MANCHESTER 1824
The University of Manchester

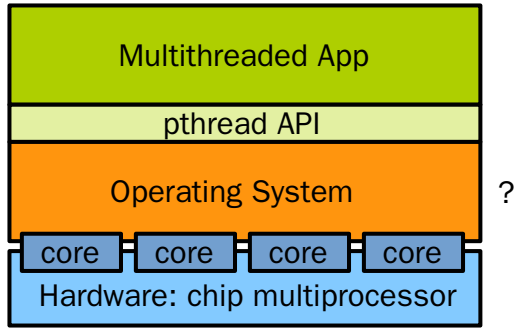
COMP35112 Chip Multiprocessors

Operating System Support for Multithreading

Pierre Olivier

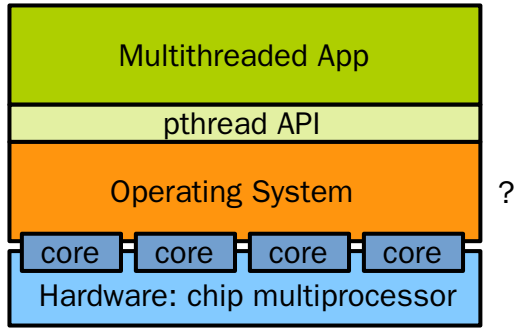
OS Support for Multithreading

- Aspects of chip multiprocessor we covered until now:
  - **Software** (user level): POSIX thread programming
  - **Hardware**: cores, cache coherence, atomic operations
- Let's have a look at what sits in between: the **operating system**



OS Support for Multithreading

- Aspects of chip multiprocessor we covered until now:
  - **Software** (user level): POSIX thread programming
  - **Hardware**: cores, cache coherence, atomic operations
- Let's have a look at what sits in between: the **operating system**

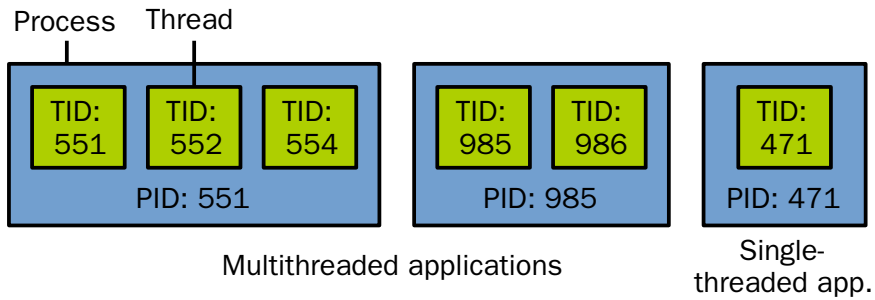


- Focusing on Linux, **what is the role of the operating system in the management and synchronisation of multithreaded programs**?
- **How is concurrency managed in the kernel**?

Thread Management

Thread Management

- A thread is a unique schedulable entity in the system
- Each process has 1 or more threads



- Each thread is uniquely identified by its TID
- Threads sharing the same address space will report the same PID (equals to the main thread's TID)
- Many system calls requiring a PID (e.g. `sched_setscheduler`) actually work on a TID, read man pages

Thread Management

```c
#define _GNU_SOURCE  // Required for gettid() on Linux
/* includes here */
void *thread_function(void *arg) {
    printf("child thread, pid: %d, tid: %d\n", getpid(), gettid());
    pthread_exit(NULL);
}

int main() {
    pthread_t thread1, thread2;
    printf("parent thread, pid: %d, tid: %d\n", getpid(), gettid());

    pthread_create(&thread1, NULL, threadFunction, NULL);
    pthread_create(&thread2, NULL, threadFunction, NULL);
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
}
```

Example of output:

```
parent thread, pid: 12674, tid: 12674
child thread, pid: 12674, tid: 12675
child thread, pid: 12674, tid: 12676
```

Thread Creation

- Processes and threads are created with the `clone()` system call:

```
long clone(unsigned long flags, void *stack, int *parent_tid, int *child_tid,
    unsigned long tls);
```

