Mutual exclusion in Linux

(or: how to avoid big messes in the kernel)

Concurrency

Sources of concurrency

Multiple processors
Hardware interrupts
Software interrupts
Kernel timers
Tasklets
Workqueues
Preemption
...

Concurrency is good

The only way to use SMP systems

Use any system to its fullest potential

Concurrency is a problem

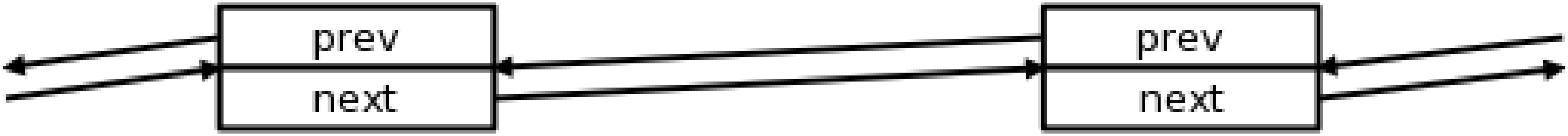
Uncontrolled concurrency leads to disaster

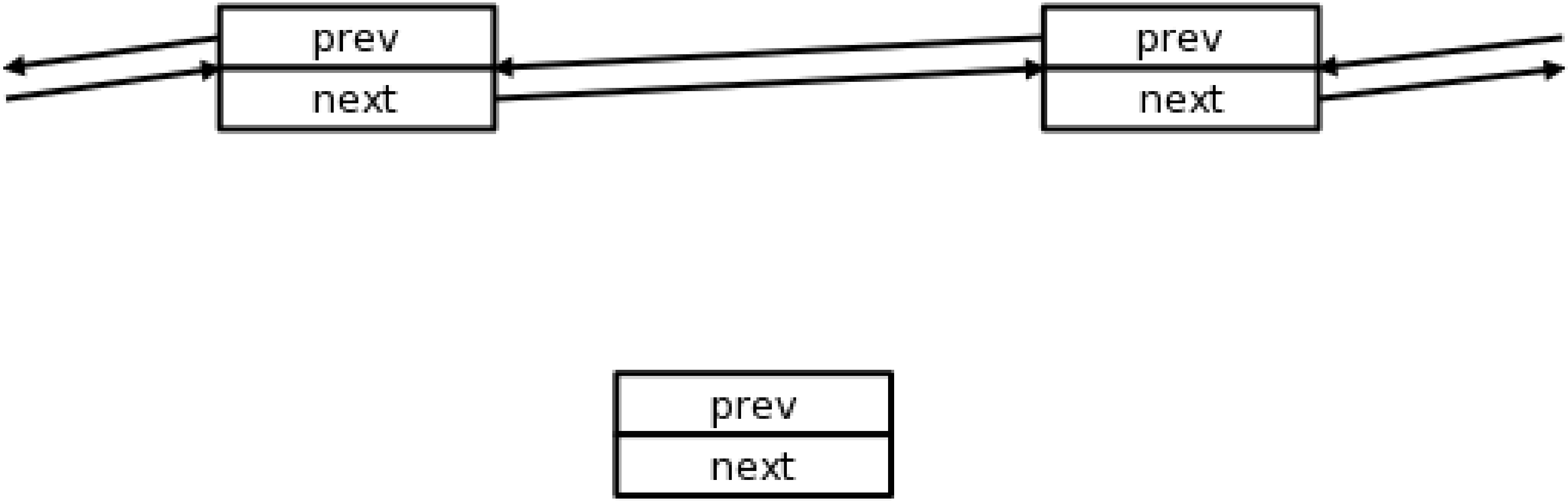
photo: Joey Parsons

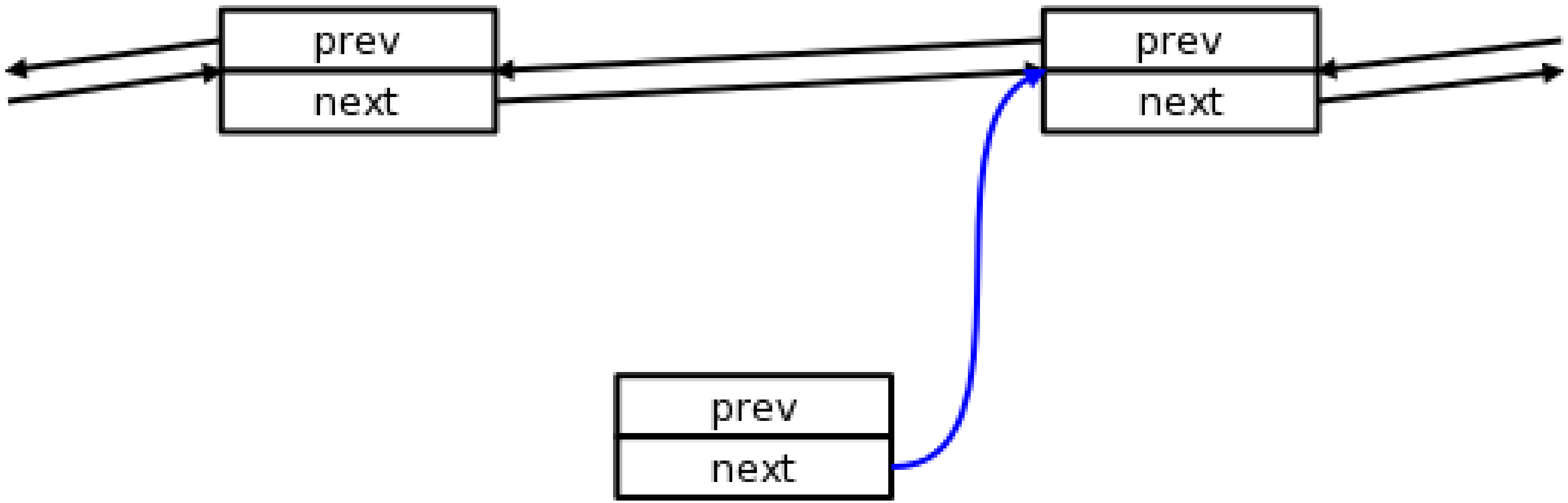
A simple example

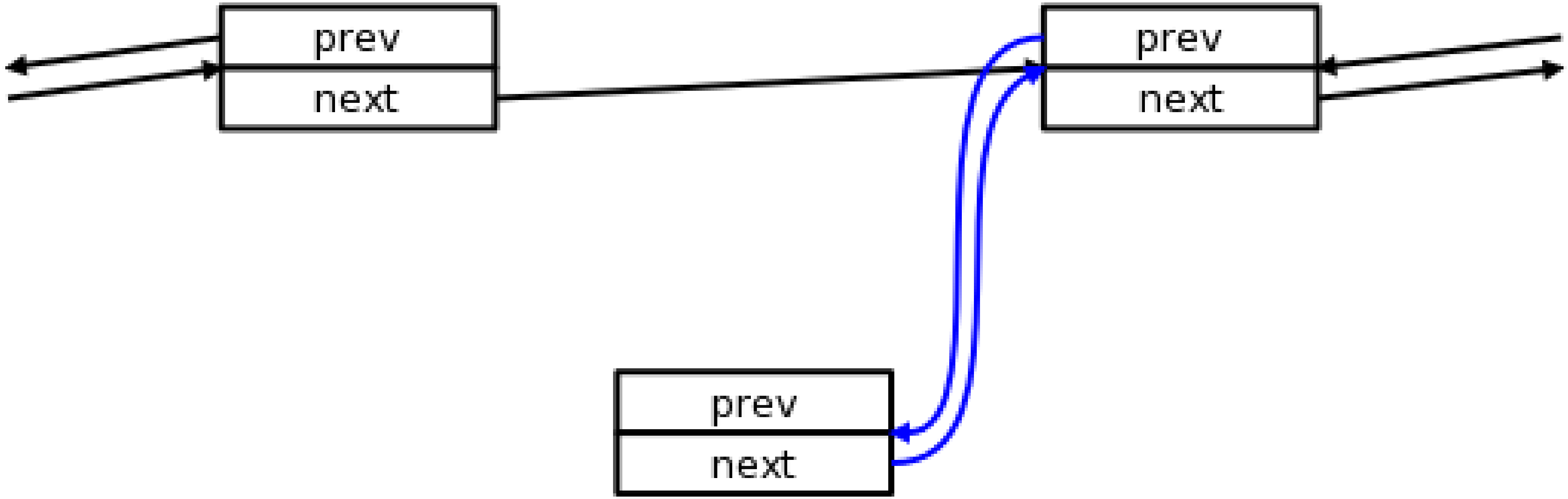
The Linux linked list type

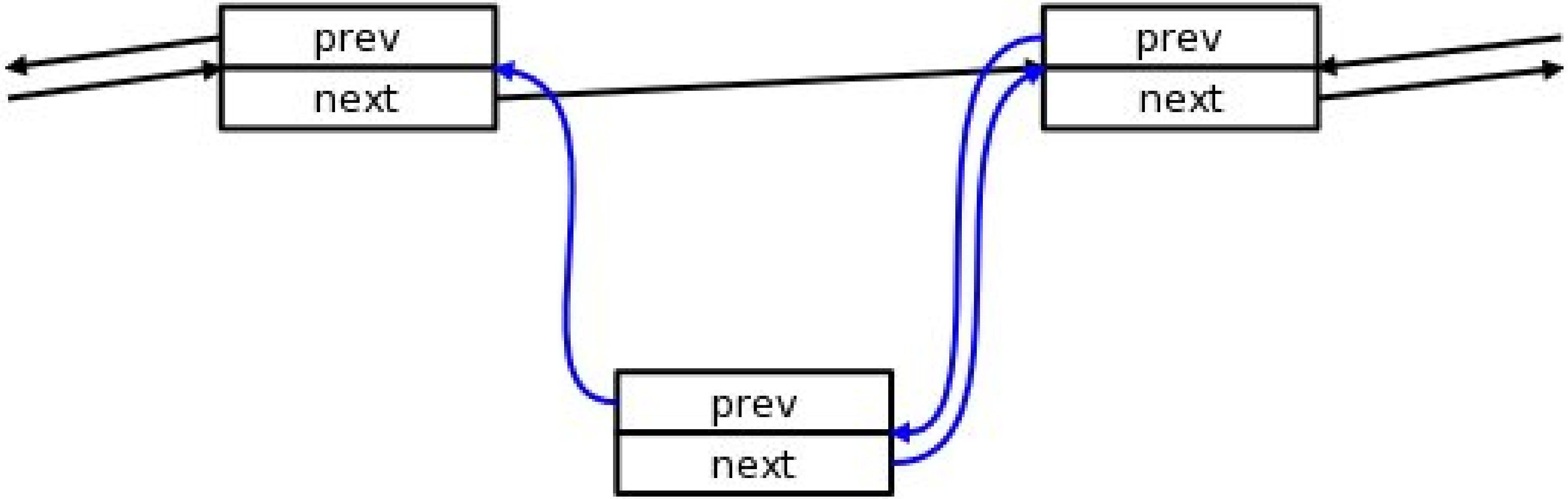
```
struct list_head {
    struct list_head *next, *prev;
};

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 = next;
}
```

