Concurrency

Concurrency = things happening at the same time

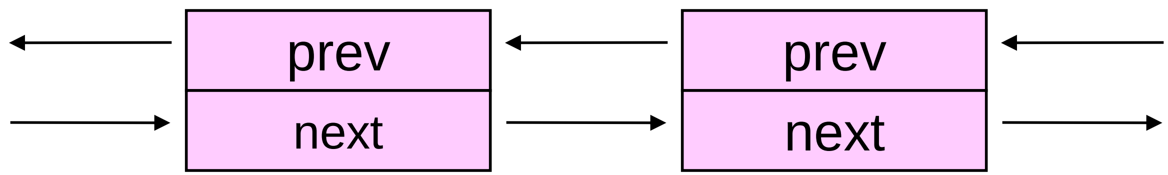
Sources of concurrency in Linux
Multiple processors
Hardware interrupts
Software interrupts
Kernel timers
Workqueues
Kernel preemption

Failure to manage concurrency leads to bugs
Difficult to track down
Devastating when they happen

One of the biggest issues for new kernel programmers

Example: linked list

A kernel linked list looks like this:



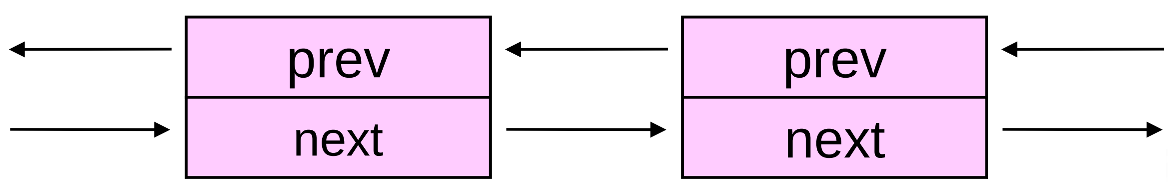
The relevant declaration is this:

```
struct list_head {
    struct list_head *next, *prev;
};
```

Example continued

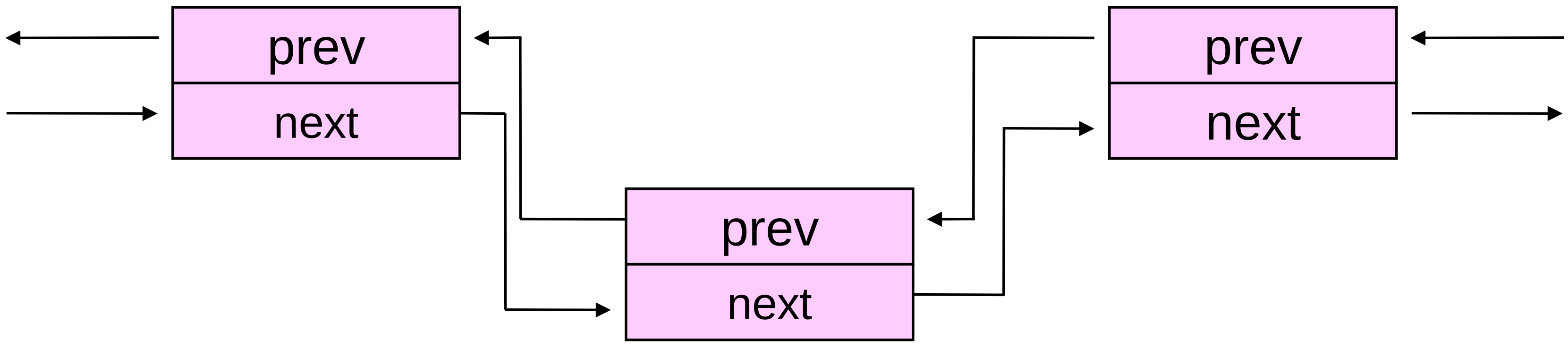
The code to add an element to the list is:

```
static inline void __list_add(struct list_head *new,
              struct list_head *prev,
              struct list_head *next)
{

next->prev = new;
new->next = next;
new->prev = prev;
prev->next = new;

}
```