Thread Creation

- Processes and threads are created with the `clone()` system call:

```
long clone(unsigned long flags, void *stack, int *parent_tid, int *child_tid,
    unsigned long tls);
```

- **When creating a process** most of these parameters do not matter and are set to `0`/`NULL`
  - Behaviour similar to the `fork` UNIX primitive: parent's resources (including address space) are duplicated for the child

Thread Creation

- Processes and threads are created with the `clone()` system call:

```
long clone(unsigned long flags, void *stack, int *parent_tid, int *child_tid,
    unsigned long tls);
```

- **When creating a process** most of these parameters do not matter and are set to `0`/`NULL`
  - Behaviour similar to the `fork` UNIX primitive: parent's resources (including address space) are duplicated for the child
- **When creating a thread**:
  - `flags` specifies creation options
  - `stack` and `tls` point to the child's stack and thread local storage
  - `parent_tid` and `child_tid` specify where to store the id of the created thread

Thread Creation

- Musl libc's `pthread_create` implementation in [`src/thread/pthread_create.c`] (simplified):
  1. Prepare `clone`'s flags with `CLONE_VM | CLONE_THREAD` and more
  2. Allocate space for a stack (pointed by `stack`) with `mmap`, and create TLS area (pointed by `new`) with `__copy_tls`
  3. Place on that stack a data structure with the thread's argument and entry point
  4. Call `clone`'s wrapper `__clone` (simplified):

```
ret = __clone(start, stack, flags, args, &new->tid, TP_ADJ(new), &__thread_list_lock);
```

Thread Creation

```
__clone(start, stack, flags, args, &new->tid, TP_ADJ(new), &__thread_list_lock); // wrapper
clone(unsigned long flags, void *stack, int *ptid, int *ctid, unsigned long tls); // syscall
```

- `__clone` is implemented for x86-64 in src/thread/x86_64/clonse.s
- x86-64 calling convention: arguments in order in `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`, return value in `%rax`

```
__clone:
    xor %eax,%eax      // clear eax
    mov $56,%al        // clone's syscall id
    mov %rdi,%r11      // entry point in r11
    mov %rdx,%rdi      // flags in rdi
    mov %r8,%rdx       // parent_tid in rdx
    mov %r9,%r8        // TLS in r8
    mov 8(%rsp),%r10
    mov %r11,%r9       // entry point in r9
    and $-16,%rsi
    sub $8,%rsi        // stack in rsi
    mov %rcx,(%rsi)    // push thread args
```

```
    syscall         // actual call to clone
    test %eax,%eax  // check parent/child
    jnz 1f          // parent jump
    xor %ebp,%ebp   // child clears base pointer
    pop %rdi        // thread args in rdi
    call *%r9       // jump to entry point
    mov %eax,%edi   // ain't supposed to return
    xor %eax,%eax   // here, something's wrong
    mov $60,%al
    syscall         // exit (60 is exit's id)
    hlt
1:  ret             // parent returns
```

Thread Creation

Inside the kernel when `clone` is implemented in [`kernel/fork.c`]:

1. System call handler calls `sys_clone`
2. `sys_clone` calls `kernel_clone`, which calls `copy_process`
3. `copy_process` implements the parent's duplication:
   - Calls various functions checking `clone`'s flags to know what needs to be copied and what needs to be shared between parent/child
   - E.g. for the address space (simplified):

```
static int copy_mm(unsigned long clone_flags, struct task_struct *tsk) {
  /* ... */
    if (clone_flags & CLONE_VM) {
        mmget(oldmm);
        mm = oldmm;
    } else
        mm = dup_mm(tsk, current->mm);
  /* ... */
}
```