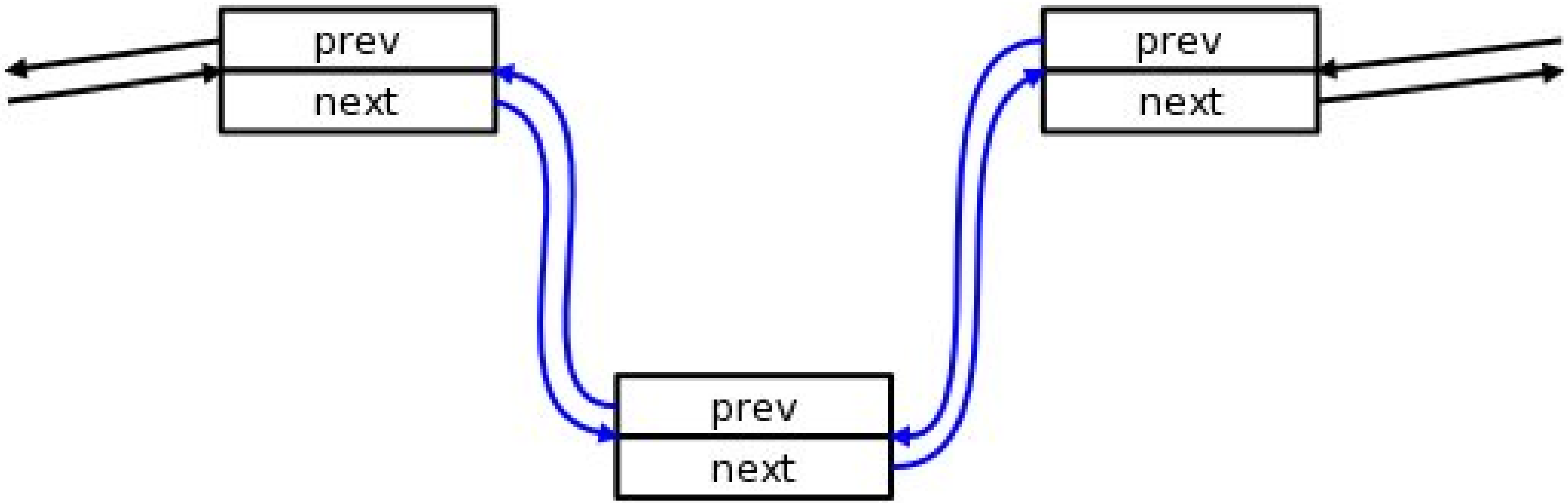

prev
next
prev
next

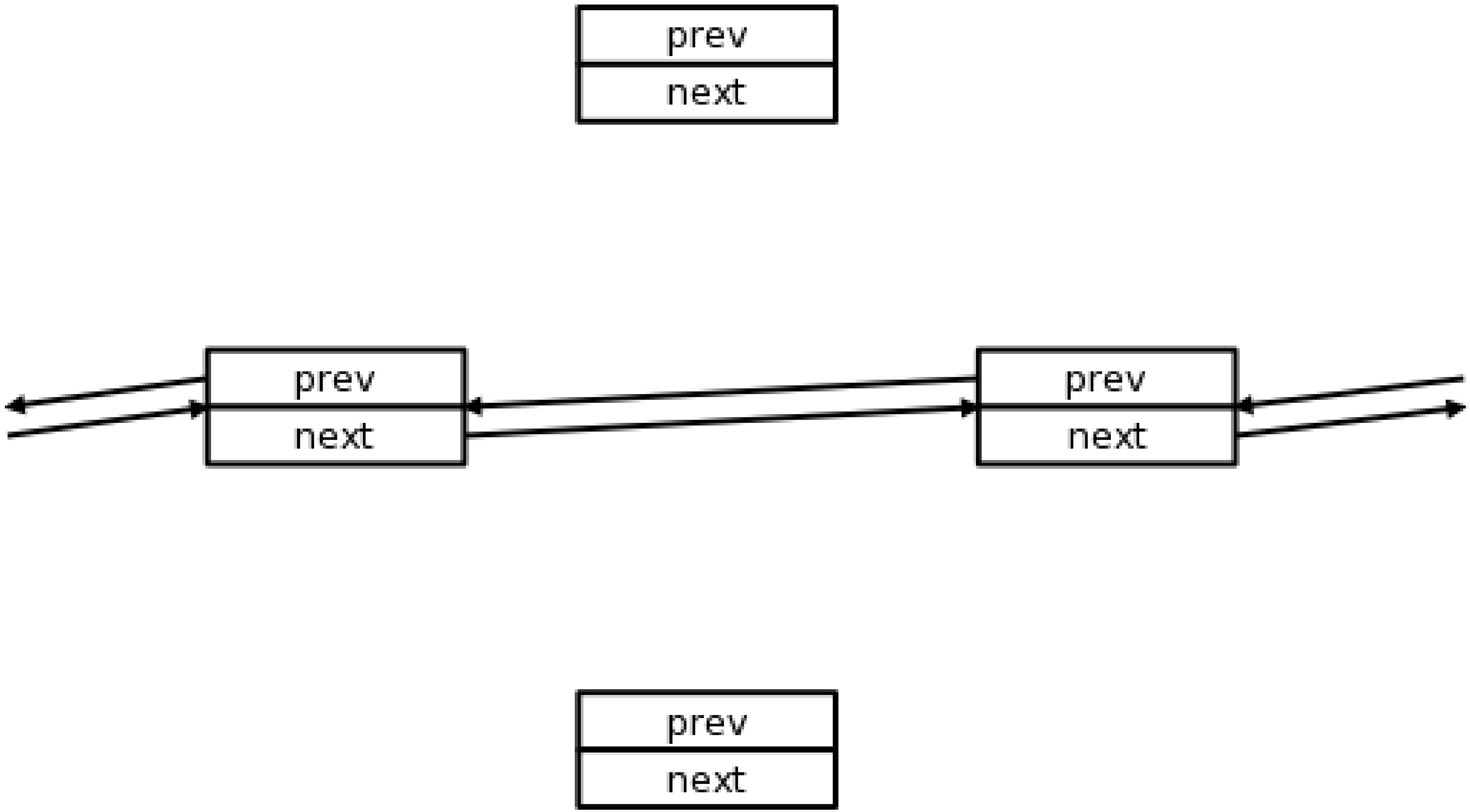

prev
next
prev
next
prev
next

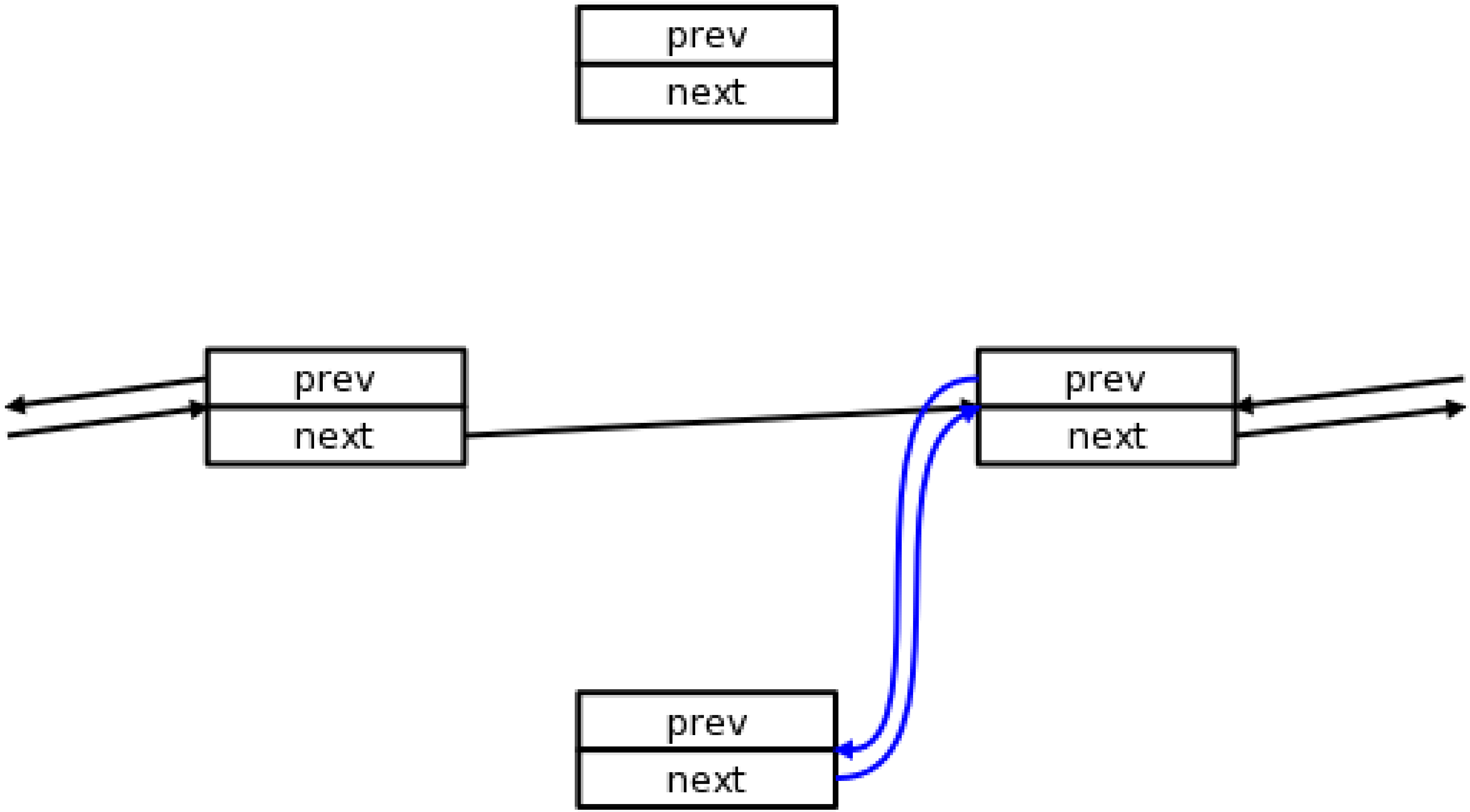

prev
next
prev
next
prev
next

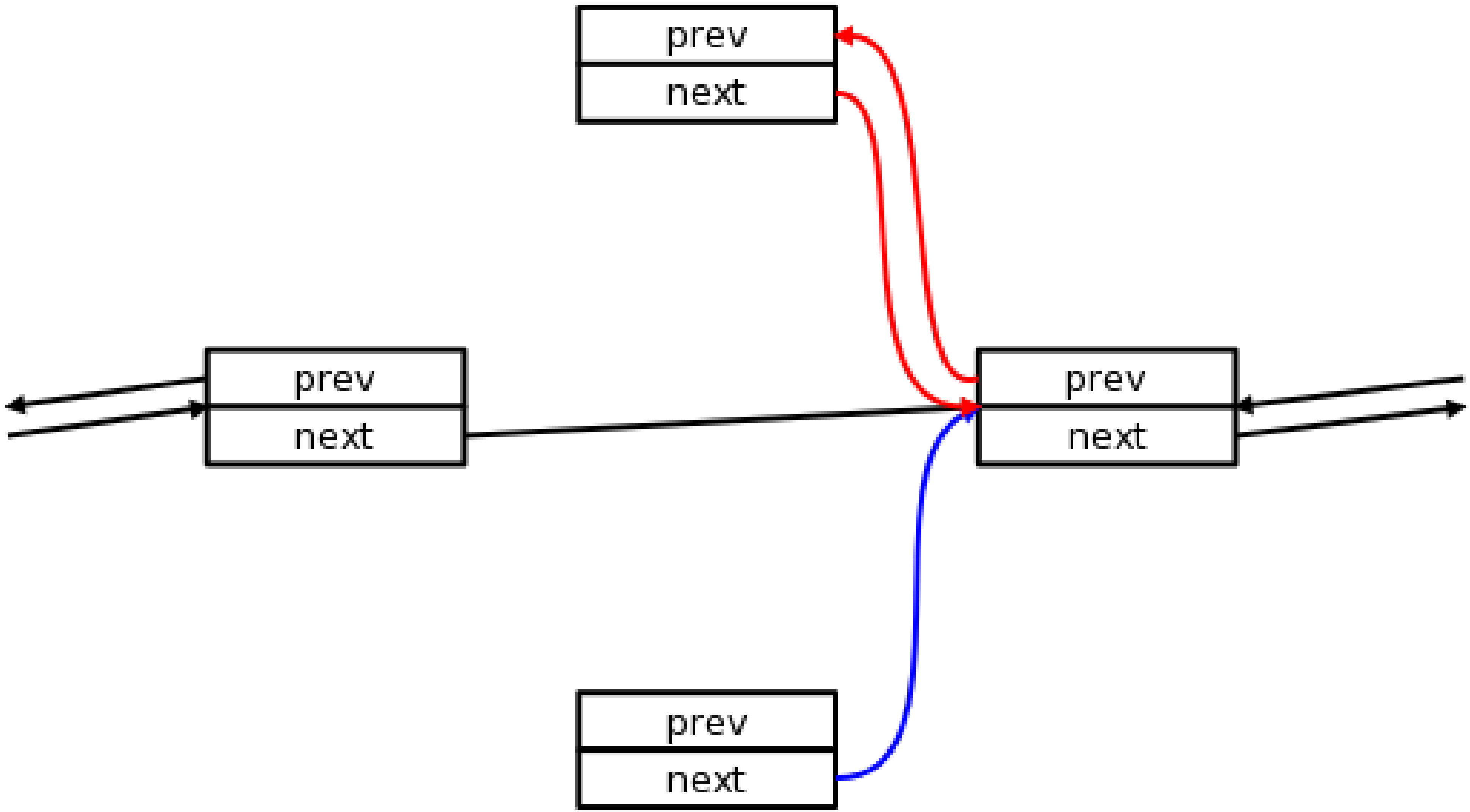

prev
next
prev
next
prev
next

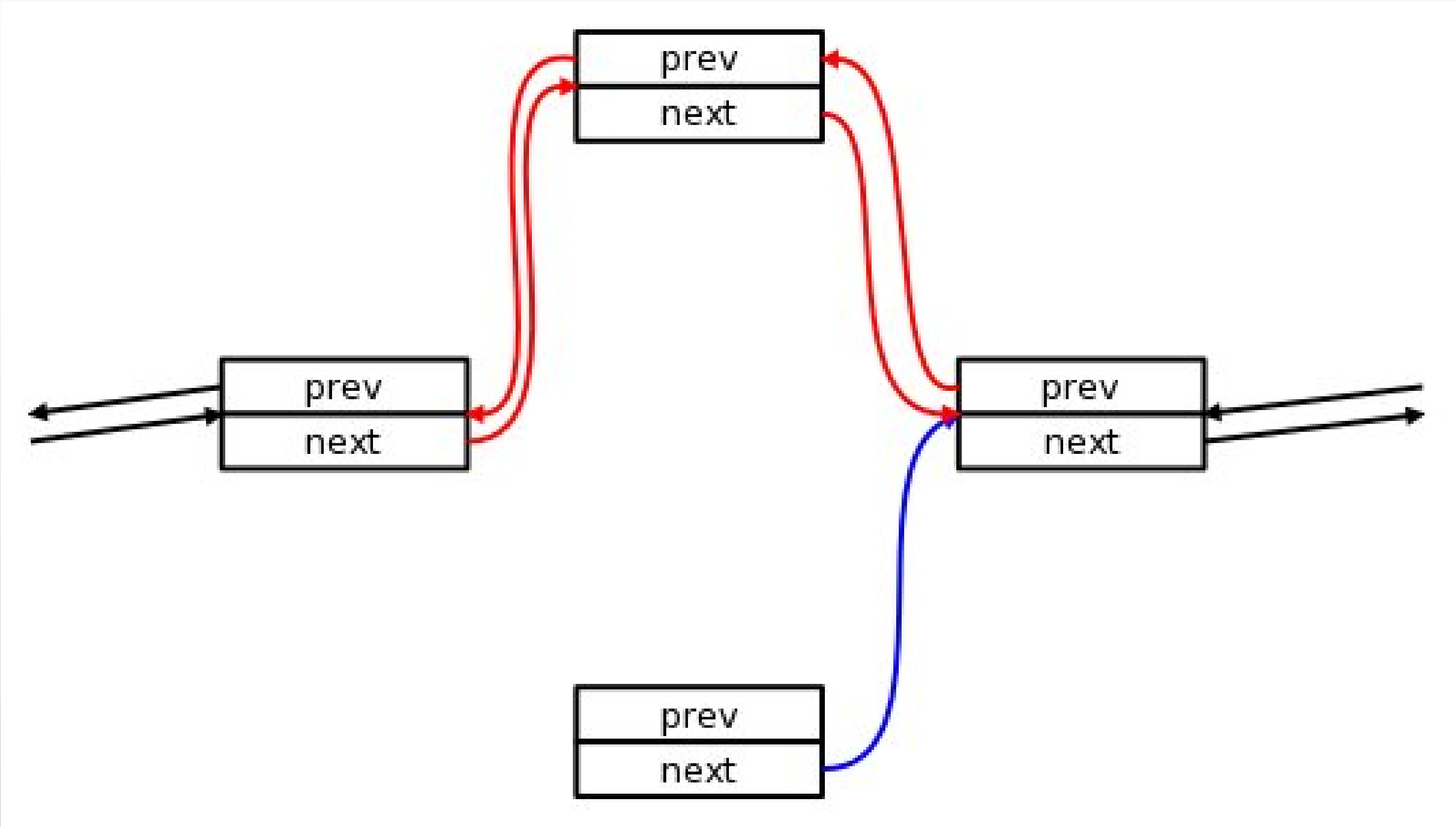

prev
next
prev
next
prev
next

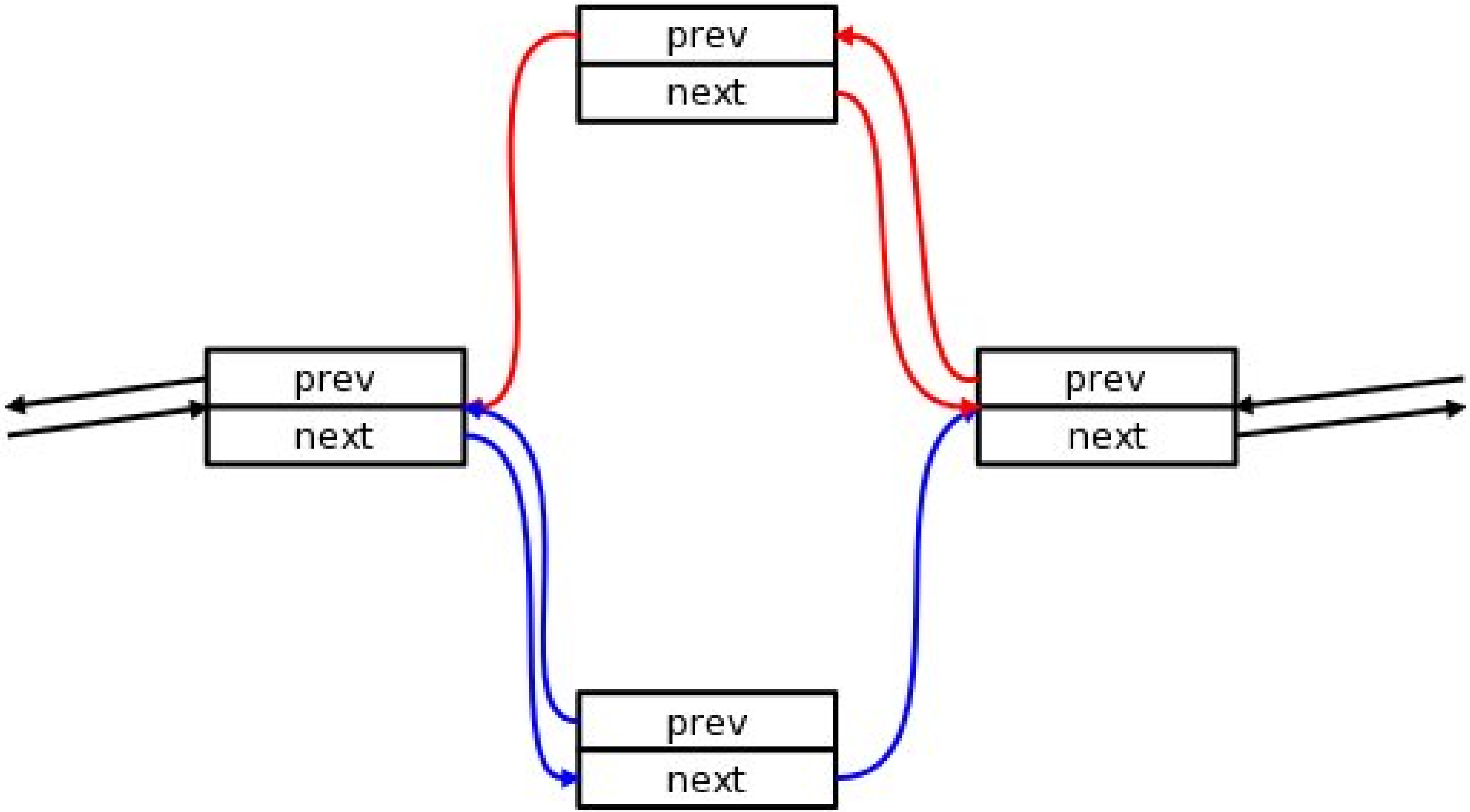

prev
next
prev
next
prev
next


prev
next
prev
next
prev
next
prev
next


prev
next
prev
next
prev
next
prev
next


prev
next
prev
next
prev
next
prev
next


prev
next
prev
next
prev
next
prev
next


prev
next
prev
next
prev
next
prev
next

Race conditions

...when concurrency goes bad

- Memory leaks
- Kernel crashes
- Security holes
- Data corruption
- ...

These bugs are

- Hard to reproduce
- Hard to find
- Easy to create

Avoiding race conditions

One must limit concurrency!

In particular, access to global resources

- Data structures
- Hardware resources

...must be controlled

Kernel resources

Concurrency control mechanisms

- spinlocks
- mutexes
- completions

Concurrency avoidance mechanisms

- atomic variables
- per-CPU variables
- read-copy-update

Spinlocks

The core kernel mutual exclusion primitive

One processor can "own" a lock

- Any others will "spin" waiting for it

Thus:

- Spinlocks are fast to acquire and release
- Spinlock contention is very expensive
- Code holding spinlocks cannot sleep

Atomic context

Threads holding spinlocks cannot sleep

No:

- kmalloc(GFP_KERNEL)
- copy_*_user()
