.txt>

# Locking in Linux kernel

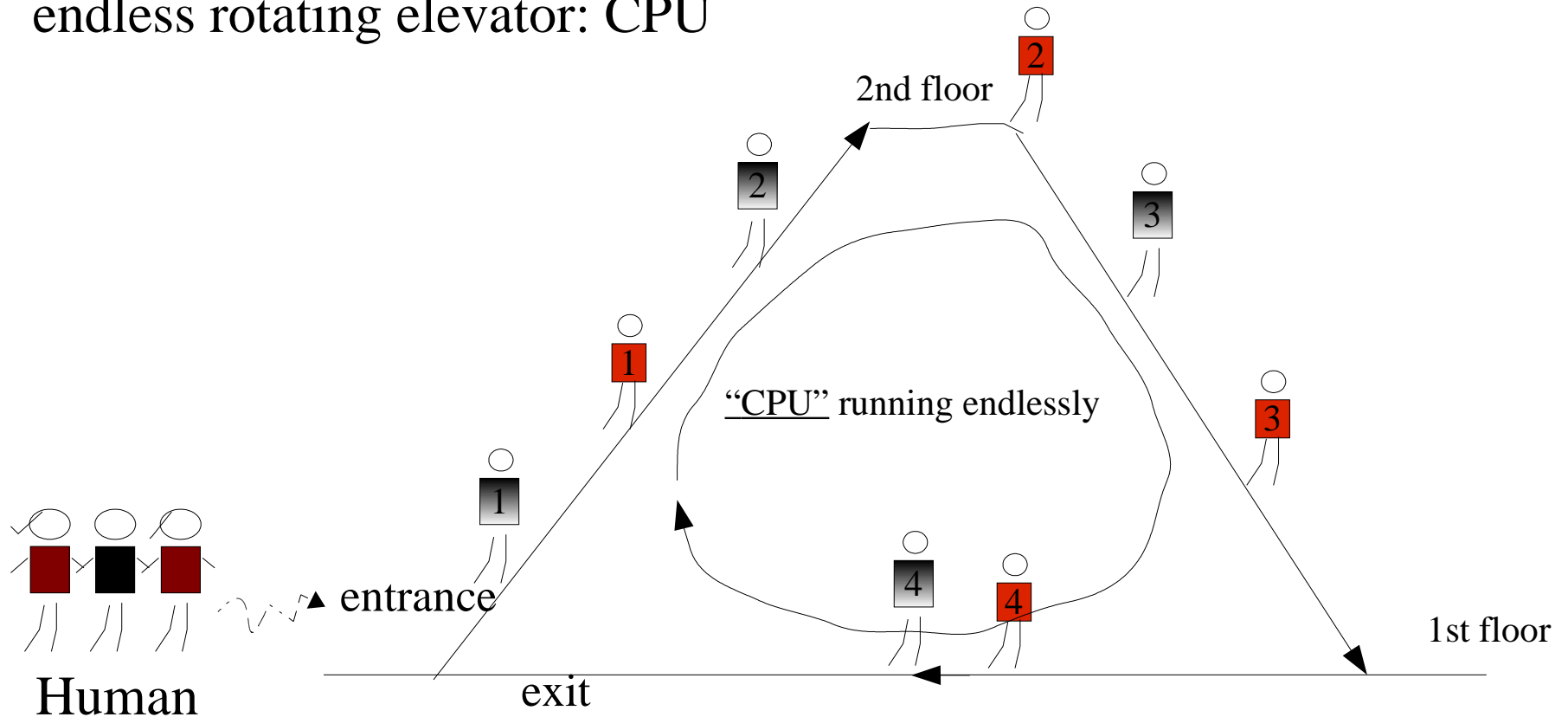
- OS review: Kernel Control Paths
- Locking in Linux
- Locking and Coding
- Conclusions


# OS review: Kernel Control Paths


cpu, operating system, and human





- CPU is stupid, running endlessly!
- But the codes in RAM is intelligent.  
(They compete for CPU resources.)
- We people control the codes.





# endless rotating elevator: CPU



 : Codes 1

 : Codes 2

  ...   : Kernel Control Path 1

  ...   : Kernel Control Path 2

# OS review: Kernel Control Paths

**Pre-KCP:** What is Kernel Control Path ?

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

**Post-KCP:** Is mm subsystem a KCP?

NO, But mm codes are called by KCP.

# OS review: Kernel Control Paths

What is the composition of a kernel?

- Kernel Control Paths(KCP).
- Kernel Data (global or local).
- Kernel Codes called by the KCPs.
- Bootstrap Codes, Initialization Codes, ...