Thread Creation

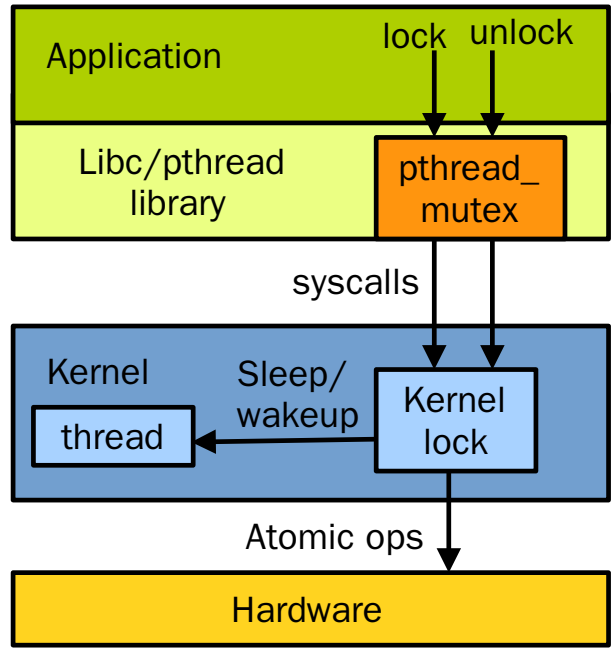
- Back to user space the thread starts at the `start` entry point defined by Musl in `src/thread/pthread_create.c`:
  - It extracts from the stack the `args` data structure containing the user-defined thread entry point and parameter
  - Then calls the entry point and passes it the parameter

```
static int start(void *p) {
  struct start_args *args = p;
  /* ... */
    __pthread_exit(args->start_func(args->start_arg));
}
```

Locks Implementations

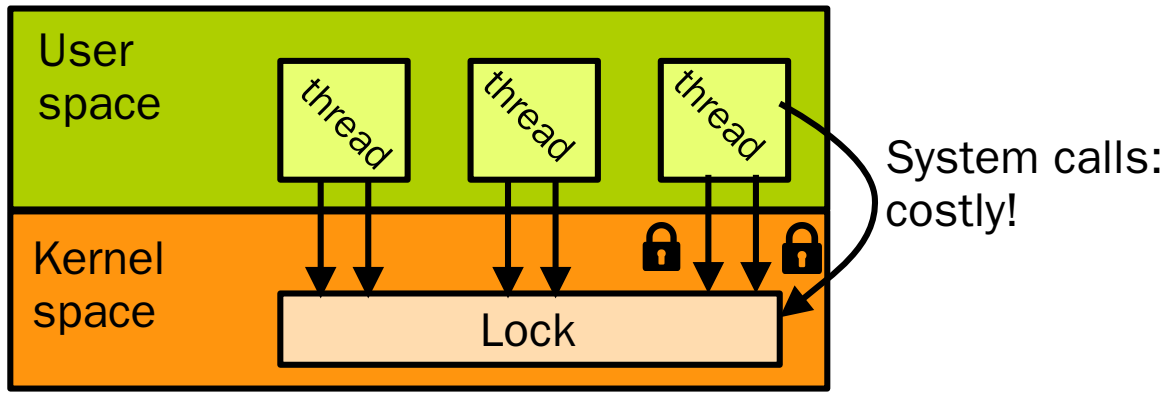
In-Kernel Locks

- Under the hood `pthread_mutex_lock` and other sleep-based lock access primitives rely on the kernel
- There is a good reason to **implement such locks in the kernel** rather than user space:
  - The kernel is the entity that can put threads to sleep and wake them up