Example continued

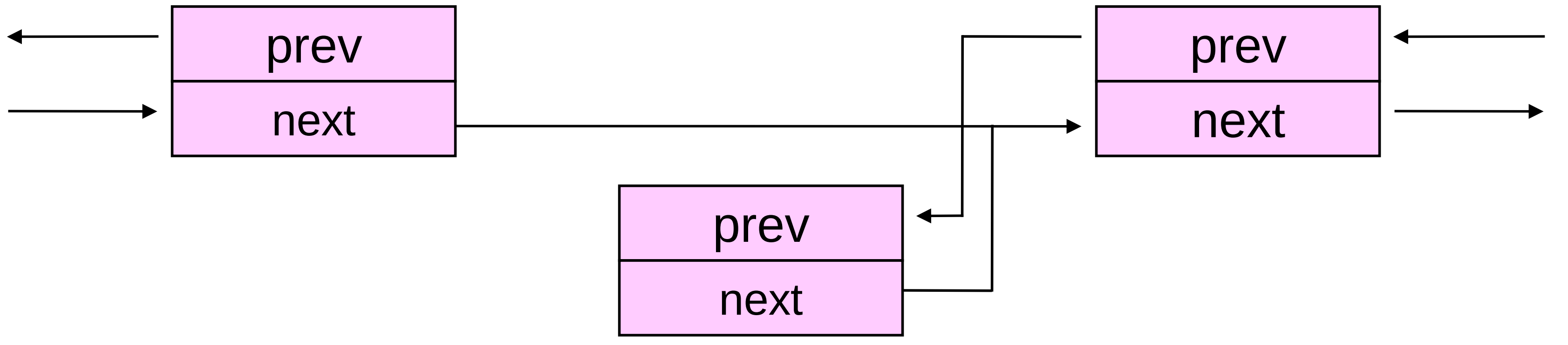
It yields results like:



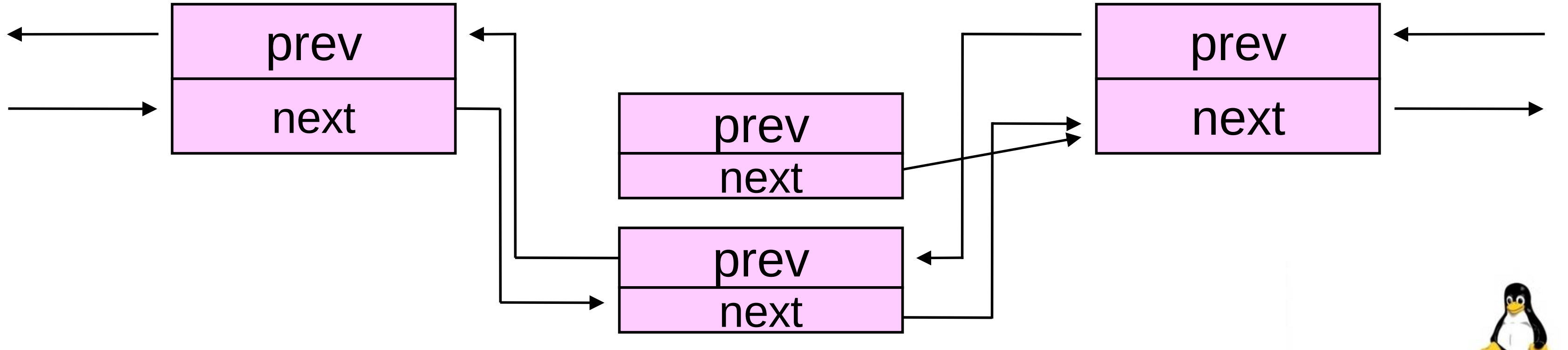
What if two processors add an element simultaneously?