- schedule()

Preemption is disabled

- Long hold times will create latencies

Also...

Never return to user space with a spinlock held

Lock declaration

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

spinlock_t my_lock;

spin_lock_init(&my_lock);
```

Basic locking

To acquire a spinlock:

```
spin_lock(&my_lock);
```

To give it back:

```
spin_unlock(&my_lock);
```

Spinlocks and interrupts

Consider this scenario:

- Device driver acquires a spinlock
- The device interrupts
- Driver's interrupt handler is called
- Interrupt handler attempts to acquire the spinlock

....

The CPU is never heard from again

Interrupt-safe locking

```
/* Unconditionally disable interrupts */
spin_lock_irq(spinlock_t *lock);
spin_unlock_irq(spinlock_t *lock);

/* Save previous IRQ state */
spin_lock_irqsave(spinlock_t *lock,
                  unsigned long flags);
spin_unlock_irqrestore(spinlock_t *lock,
                       unsigned long flags);

/* Software interrupts only */
spin_lock_bh(spinlock_t *lock);
spin_unlock_bh(spinlock_t *lock);
```

Mutexes

Another low-level locking primitive

Differences from spinlocks:

- Slightly heavier-weight
- Mutex acquisition can sleep
- Code holding mutexes can sleep

Mutex basics

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

struct mutex *my_mutex;

mutex_init(&my_mutex);
```

Mutex locking

Ways to acquire a mutex:

```
void mutex_lock(struct mutex *m);
int mutex_lock_interruptible(struct mutex *m);
int mutex_lock_killable(struct mutex *m);
```

Giving it back:

```
mutex_unlock(struct mutex *m);
```

Mutex rules

Mutexes can only be locked once

No mutex acquisition in atomic context

Holder must unlock the mutex

Code holding a mutex can be preempted

Adaptive spinning

If a mutex is contended

- Other acquirers will sleep

Except...

- If the owner is currently running
- Then acquirers will spin for a bit

The result

- Slightly unfair acquisition
- Better cache performance

Spinlock or mutex?

Use spinlocks when:

- Performance matters
- Critical sections are short
- Critical sections are accessed in atomic context

Use mutexes when:

- Critical sections must be able to sleep
- Hold times could be long

Mixing spinlocks and mutexes

It is possible to hold both types at once

Acquire the mutexes first!

Completions

Do not use mutexes to signal action completion

We have completions for that

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

void init_completion(struct completion *c);
```

Waiting for completion

```
void wait_for_completion(struct completion *c);
int wait_for_completion_interruptible(
                         struct completion *c);
int wait_for_completion_killable(
                         struct completion *c)
long wait_for_completion_timeout(
                         struct completion *c,
                         unsigned long timeout);
unsigned long
wait_for_completion_interruptible_timeout(
                         struct completion *c,
                         unsigned long timeout);
```

Signaling completion

Use one of:

```
void complete(struct completion *c);
void complete_all(struct completion *c);
```

To be avoided

Semaphores

- Unless you have a real counting semaphore need

rwlocks

Big kernel lock

- lock_kernel(); unlock_kernel();

Homebrew locking schemes

Questions on locking primitives?

Locking problems 1: contention

Contention for locks kills performance
- Especially when spinlocks are involved

One possible solution: finer-grained locks
- The kernel now has thousands of locks
- This has helped, but...

Locking problems 2: lock ordering

Multiple locks must always be taken in the same order

The alternative: ABBA deadlocks

Finer-grained locking makes the problem worse

ABBA?

ABBA deadlocks

Thread 1 takes lock A
...then attempts to take lock B

Thread 2 takes lock B
...then attempts to take lock A

Everybody waits for a very long time

The problem

What are the rules when you have thousands of locks?

One solution

Lockdep - the kernel lock prover

Configuration-time option

Will track all lock ordering
- IRQ states too

Complains on inconsistent usage

Significant performance impact

Locking problems 3: cache bouncing

Cacheline bouncing kills performance

Only on SMP systems

- ...but all systems are SMP now

Adding more locks may make the problem worse

A solution

Avoid locking altogether

Can greatly increase performance

- At the cost of trickier code

Atomic variables

Special variables which can be changed without locking

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

atomic_t my_atomic;
```