In-Kernel Locks

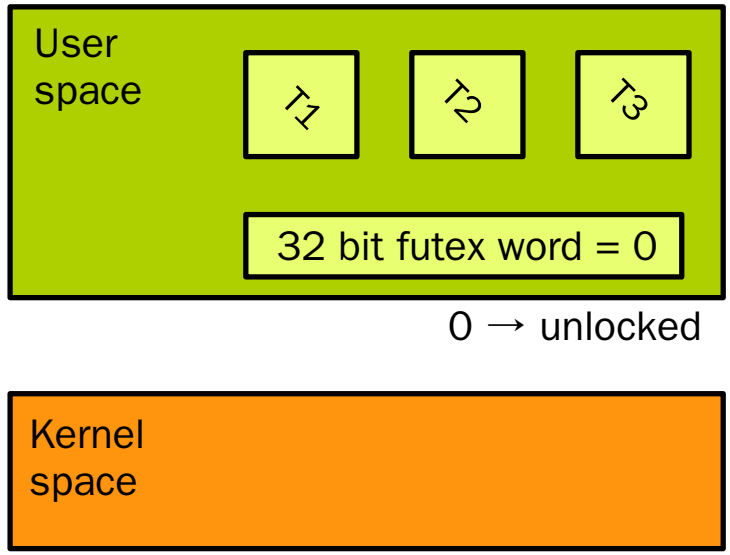
- Historically each lock operation, including lock take/release, required a system call
  - As implemented with e.g. System V semaphores
  - User/kernel world switches are **expensive** and the resulting overhead is non-negligible in low-contention scenarios



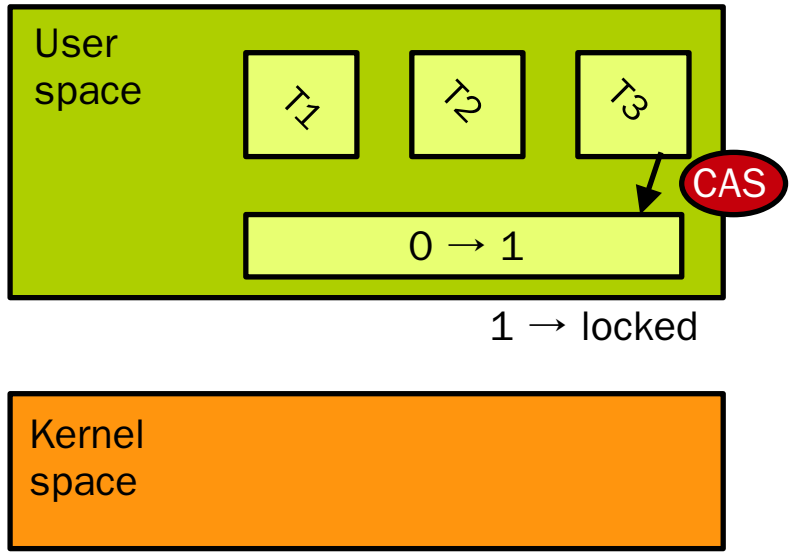
Futex

- Observation: we only need kernel intervention when there is contention, i.e. when a thread needs to sleep
- **Futex**: Fast User space mutEX
  - Lock implemented **in part in user space** with atomic operations when there is no contention
  - And **another part in kernel space** when there is contention and threads need to be put to sleep