A concurrency disaster

The first processor gets started

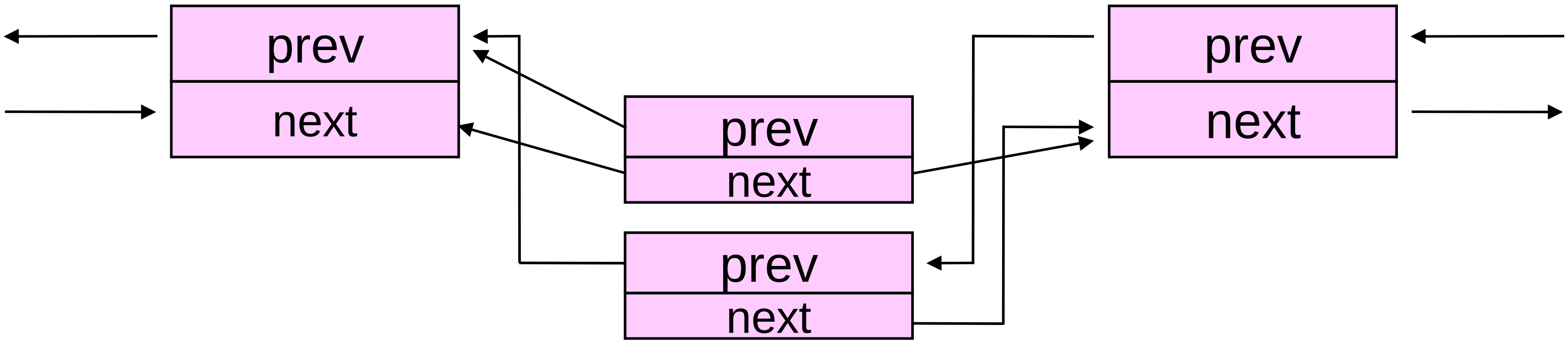


The second one slips in



The disaster is complete

Finally the first processor finishes its job



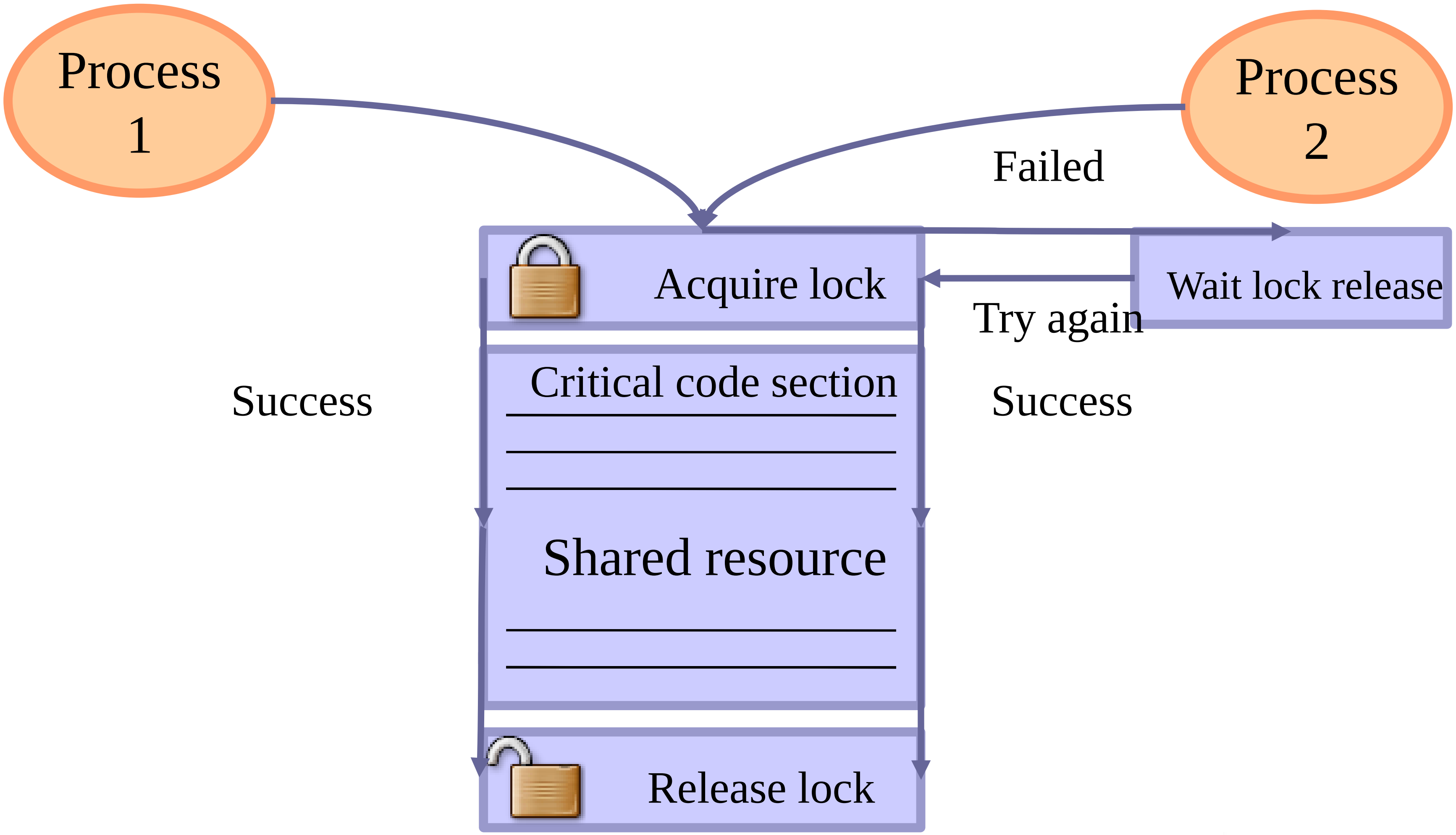
The outcome is a corrupted data structure