*Now we need **Locking** (between KCPs), Let's GO!*

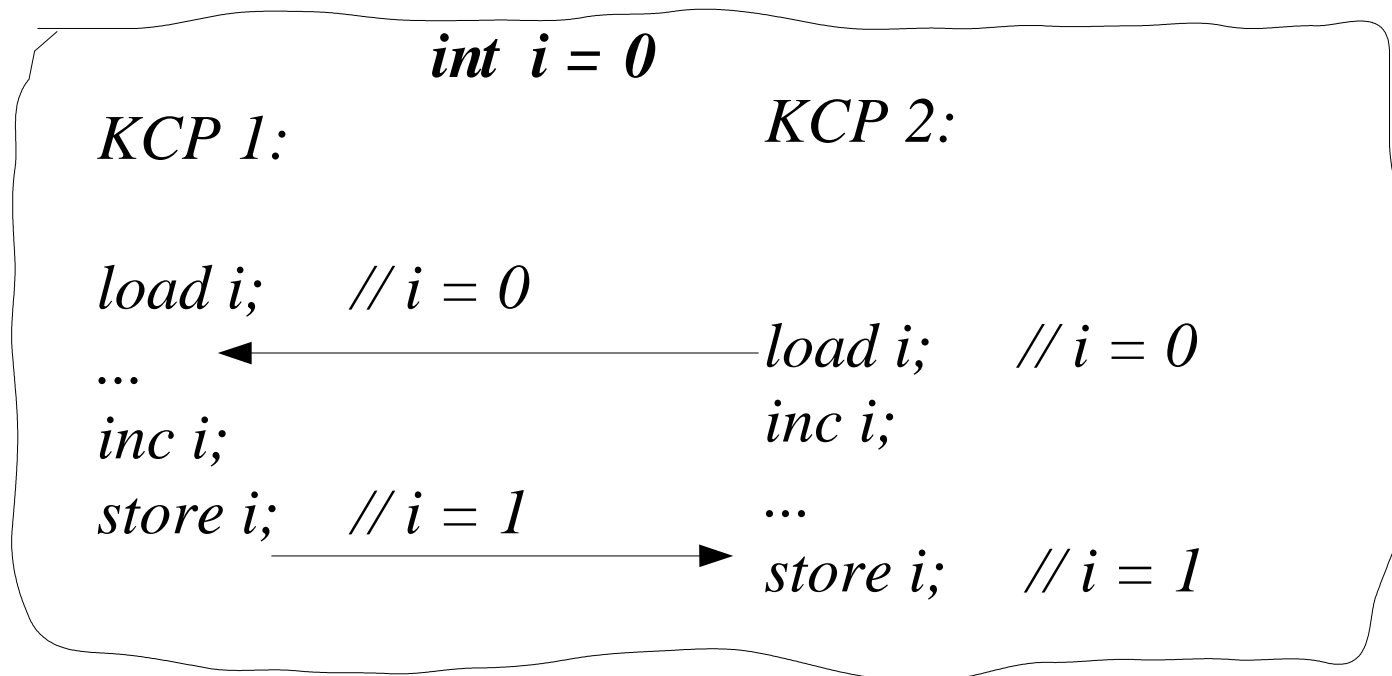
# Locking in Linux kernel

- OS review: Kernel Control Paths
- **Locking in Linux**
- Locking and Coding
- Conclusions



# Locking in Linux

What is Locking? A Simple example.



`i = 1, wrong`

The result is wrong because of accessing “i” at the same time.

# Locking in Linux

What is Locking? A Simple example. (cont.)