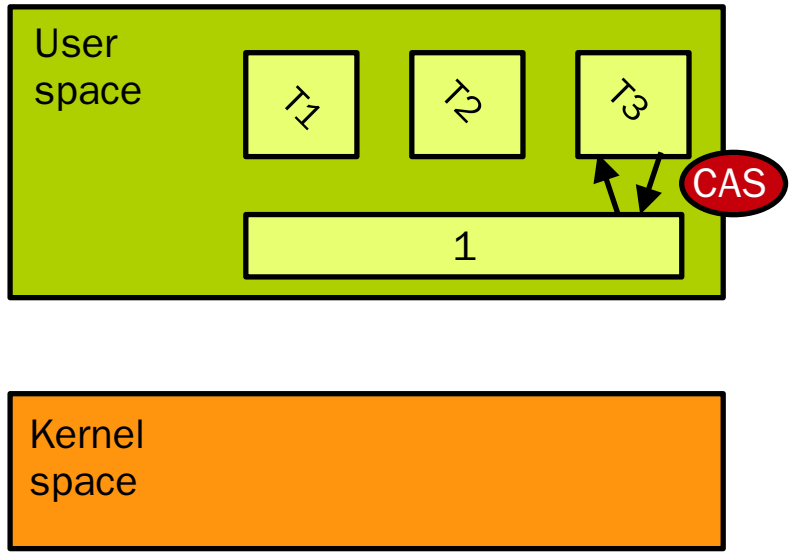
Futex


User space
T1
T2
T3
32 bit futex word = 0
0 → unlocked
Kernel space

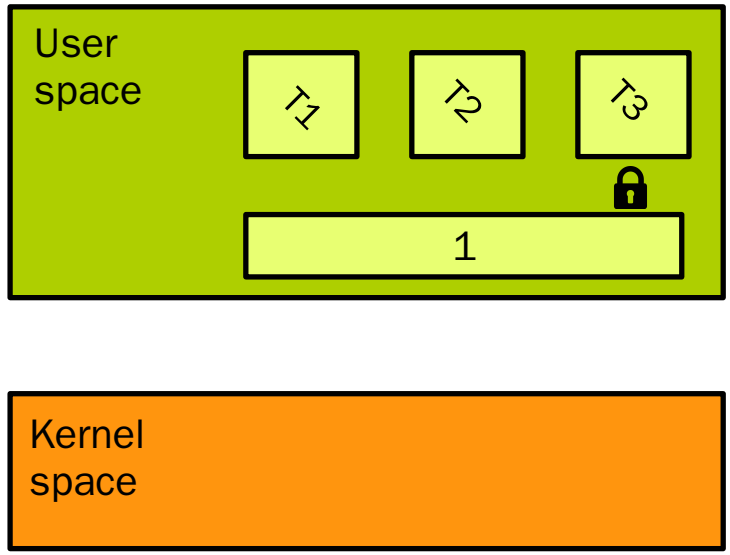
Futex



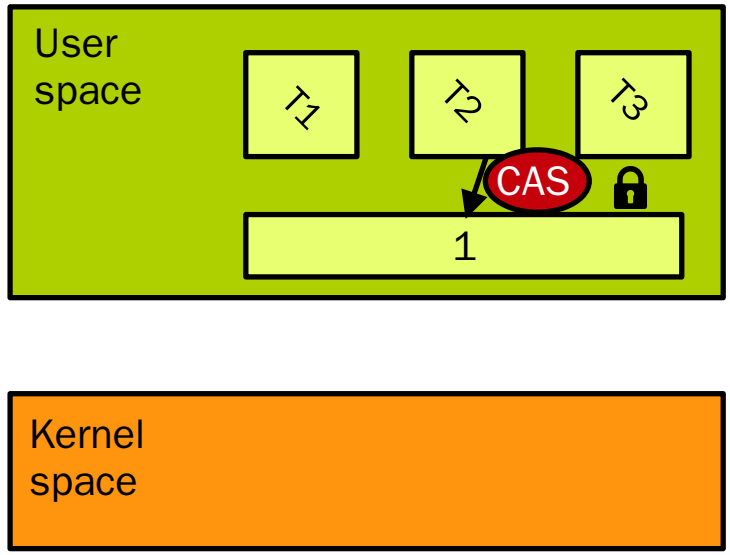
Futex



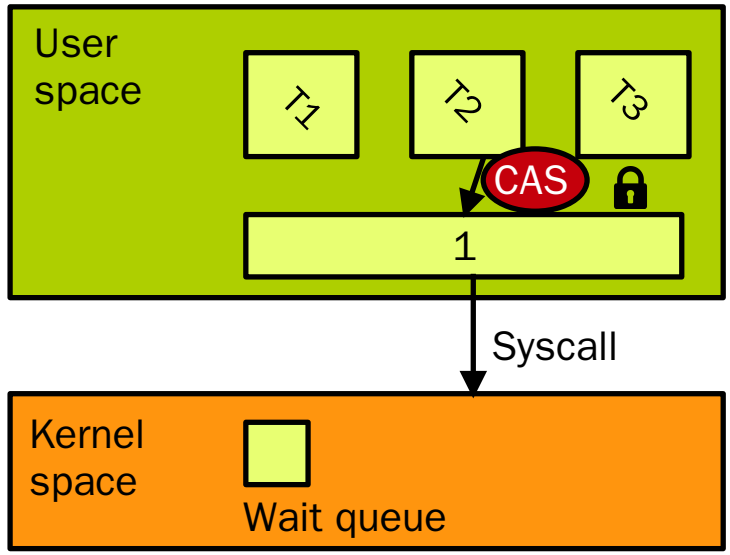
Futex



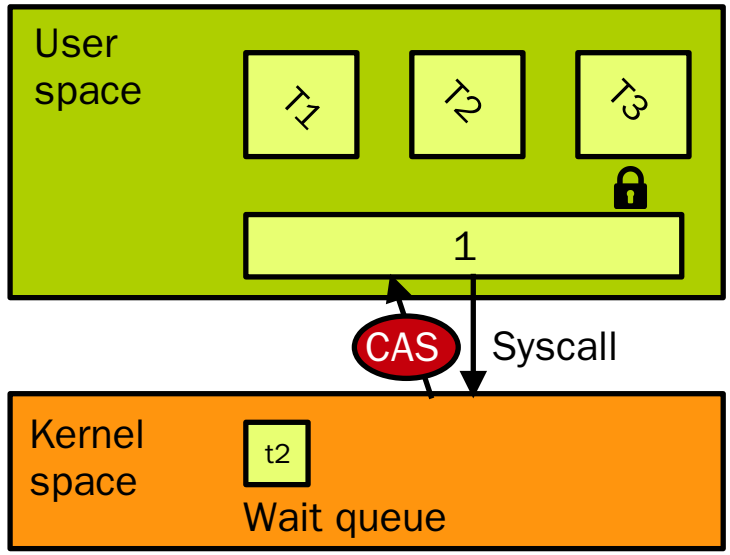
Futex



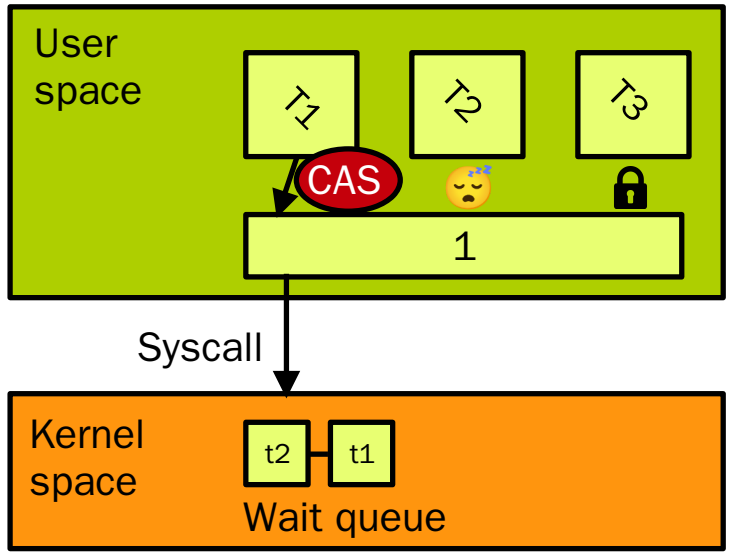
Futex



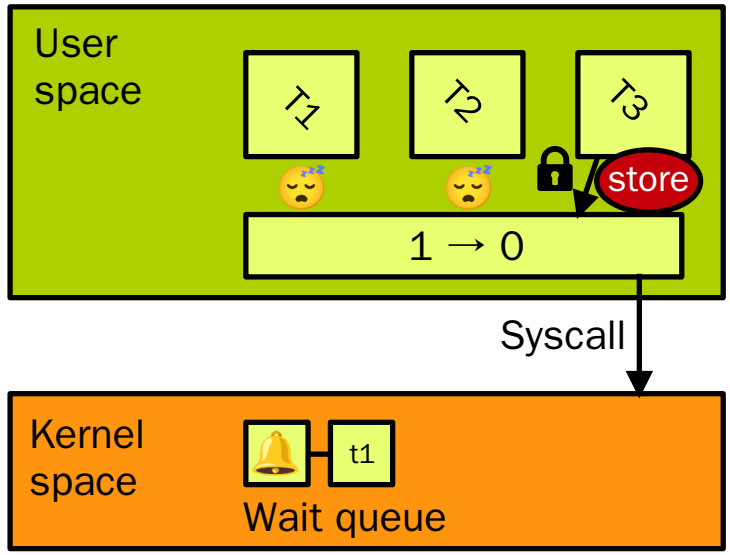
Futex



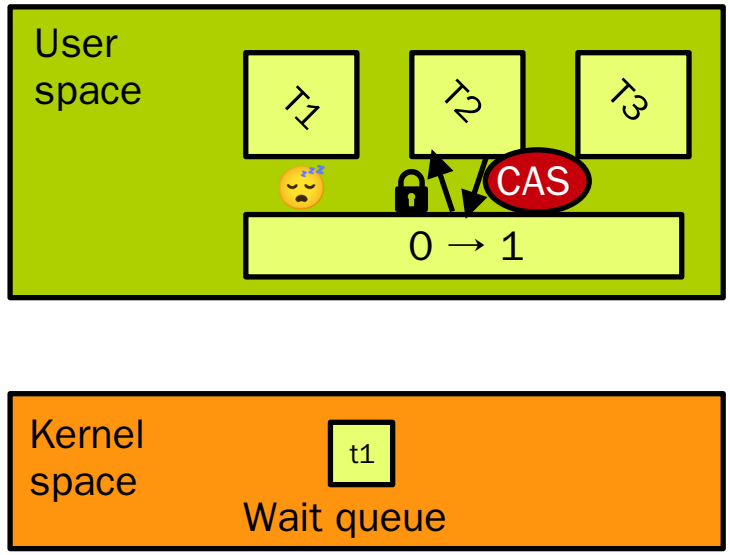
Futex


User space
T1
T2
T3
CAS
1
Syscall
Kernel space
t2
t1
Wait queue

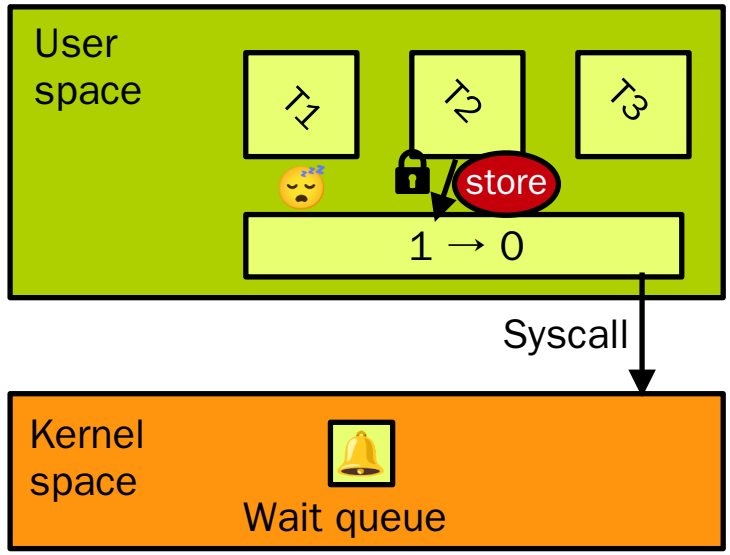
Futex


User space
T1
T2
T3
store
1 → 0
Syscall
Kernel space