Results:
Memory leaks
Weird behavior
Kernel crashes
Security problems

Dealing with concurrency

- Concurrency must be managed
  - Or the kernel will not run reliably
- Mutual exclusion ("Critical sections")
  - Only allow one thread in at once
  - The core concurrency management technique
- Kernel programmers must protect:
  - Shared data structures
  - Hardware resources
- Linux mutual exclusion methods:
  - Semaphores
  - Spinlocks
  - Completions
  - Seqlocks
  - Read-copy-update

Concurrency Protection With Locks



Semaphores

Semaphores are a standard mutual exclusion primitive

Semaphores can sleep
Can't be used in atomic context

Optimized for the non-contention case

Relatively fair
Waiting threads are queued

Reader/writer variant exists

Semaphore initialization

You can declare and initialize a semaphore with:

```
#include <asm/semaphore.h>

struct semaphore sem;
void sema_init(struct semaphore *sem, int value);
```

The usual technique, however, is:

```
DECLARE_MUTEX(name);
DECLARE_MUTEX_LOCKED(name);
```

Or, at run time:

```
void init_MUTEX(struct semaphore *sem);
void init_MUTEX_LOCKED(struct semaphore *sem);
```

Semaphore acquisition

There are three functions to acquire a semaphore:

void down(struct semaphore *sem);

- Blocks (uninterruptible) until sem is available

int down_interruptible(struct semaphore *sem);

- Blocks until sem is available
- Can be interrupted by signals

Non-zero return indicates signal

Semaphore *not* acquired

- This form should usually be used

int down_trylock(struct semaphore *sem);

- Will acquire semaphore if available

But will not sleep

Return value of zero indicates success

Releasing a semaphore

A semaphore is released with:

```
void up(struct semaphore *sem);
```

Notes:

Only a legitimate holder can call up()
It always succeeds

Semaphore usage

A typical critical section looks like:

```
if (down_interruptible(&the_sema))
    return -ERESTARTSYS;
/* Critical section work here */
up(&the_sema);
```

Reader/writer semaphores

A reader/writer semaphore (rwsem) may be a better choice

You have lots of readers
- And they can all access the data simultaneously

...and relatively few writers
- Who need exclusive access

An rwsem allows this mode of access

Readers can share the data

One writer will block all readers

Can be relatively expensive

The rwsem API in brief

The rwsem API looks like:

```
#include <linux/rwsem.h>

void init_rwsem(struct rw_semaphore *sem);

void down_read(struct rw_semaphore *sem);
int down_read_trylock(struct rw_semaphore *sem);
    /* non-zero = success! */
void up_read(struct rw_semaphore *sem);

void down_write(struct rw_semaphore *sem);
int down_write_trylock(struct rw_semaphore *sem);
void up_write(struct rw_semaphore *sem);
```

There is no interruptible version

Completions

It can be tempting to use semaphores as event flags

```
/* Wait for the data to arrive */
down(&data_semaphore);

/* (somewhere else) */
/* Tell them the data is here */
up(&data_semaphore);
```

- This is not the best approach
  - Semaphores are optimized for the no-wait case
- This usage always waits
  - Race conditions are possible

Use completions instead

The completion API

Completions are simple to use

```
#include <linux/completion.h>

struct completion *my_compl;
Init_completion(&my_compl);
/* ... or ... */
DECLARE_COMPLETION(my_compl);

void wait_for_completion(&my_compl);

void complete(&my_compl); /* Only wakes one */
void complete_all(&my_compl);
```

Linux Mutexes

- The main locking primitive since Linux 2.6.16.
  Better than counting semaphores when binary ones are enough.