*int i = 0*

*KCP 1:*

*<locking starts>*

*load i; // i = 0*

*inc i;*

*store i; // i = 1*

*<locking ends>*

*KCP 2:*

*<locking starts>... failed.*

*waiting..*

*waiting..*

*<locking succeeds>*

*load i; //now i = 1*

*inc i;*

*store i; // i = 2*

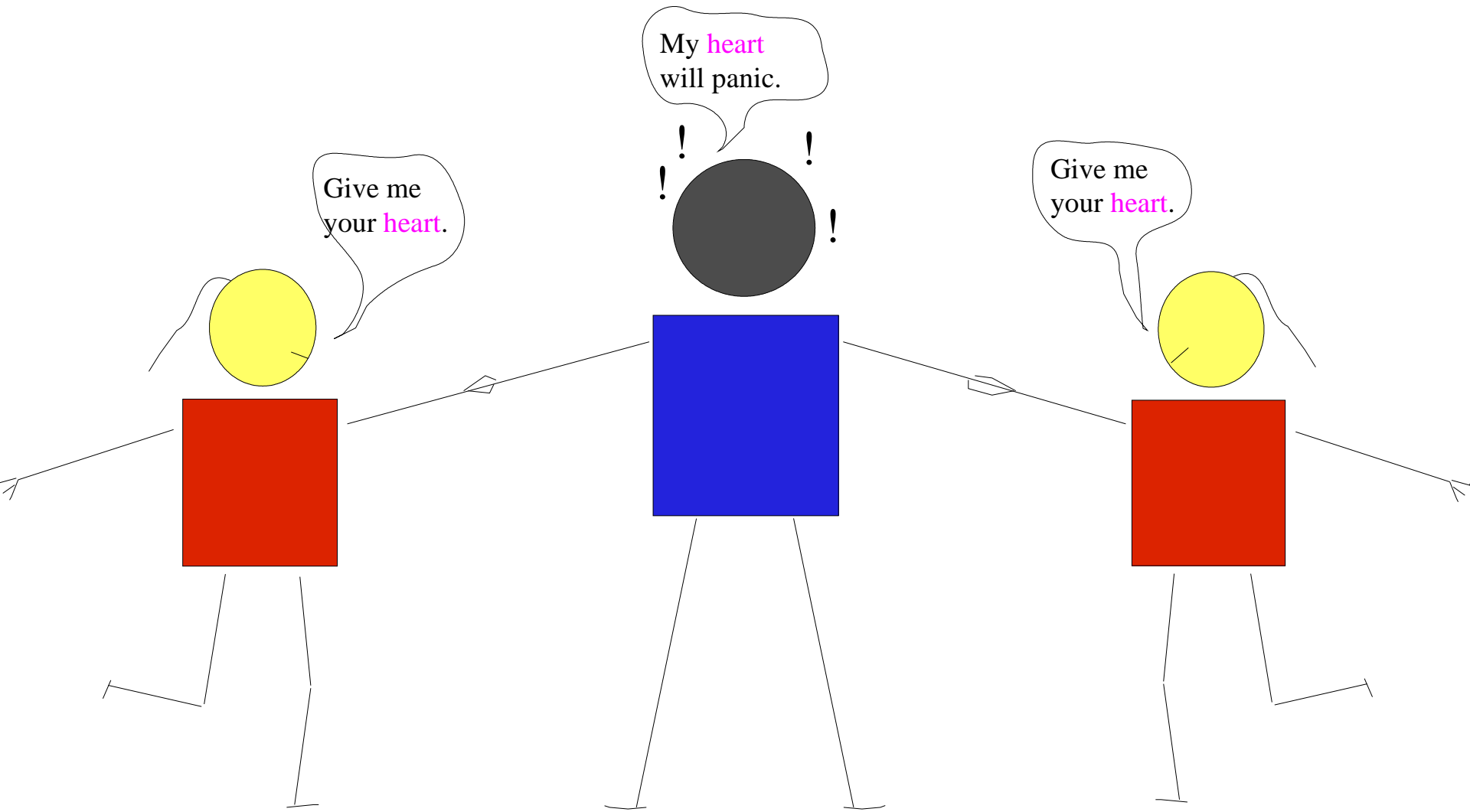
*<locking ends>*

*i = 2, right!*

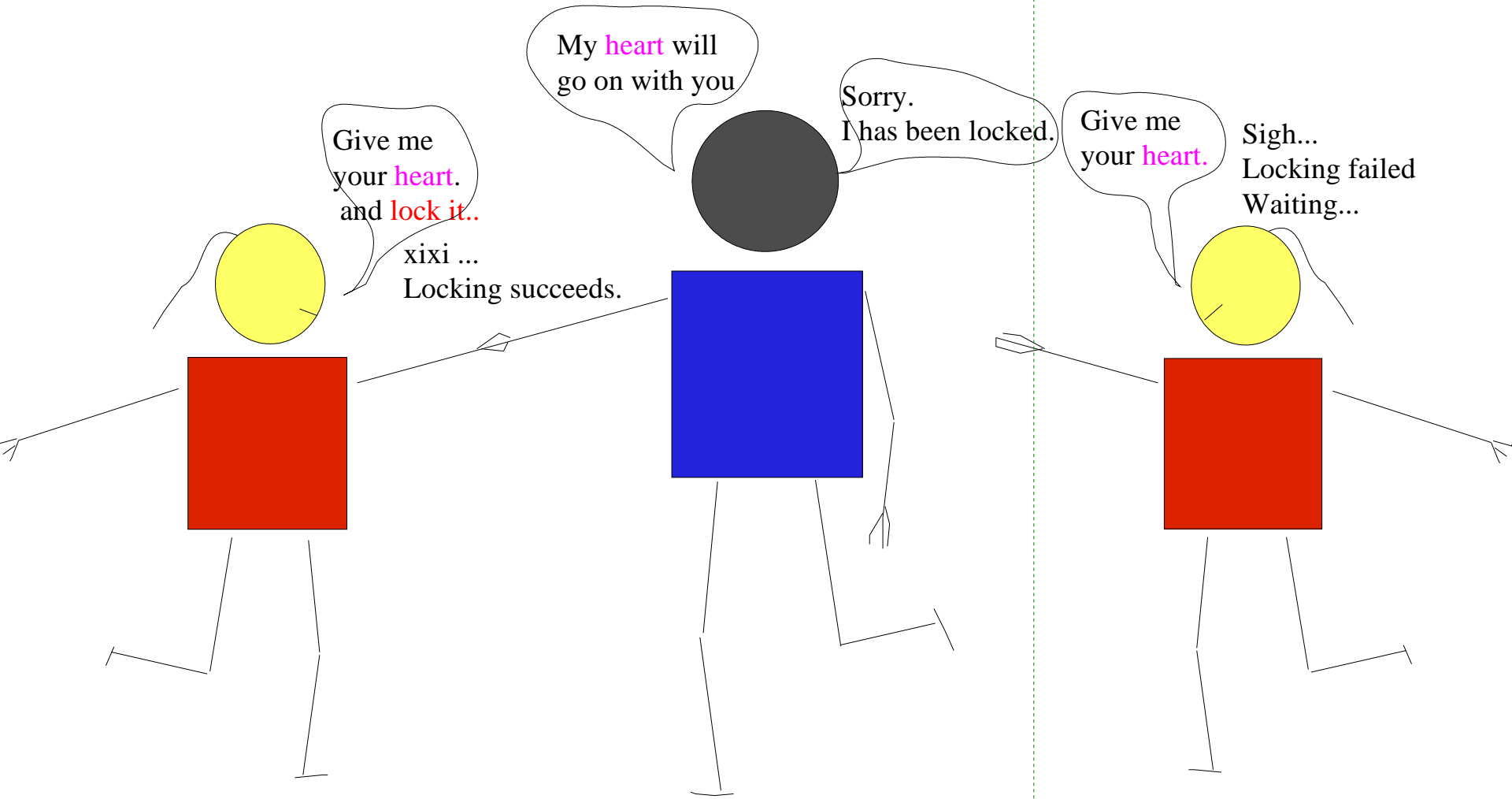
x86 lock directive.:

# Locking in Linux

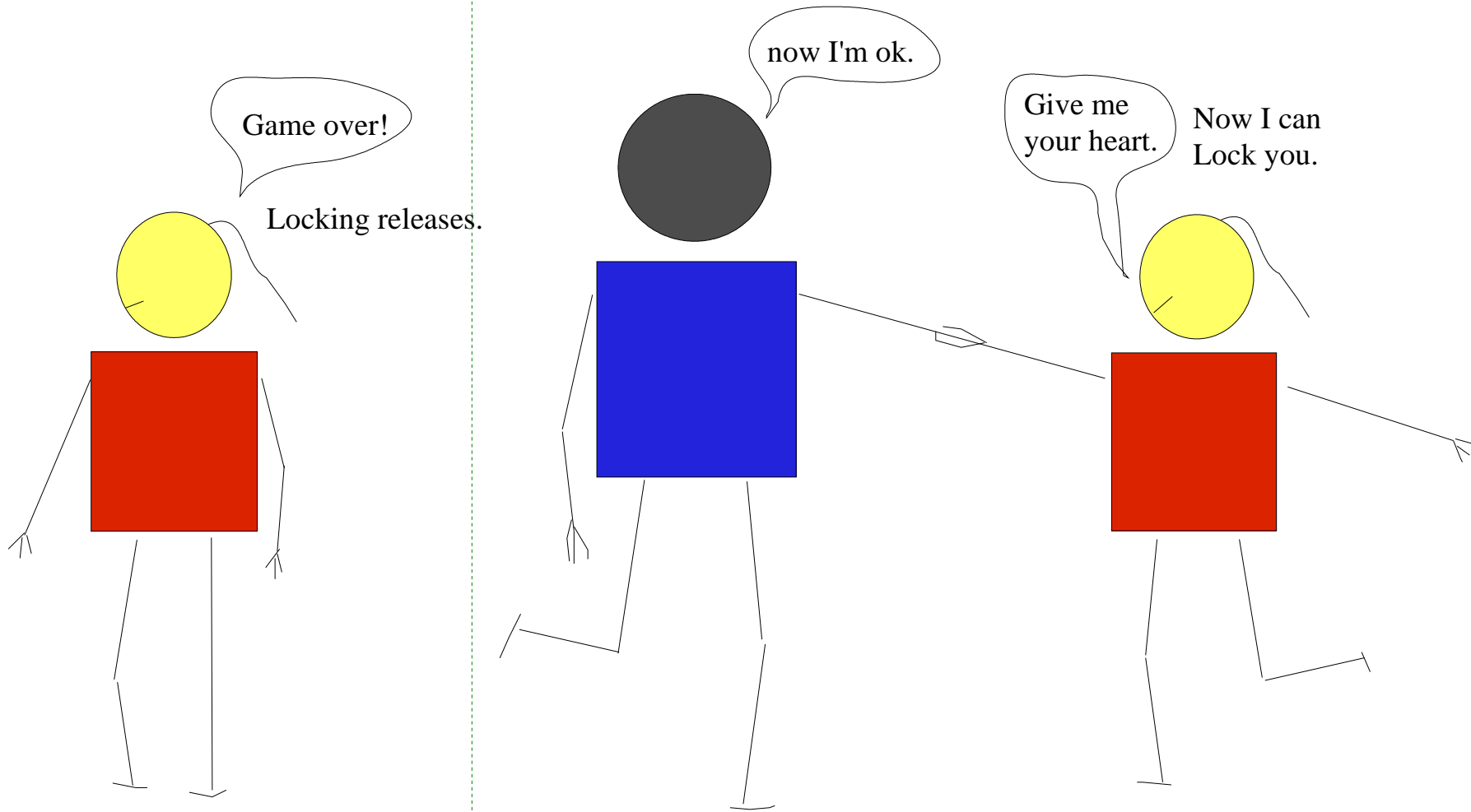
Another example...



Heart has no locking: a disordered world.



Locking: the world is well-ordered.



Locking: the world is well-ordered.

Now we know that  
**locking**  
makes the world romantic and beautiful.

# Locking in Linux

- What is Locking. (cont.)
  - Shared Data
  - Does Code need locking ? - Yes, surprising!
    - Critical Regions
  - Concurrency caused by Kernel Control Paths
    - Race Conditions ? - Must be avoided.
  - Now we needs Synchronization . - Locking.



# Locking in Linux

Concurrency and Locking : another example.

## KCP 1

- Locking the queue
- **Succeeded: acquired lock**
- Access queue
- **Unlock the queue**

## KCP 2

- Locking the queue
- **Failed: waiting...**
- Waiting...
- ...
- **Succeeded: acquired lock**
- Access queue
- **Unlock the queue.**

*Cited from LKD by R. Love*

# Locking in Linux

What Causes Concurrency?

- Interrupts and Exceptions.
- Sleeping and synchronization.
- Kernel Preemption.
- SMP ! - a hot topic, even in embedded apps.

# Locking in Linux

Locking(Sync.) is important.  
Let's get into the Locking details.

# Locking in Linux

Various Locking mechanisms.

- 1 Atomics operations
- 2 Memory barriers
- 3 Spin locks
- 4 Reader-writer spin locks
- 5 Semaphores
- 6 Reader-writer semaphores

# Locking in Linux

Various Locking mechanisms. (cont.)