t1
Wait queue

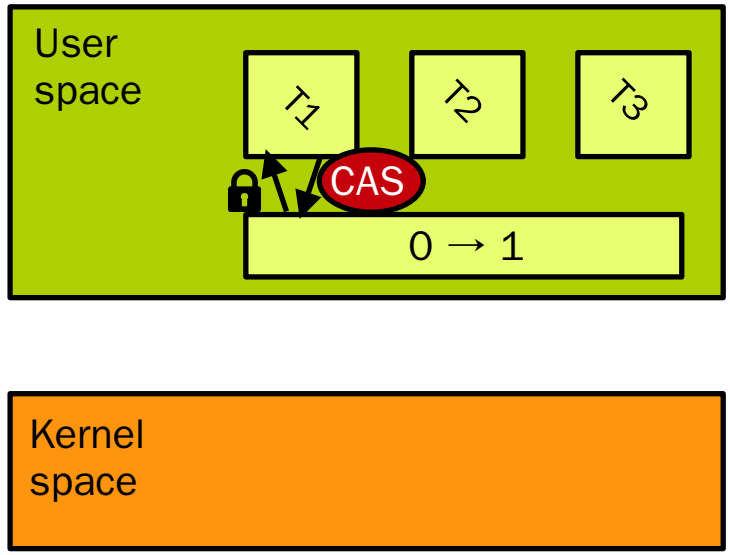
Futex



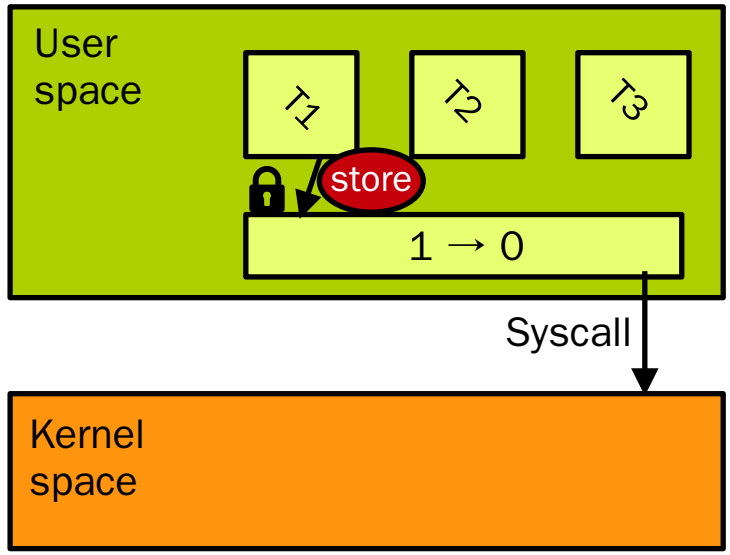
Futex



Futex



Futex



Basic Futex Lock Implementation

```c
/* check full implementation for includes */

atomic_int my_mutex = ATOMIC_VAR_INIT(0);

int my_mutex_lock() {
    int is_free = 0, taken = 1;

    // cas(value_to_test, expected_value, new_value_to_set)
    while(!atomic_compare_exchange_strong(&my_mutex, &is_free, taken)) {
        // put the thread to sleep waiting for FUTEX_WAKE if my_mutex is still equal to 1
        syscall(SYS_futex, &my_mutex, FUTEX_WAIT, 1, NULL, NULL, 0);
    }
    return 0;