Atomic operations

```
void atomic_set(atomic_t *a, int value);
int atomic_read(atomic_t *a);

void atomic_add(int value, atomic_t *a);
void atomic_sub(int value, atomic_t *a);
int atomic_sub_and_test (int value, atomic_t *a);
void atomic_inc(atomic_t *a);
void atomic_dec(atomic_t *a);
int atomic_inc_and_test(atomic_t *a);
int atomic_dec_and_test(atomic_t *a);
...
```

Atomic ups and downs

Atomics can help avoid locking
- but only for simple operations

Their use can be expensive
- Cache bouncing
- Locked operations

Bit operations

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

void set_bit(int bit, unsigned long *v);
void clear_bit(int bit, unsigned long *v);
int test_bit(int bit, unsigned long *v);
int test_and_set_bit(int bit, unsigned long *v);

...
```

Per-CPU variables

An array of copies of a variable, one per CPU

Local access requires no locking
- Preemption must be disabled

Cross-CPU access may require locking

Creating per-CPU variables

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

/* At compile time */
DECLARE_PER_CPU(type, name);  /* in .h file */
DEFINE_PER_CPU(type, name);   /* in .c file */

/* At run time */
type var = alloc_percpu(type);
```

Local access to per-CPU variables

Simple case:

```
get_cpu_var(simple_counter)++;
put_cpu_var(simple_counter);
```

More complicated:

```
type &var = &get_cpu_var(percpuvar);
/* Do stuff; preemption is disabled */
put_cpu_var(percpuvar);
```

Cross-CPU access

Get a pointer with:

```
type *ptr = per_cpu_ptr(var, cpu_no);
```

Do you need some other locking?

Read-copy-update (RCU)

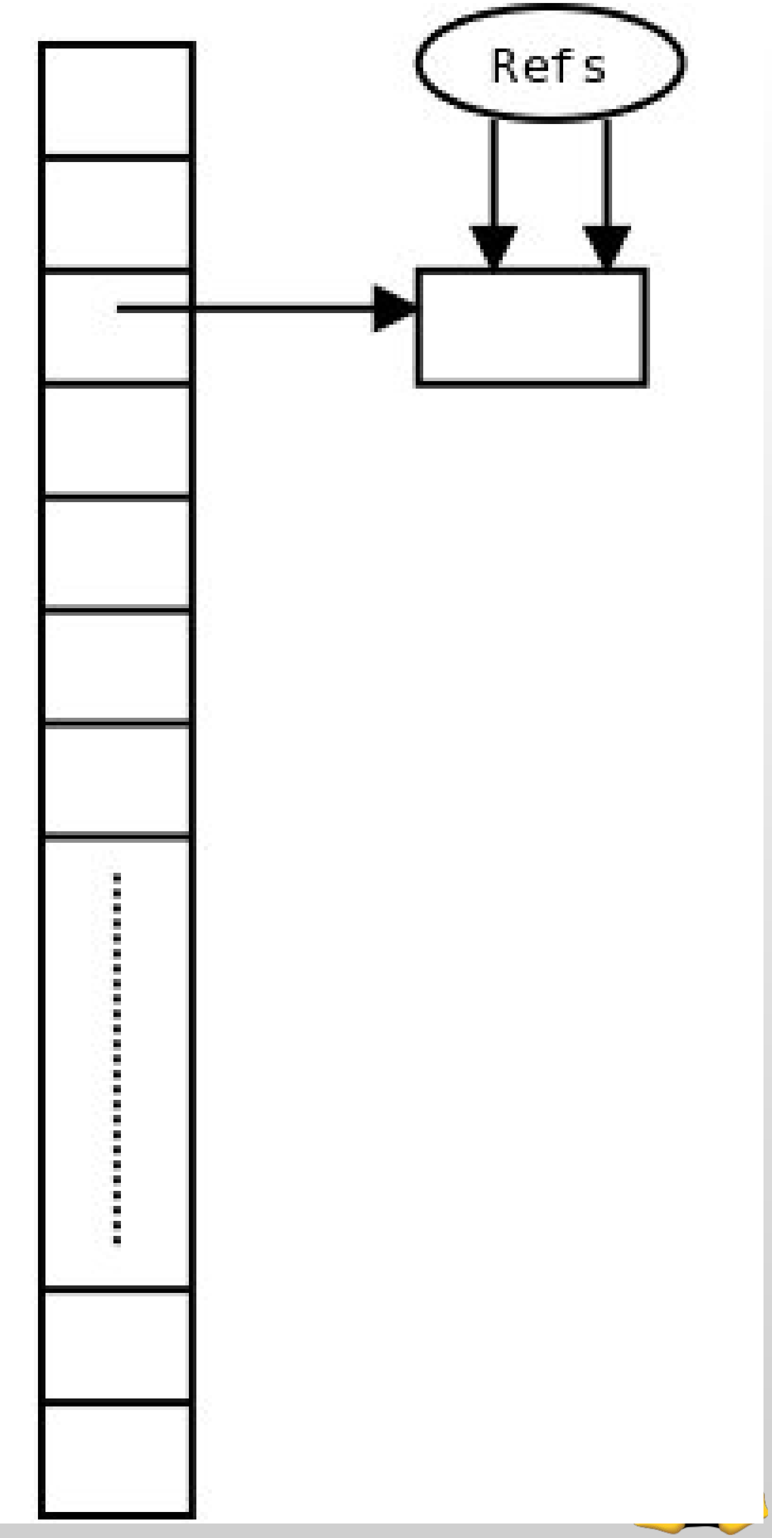
- An advanced locking-avoidance algorithm
  - Patented by IBM - GPL code only
- Useful for:
  - Frequently-read, rarely changed structures
  - Pointer-oriented data structures
- Several implementations
  - Lots of subtlety
  - http://lwn.net/Kernel/Index/ under read-copy-update

Example

Imagine an array of pointers to some structure of interest.

Kernel code holds some references to that structure

We need to update it.