- 7 Condition(Completion) Variables
- 8 Sequence locks
- 9 Mask Interrupts(local and global)
- 10 Mask Bottom Halves
- 11 Disable Kernel Preemption
- 12 Read-Copy Update
- 
- Big Kernel Lock                      - Historical, will be removed
- FUTEX ?                                - NO

# 1 Atomics Operations

- atomic ops is for the concurrency caused by MP. not for other concurrencies caused by preemption, sleep...
- atomic operations mechanisms(SMP env):
  - cpu guaranteed atomic ops: read/write a byte, alined word...
  - lock prefix: add, adc, and, cmpxchg, cmpxch8b, **dec, inc, neg, not,** or, sbb, sub ,xor, xadd, **btc,bts, btr**
  - **xchg is automatically added lock prefix.**
  - cache coherency protocols.

# 1 Atomics Operations

<asm/atomic.h> <asm/bitops.h>

- atomic integer ops: (on atomic\_t type v)
  - atomic\_read(v)      v->counter      not necessary
  - atomic\_set(v)      v->counter      not necessary
  - atomic\_add(i,v)      v->counter+i      lock;addl %1,%0
  - ...
- atomic bitwise ops:
  - set\_bit(i,addr)      set the i-th bit      lock;btsl %1,%0
  - clear\_bit(i,addr)      clear the i-th bit      lock;btrl %1,%0
  - test\_and\_set\_bit      lock;btsl %2,%1;sbb1 %1,%0
  - ...
- pseudo atomic bitwise ops: carefully!
  - \_\_set\_bit(), \_\_xxx() .....      there is no lock prefix.

# 2 Memory Barriers basics

- gcc optimizes instruction streams.
- 386 is strong ordering, where read and write are issued on the system bus in the order they occur..but pentium 4 is **processor ordering**, by which cpu could improve performance.
- memory barriers hardware technologies(x86):
  - serializing instructions
    - mov(to control register/debug register), wrmsr, invd, invlpg, wbinvd,lgdt,lldt,lidt,ltr;
    - cpuid,iret,rsm (non-privileged)
    - sfence(store), mfence(all), lfence(load) (non-privileged)
  - io instructions, read/write to uncached memory, interrupt occurrence, **lock prefix**
  - mtrr and pat could control memory ordering.



# 2 Memory Barriers Methods

<asm/system.h>

- `rmb()`, prevents loads being reordered
- `read_barrier_depends()`, prevents data-dependent loads being reordered.
- `wmb()`, prevents stores being reordered.
- `mb()`, prevents loads and stores being reordered.
- 
- `barrier()`, prevents GCC optimize loads and stores.
- 
- `smp_XXX()`, on smp, provides XXX; on up provides `barrier()`

Note: “XXX” refers to `rmb`, `wmb`...

# 3 Spin locks

<linux/spinlock.h><asm/spinlock.h>

- Spinning on SMP. Spinning is null on UP.
- Don't hold it for a long time. less than context switch time.
- spinlock automatically **disables preemption**, which avoids deadlock caused by interrupts.
- when data is shared with interrupt handler, before holding spinlock we must **disable interrupts**.
- when data is shared with bottom halves, before holding spinlock we must **disable bottom halves**.

# 4 Spin Locks

(cont.)

- `spin_lock()`            acquire lock
- `spin_unlock()`        release lock
- `spin_lock_irq()`      disable local interrupts and acquire lock
- `spin_unlock_irq()`
- `spin_lock_irqsave()`    save current state of ints, ...
- `spin_lock_irqrestore()`   restore....
- ...

# 5 Reader-writer spin locks

<asm/spinlock.h><linux/spinlock.h>

- Writing demands mutual exclusion.
- Multiple concurrent Readings is ok.
- When Reading, Writing must be disabled.
- 
- Reading locks and writing locks are seperated.
- read\_lock\_xxx()                      read\_unlock\_xxx()
- write\_lock\_xxx()                      write\_unlock\_xxx()
- ...                                      ...
- **Problems: This locks favor readers over writers, which may starve pending writers.**

# 6 Semaphores

<asm/semaphore.h><arch/xxx/kernel/semaphore.c>

- Checking (struct semaphore\*)->count, dec&inc is spinlocked.
- when initial count > 1, it allows arbitrary number of lock holders. when initial count = 1, it is binary semaphore, also called mutex which is used in many places.
- It is sleeping locks.
- Threads may sleep while holding semaphores.
- Threads can't acquire semaphores while holding spin lock.
- 
- down() threads get into uninterruptible state
- down\_interruptible(), threads get into interruptible state
- up() inc count, if count<=0, wake up waiting thread
- ...

# 7 Reader-writer semaphores

<linux/rwsem.h>

- WE can understand it.
- 
- `down_read()`, `down_read_trylock()`
- `up_read()`
- `down_write()`, `down_write_trylock()`
- `up_write()`
- 
- **NOTE: unlike rw-spinlock, we can downgrade from writelock to readlock.**

# Spin locks VS. semaphores

(recommended)

- low overhead locking, —————> spinlock
- short lock hold time , —————> spinlock
- long lock hold time , —————> semaphore
- **for interrupt context use,** —————> spin lock
- sleep while holding lock, —————> semaphore

# 8 Condition(Completion) Variables

<linux/completion.h><kernel/sched.c>

- It is a very **simple solution** to a problem that **semaphore** could resolve otherwise. but maybe it is not wise to fix semaphore.
- It just checks a condition to decide what to do: sleep(wake up) or continue(null). *sleeping+spinning==>cv*
- It is mainly for SMP.
- 
- only 2 functions:
- wait\_for\_completion()      if ok, then continue, else wait.
- complete()                    signal any waiting threads.