  Mutex definition:

  ```
  #include <linux/mutex.h>
  ```
- Initializing a mutex statically:
  `DEFINE_MUTEX(name);`
- Or initializing a mutex dynamically:
  `void mutex_init(struct mutex *lock);`

Locking & Unlocking Mutexes

- void mutex_lock (struct mutex *lock);
  Tries to lock the mutex, sleeps otherwise.
  Caution: can't be interrupted, resulting in processes you cannot kill!
- int mutex_lock_killable (struct mutex *lock);
  Same, but can be interrupted by a fatal (SIGKILL) signal. If interrupted, returns a non zero value and doesn't hold the lock. Test the return value!!!
- int mutex_lock_interruptible (struct mutex *lock);
  Same, but can be interrupted by any signal.
- int mutex_trylock (struct mutex *lock);
  Never waits. Returns a non zero value if the mutex is not available.
- int mutex_is_locked(struct mutex *lock);
  Just tells whether the mutex is locked or not.
- void mutex_unlock (struct mutex *lock);
  Releases the lock. Do it as soon as you leave the critical section.

Spinlocks

The core Linux mutual exclusion primitive

A simple, shared integer variable

To lock a spinlock:
    Decrement it by one
    If resulting value is zero
it's yours
     else
increment value
go try again (spin)

Some implications
Spinlocks are very fast
Contention is very expensive
Sleeping is out of the question

A couple of spinlock facts

- Uniprocessor kernels optimize out spinlocks
  - Locking is a no-op

- Holding a spinlock disables preemption
  - Required for deadlock avoidance

- Long lock hold times can create latency problems
  - Even for unrelated code

Simple spinlock operations

Declare/initialize with:

```
#include <linux/spinlock.h>

spinlock_t my_lock = SPIN_LOCK_UNLOCKED;
/* ... or (preferred) ... */
spinlock_t my_lock;
spin_lock_init(&my_lock);
```

Acquisition functions:

```
void spin_lock(spinlock_t *lock);
void spin_unlock(spinlock_t *lock);
```

Notes

Spinlock ops are not interruptible

Deadlocks are easy to create

Rules for spinlocks

Some rules apply to spinlocks

Critical sections must be short

- Or you will create problems for others

They must be atomic

- No sleeping!
- No user space access

Multiple locks may be held

- Be careful about order

spin_lock() calls do not nest

- Double-locking will cause deadlocks

Spinlocks and interrupts

Imagine this scenario

- Your driver holds a spinlock
- The device interrupts
- Your driver's interrupt handler runs

...on the same processor

...and it tries to take the lock

- The processor is now hung

Spinlocks disable kernel preemption

- Blocks some deadlock scenarios

But interrupts are not disabled

- ...by default

If interrupts are a possibility

- You must disable them

Interrupts and spinlocks

The full locking API is:

```
void spin_lock(spinlock_t *lock);
    The normal locking function

void spin_lock_irqsave(spinlock_t *lock,
              unsigned long flags);
    Disables interrupts, saves previous state
    Note flags is passed "by value"

void spin_lock_irq(spinlock_t *lock);
    Disables interrupts without saving state
    Use when you know interrupts are enabled

void spin_lock_bh(spinlock_t *lock);
    Only disables software interrupts
```

Unlocking API

The full unlocking API is:

```
void spin_unlock(spinlock_t *lock);
void spin_unlock_irqrestore(spinlock_t *lock,
                unsigned long flags);
void spin_unlock_irq(spinlock_t *lock);
void spin_unlock_bh(spinlock_t *lock);
```

Use the one which matches the locking function

There is also a reader/writer spinlock type
- rwlock_t
- See LDD3 for details

Which to use?