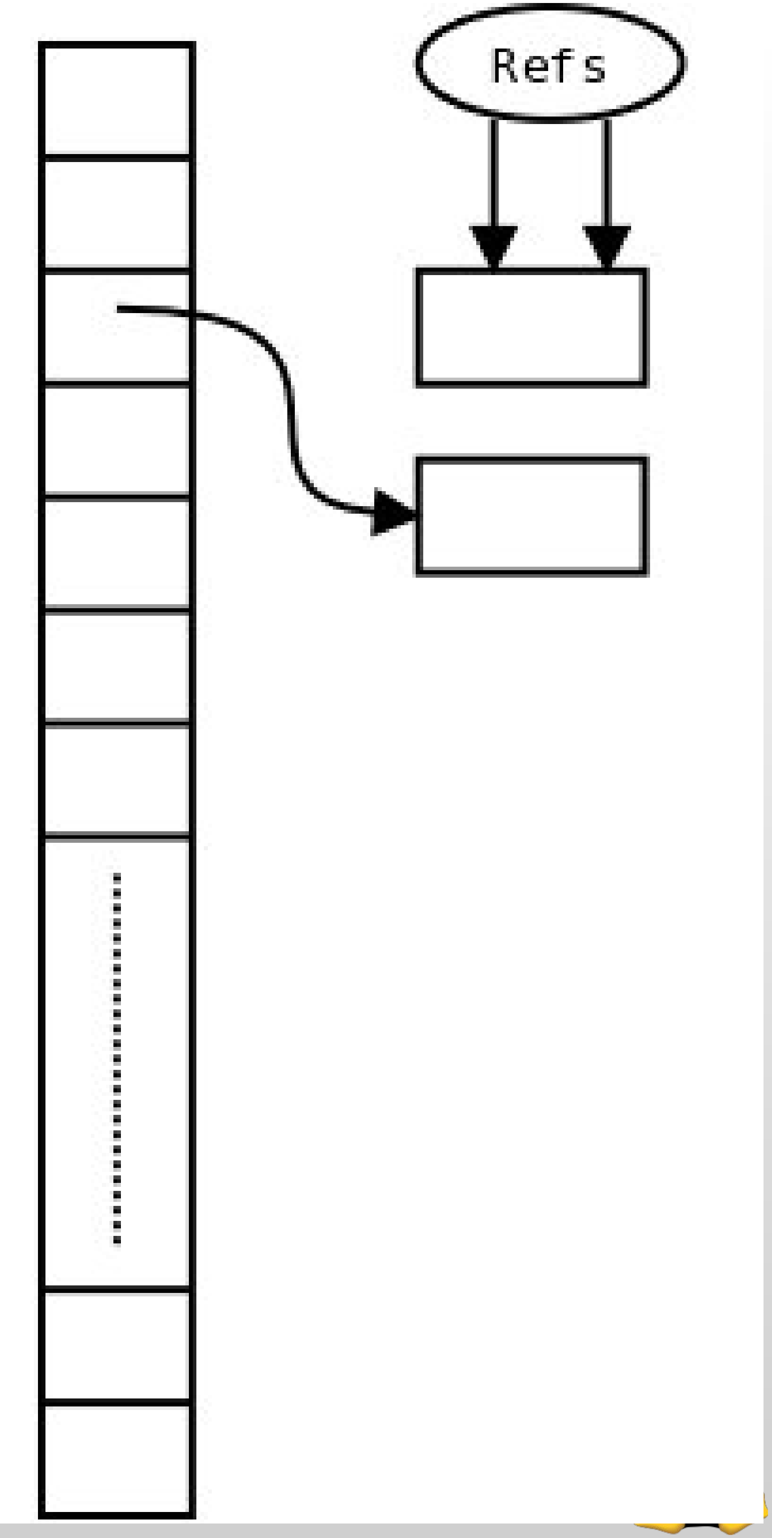


Step 1

Copy the object and update the information

Change the pointer to the new object

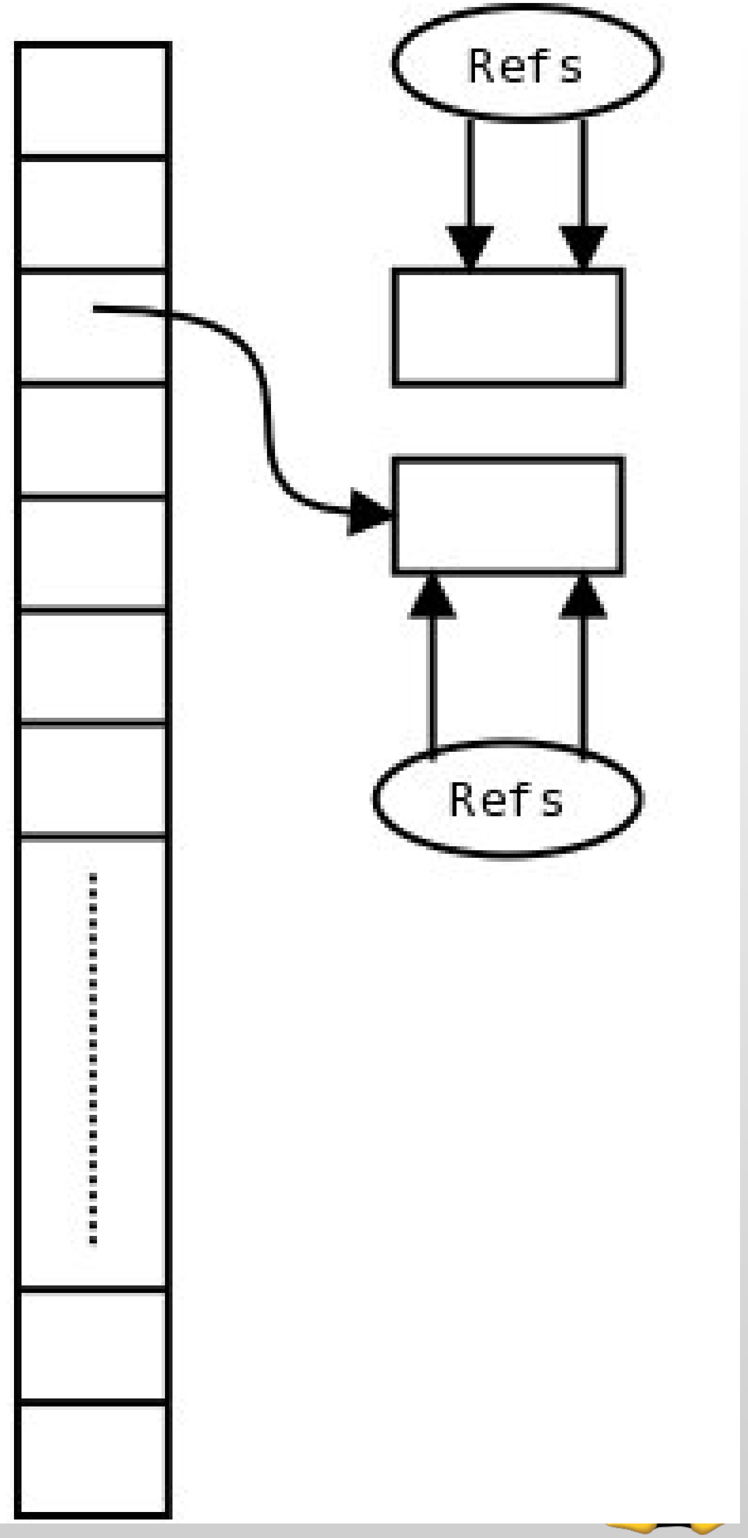
References to the old copy still exist



Step 2

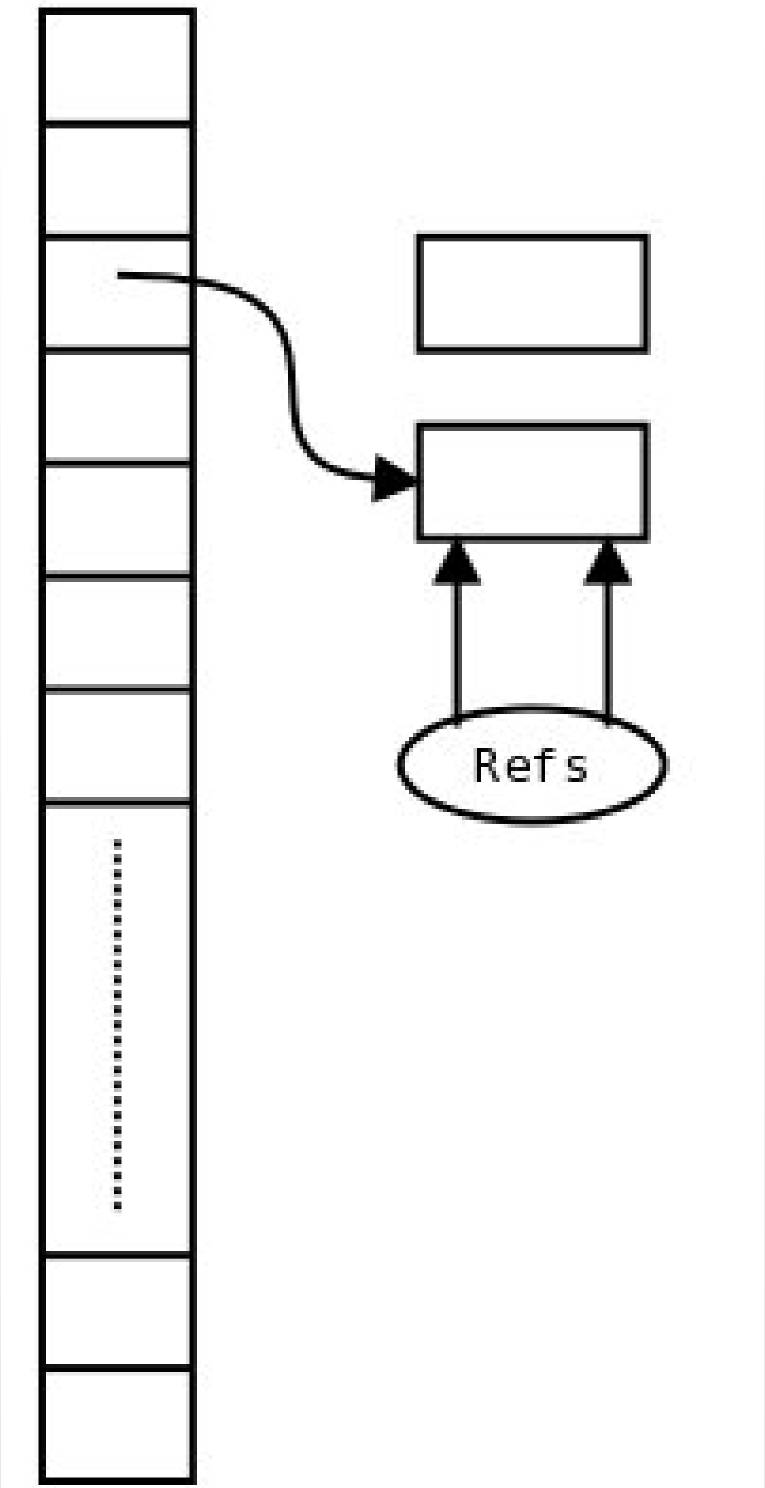
The new object may begin to gain references

The old one remains in use



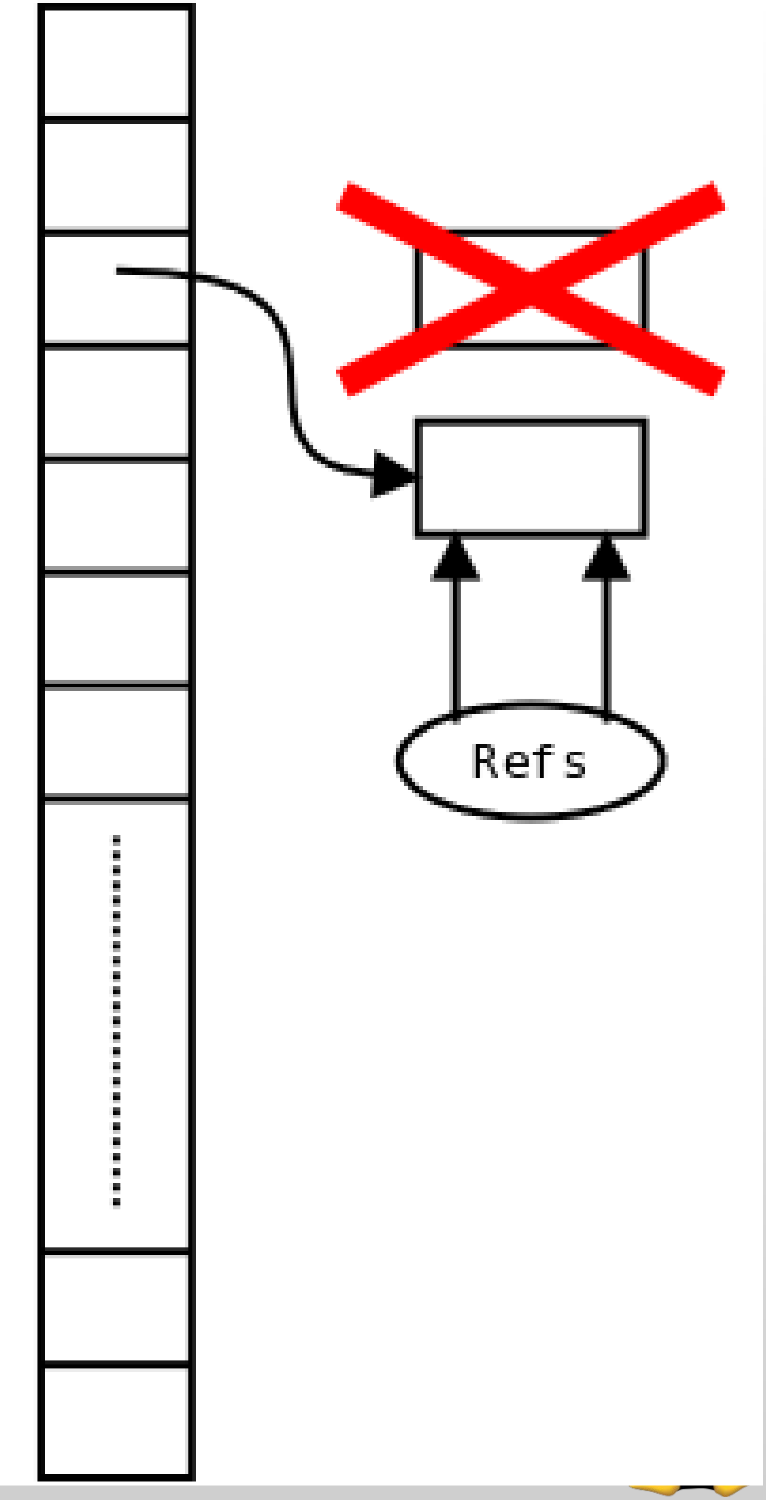
Step 4

- Eventually all users of the old object drop their references



Step 4

The old object may now be safely deleted.



RCU rules

Object may not be changed in place

- RCU must be used instead

Read access to objects in atomic code only

- Preemption must be disabled

References to objects cannot be kept past scheduling

Why these rules?

How do you know when all references are gone?

...When every processor has scheduled once

Using RCU

Read side

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

rcu_read_lock();   /* Disables preemption */
struct something *p = rcu_dereference(object);
...
rcu_read_unlock();
```

RCU write side

Embed this in your structure

```
struct rcu_head rcu;
```

When it is time to free the structure:

```
void call_rcu(struct rcu_head *rcu,
         void (*func)(struct rcu_head *rcu));
```

func() will be called when the structure can be freed

RCU Questions?

Realtime preemption

The goal of the realtime project

- Deterministic response times - always

Realtime makes determinism the top priority

- Ahead of throughput

Realtime changes

Spinlocks become mutexes

- The can sleep at any time
- Preemption not disabled
- Priority inheritance implemented

Old-style spinlocks still exist

- Called raw_spinlock_t;
- Use of these will attract scrutiny

Realtime changes

- Per-CPU variables no longer exist
  - Access protected by spinlocks
  - Long-term solution still unclear

Realtime changes

- Read-copy-update becomes more complex
  - Can't disable preemption
  - Can't wait for everybody to schedule
  - Throughput drops accordingly

The last slide

What else would you like to know?