# Semaphore VS. Con. Variable

*down()*  
lock; dec %0  
...  
spin\_lock(sem->wait.lock)  
//..., wait queue ops;  
spin\_unlock(sem->wait.lock)

*wait\_for\_completion()*  
spin\_lock(cv->wait.lock)  
//wait queue ops;  
//may unlock spin and sleep  
//dec cv->done  
spin\_unlock(cv->wait.lock)

*up()*  
lock; inc %0  
...  
spin\_lock(sem->wait.lock)  
//..., wait queue ops;  
spin\_unlock(sem->wait.lock)

*complete()*  
spin\_lock(cv->wait.lock)  
//inc cv->done  
//wait queue ops;  
spin\_unlock(cv->wait.lock)

complex and seperated locking

simple and totally spinlocked

# 9 Sequence Locks

<linux/seqlock.h>

- For this situation: data has *many readers* and *a few writers*. like RCU mechanism
- Unlike reader-writer locks, seqlock favors writers over readers.
- Readers never blocks, but have to retry for arbitray times if a writer is in progress.
- Writers are mutually exclusive to change data, which is like spin locks. But writers do not wait for readers.

```
write_seqlock_xxx();  
// change data...  
write_sequnlock_xxx();
```

Writers

```
do {  
    seq = read_seqbegin_xxx(seq);  
    // read data ...  
} while (read_seqretry_xxx(seq))
```

Readers

# 10 Mask interrupts(local and global)

<linux/interrupt.h><asm/system.h><kernel/irq/manage.c><asm/processor.h>

- Deal with CPU IF flag. which disable all interrupts of local CPU (cli and sti instructions.)
- Masking PIC's irq line is another story. It makes serial execution of same interrupt. but it could not prevent the preemption from other interrupt.
- local\_irq\_disable(), local\_irq\_enable()
- Do you remember: spin\_lock\_irq() ? Disabling interrupts are used with spin\_lock().
- 
- Global disabling: cruel! I don't know wheather removed. but we can use synchronize\_irq() to synchronize all CPUs.

# 11 Mask Bottom Halves

<linux/interrupt.h>

- when data is shared with bottom halves, maybe we need to disable bottom halves.
- 
- `local_bh_disable()`, `local_bh_enable()`:
  - *calling `add_preempt_count()`*
- `spin_lock_bh()`

# 12 Kernel Preemption Disable

<linux/preempt.h>

- **preemption points:**
  - interrupt return path,
  - arbitrary preemption points in kernel codes.
- 
- **preempt\_disable() and preempt\_enable()**

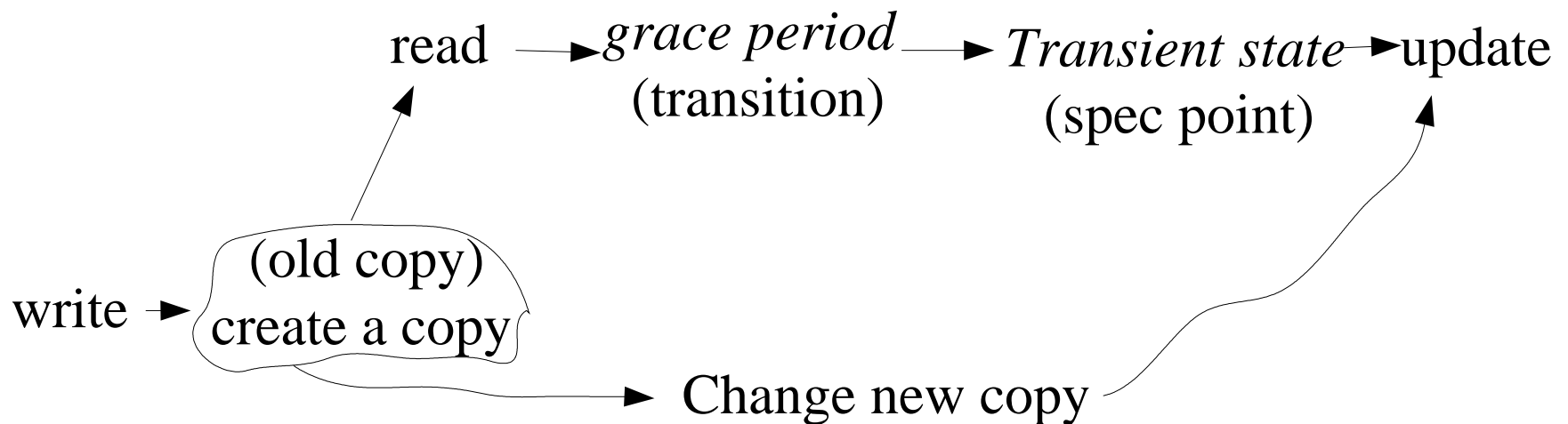
```
preempt_disable();
int cpu = get_cpu();
// manipulating per_cpu(xxx, cpu);
// xxx is per_cpu data, such as runqueues.
preempt_enable
```

Thread 1, running on CPU 0

# 13 Read-Copy Updates

<linux/rcupdate.h>

- Best for read-mostly linked list(struct list\_head).
- another Reader-Writer lock, but more complex and advantaged.
- Reader will not block.