Use semaphores when:
- The critical section is long
- Sleeping is a possibility
- Delays can be tolerated

Use spinlocks when:
- The critical section is short
- Code is running in atomic context

Interrupt handlers
Atomic kmaps held
Preemption is disabled
Tasklets
- Latency must be minimized

seqlocks

Seqlocks are a specialized lock avoidance technique
- Protect small amounts of data
- No pointers
- No side effects
- Writers get priority

Used to protect:
- System time updates
- dentry lookups (w/respect to renames)

Initialization:

```
#include <linux/seqlock.h>

seqlock_t lock = SEQLOCK_UNLOCKED
```

Seqlocks - read side

- The read interface looks like:
  - unsigned int seq;

    do {
```
seq = read_seqbegin(&lock);
/* Do something with it */
```
    } while read_seqretry(&lock, seq);

- This is a retry-based mechanism
  - Retry on race with writer

- Other notes:
  - Reader should copy data of interest
  - Preemption disabled within critical section

Seqlocks - write side

The write interface is:

```
    write_seqlock(&lock);
/* make changes here */
    write_sequnlock(&lock);
```

- Implemented with a spinlock
  - Usual constraints apply
- Writers get immediate access
  - No need to wait for readers
  - Will block other writers, though

Read-copy-update

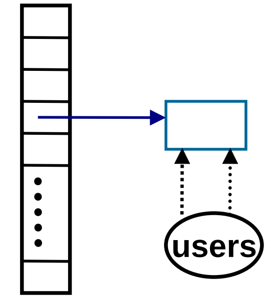
- RCU is an advanced locking avoidance mechanism
  - Patented by IBM
  - Available for GPL code only

- Used to protect complex data structures
  - Pointer-oriented
  - Many readers
  - Frequently updated

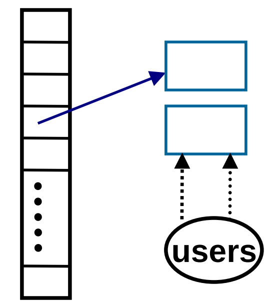
- Example: network routing table
  - Routes change frequently
  - Others may be using a given route
  - Using old or new route is legit
- But freeing old too early is not

An RCU example

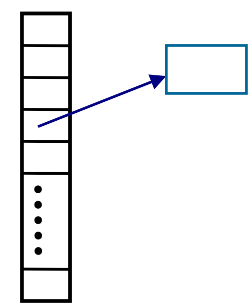
- Step 1: The initial data structure
  - Contains a pointer to something interesting
  - Read-only users have free access
- No locks required



- Step 2: Something changes
  - The new structure is added
    - Pointer changed immediately
  - Prior users still use old copy



- Step 3: No more users of old copy
  - It can be recycled
  - Data structure update is complete



RCU - how it works

- RCU sets some rules
  - Data structures are accessed via pointers
  - All accesses are atomic
- When the structure changes
  - Set a pointer to a new, updated object
  - Wait for all processors to schedule
  - Invoke cleanup callbacks
- This works because
  - Access is atomic
  - A processor which schedules can hold no references
    - If it follows the rules
  - When all have scheduled, no references remain

RCU - read side

The basic RCU read-side interface is:

```
#include <linux/rcupdate.h>

struct my_stuff *stuff;

rcu_read_lock();
stuff = find_the_stuff(...);
do_something_with(stuff);
/* Everything here is atomic */
rcu_read_unlock();
```

Some problems:

- Preemption is disabled
- No indication of what is being protected

RCU - write side

1) Copy the data structure to be updated
   Leaving the current one in place

2) Make changes to the copy

3) Adjust pointer to new, updated copy

4) Call:
```
void call_rcu(struct rcu_head *head,
    void (*callback)(void *arg),
    void *arg);
```

5) After all processors have scheduled
   callback() will be called
   It can free the old structure

Final thoughts on locking

- Design locking in from the beginning
  - Code will not be correct without
  - Can be hard to retrofit
- Know what you are protecting
  - Any shared data structure
  - Hardware resources
  - Have clear access rules
- Avoid fine-grained locking
  - Until you have no other choice