# Big Kernel Lock: history

- Linux 2.0 - BKL about 1996 - SMP
- BSD/OS 4.x:
- FreeBSD 4.x: XXX – Giant (2000 -)
  - goal : fine-grained locking
- 
- Dragonfly BSD: forked from FreeBSD 4.x
  - goal: lockless mem allocator and scheduling system

# FUTEX

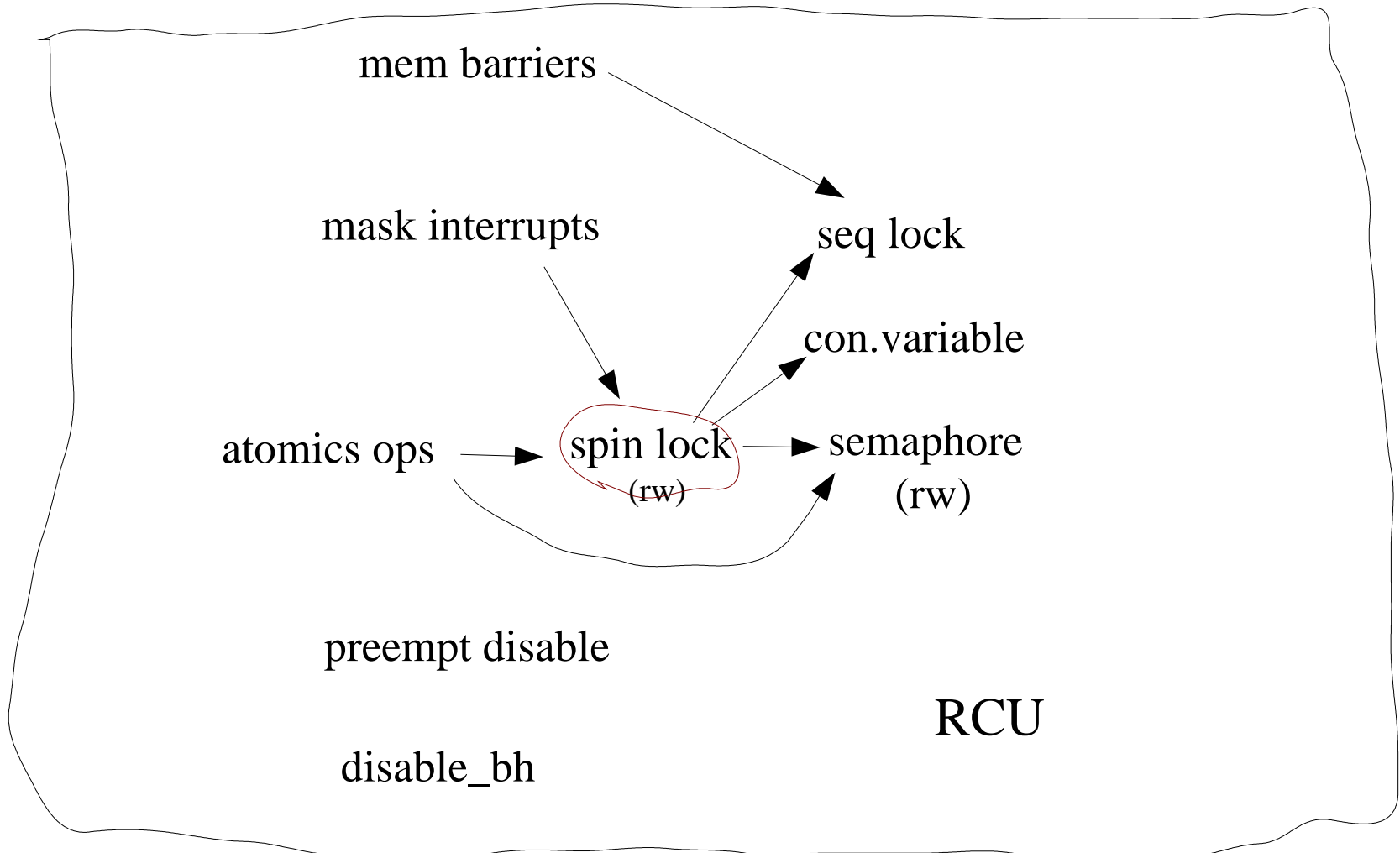
- Fast User Space Mutex
  - It's for user-space threads synchronization.
  - It's not a locking mechanism for kernel.
  - It is implemented in kernel.



# Relation of different locks implementations

*simple*

*complex*



# Locking in Linux kernel

- Kernel Control Paths
- Locking in Linux
- Locking and Coding
- Conclusions

# Locking and Coding

- Is the **data** shared? Can other threads(contexts) access it?
- Is the **data** per-CPU's? Can other CPUs access it? \*
- Is the **data** shared between threads context and interrupt context? Is it shared between two different interrupt handlers? ...
- If a context is preempted while accessing this **data**, can the newly scheduled context access the same **data**?
- Can the current context sleep on anything while accessing the **data**? If it does, what state does that leave the shared **data** in?
- Does the **data** has special application? Keep in mind. \*
  
- Now LET'S Continue CODING!

# Locking and Coding

- Interrupt safe
- Preempt safe
- SMP safe
  - (preempt safe  $\cong$  SMP safe)

# Locking between various KCPs

- Exceptions..
- Interrupts..
- Bottom Halves..
- Kernel threads..
- System calls by user space threads..

# 1 between exception contexts

(UP:sleeping locks, SMP:+0)

- 1. **Exception could not be caused in kernel.** If any kernel codes trigger an exception, this is a bug.
- 2. BUT page\_fault and float-point registers exceptions
- 3. Exceptions could be caused by user-space codes.
- 4. According to 1<sup>st</sup> item, **exception contexts could not trigger another exceptions, including page\_fault and float-point registers exceptions.** But exception contexts could be preempted by interrupts, and after interrupts return ,the preempted exceptions continue on same CPU.
- 5 so we could conclude that **sleeping locking are enough.**

# 2 between interrupts contexts

(UP:mask local interrupts, SMP:+spinlock)

- Interrupts contexts have no kernel stack. It could not sleep. **Do not use sleeping locks.**
- Same interrupt context runs serially on same CPU because `irq_desc->handler.ack()` in `do_IRQ()` masks the irq line. On UP, This situation is simple.
- Same or different interrupts could be triggered on different CPUs, so SMP requires spinlock to prevent race condition.

# 3 between Bottom Halves

(UP:null, SMP:+spinlock)

- **Do not use old BH mechanism**, it has poor performance and has been removed in 2.6.
- Softirqs could not be preempted, except by interrupts. so on UP, there is no race conditions.
- **Bottom Halves could not sleep like interrupts** for the same reasons.
- Same or different softirqs could run on different CPUs.
- Tasklets are based on softirqs. Only different tasklets could run on different CPUs.
- From above descriptions, we can conclude that on SMP softirqs and different tasklets should be protected with spinlocks, same tasklet could be used locklessly.



# 4 between exceptions and interrupts/bh

(UP: mask interrupts, SMP:+spinlocks)

- Interrupts could not be preempted by exceptions, if this situation happens, this is a bug!
- So exceptions could disable interrupts to avoid preemption by interrupts.
- 
- bh is like interrupts, it is executed in interrupt contexts.
- However, exceptions could use `local_bh_disable()` to disable bottom halves.

# 5 between BottomHalves and interrupts

(UP: mask interrupts, SMP: spinlock)

- Bottom halves could use disabling interrupts to avoid concurrency.
- for SMP, spinlock is necessary and enough.

# 6 between kernel threads and interrupts/bh

(UP: mask interrupts, SMP:+spinlock)

- Interrupts could preempt threads. so disable interrupts to protect data used by threads.
- Because interrupts could not be preempted, so we use spinlock.

# 7 between threads

(spinlock or sleeping lock)

- NOTE: in 2.6, spinlock automatically disabling preemptions.
- what to use: spinlock or sleeping lock?

*low overhead locking,*

*spinlock*

*short lock hold time,*

*spinlock*

*long lock hold time,*

*semaphore*

*sleep while holding lock,*

*semaphore*

# 8 between system calls

(spinning lock or sleeping lock)

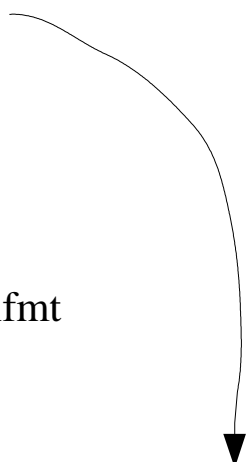
- This is same as between kernel threads.

# Locking used between various KCPs

	UP	SMP+
exceptions -----	sleepinglock	null
interrupts -----	mask interrupts	spinlock
bottom halves -----	null	spinlock or null
exceptions and interrupts/bh -----	mask interrupts	spinlock
bottom halves and interrupts -----	mask interrupts	spinlock
kernel threads and interrupts/bh ----	mask interrupts	spinlock
kernel threads -----	sleeping or spin lock	
system calls -----	sleeping or spin lock	

# Kernel Configuration Tree and Debug

<make menuconfig>

- arch/xxx/Kconfig (mainmenu, <menu,endmenu>\*)
    - arch/xxx/Kconfig.debug
      - lib/Kconfig.debug
    - init/Kconfig
    - fs/Kconfig.binfmt
    - fs/Kconfig
    - drivers/Kconfig.binfmt
    - lib/Kconfig
    - ...
  - CONFIG\_DEBUG\_KERNEL
    - CONFIG\_DEBUG\_SPINLOCK, CONFIG\_SPINLOCK\_SLEEP
    - CONFIG\_DEBUG\_STACKOVERFLOW, CONFIG\_4KSTACKS
    - CONFIG\_KDB(patch)
    - ...
- 

# Locking in Linux kernel

- Kernel Control Paths
- Locking in Linux
- Locking and Coding
- **Conclusions**



# Conclusions

- Locking or synchronization is a complex problem, especially for **large and/or complex system.**
- The problem caused by Locking in kernel is **not entirely predictive.**

# Locking: What is the problem?

- Implementing the actual locking in the code to protect shared data is not hard.
- The tricky part is **identifying the actual shared data and corresponding critical sections.**

# Locking: What is the problem?

- Deadlocks
- Priority Inversion
- Locking latency
- Locking: Coarse or fine-grained.
  - Scalability VS. Overheads(performance).
  - Not only Linux has the dilemma.
  - Let's keep close eyes at DragonflyBSD's progress

# References

- Linux kernel source tree by Linus Torvalds and various patches by hackers.
- Linux Kernel Development. by Robert Love.
- Understanding the Linux Kernel. by Daniel Bovet etc.
- [www.freebsd.org/smp](http://www.freebsd.org/smp)
- [www.dragonflybsd.org](http://www.dragonflybsd.org)
- .../kernel/Documents/\*, google, gcc document...
- Pentium 4 software development document(3 volumes).

# Thanks

- USTC BBS embedded board master: *dj*
- All the organizers and/or friends of the *USTC 2005 developer workshop of embedded system.*
- USTC Linux Users Group.

Happy Life, Happy Hacking.

THANKS