}

int my_mutex_unlock() {
    atomic_store(&my_mutex, 0);

    // wake up 1 thread if needed
    syscall(SYS_futex, &my_mutex, FUTEX_WAKE, 1, NULL, NULL, 0);
    return 0;
}
```

12-os-support-for-multithreading/lock-bench-custom-futex.c

Investigating Lock Performance

- Measure latency introduced by locking/unlocking operations:

```
void *thread_function(void *arg) {
  for(int i=0; i < CS_NUM; i++) {

    lock();
    // instantaneous critical section to maximise the impact of the latency introduced by the
    // lock/unlock operations
    unlock();
  }

  return;
}
```

12-os-support-for-multithreading/lock-bench

Pthread Mutex Implementation

- Previous custom futex lock implementation is suboptimal (and not 100% correct https://www.akkadia.org/drepper/futex.pdf)
- Musl implementation of `pthread_mutex_lock` in `src/thread/pthread_mutex_lock.c`:

```
int __pthread_mutex_lock(pthread_mutex_t *m) {
    if ((m->_m_type&15) == PTHREAD_MUTEX_NORMAL
        && !a_cas(&m->_m_lock, 0, EBUSY))        // CAS, futex fast path
        return 0;

    return __pthread_mutex_timedlock(m, 0);    // Didn't get the lock
}
```

`pthread_mutex_timedlock` (in `src/thread/pthread_mutex_timedlock.c`) calls a bunch of functions that end up in `FUTEX_WAIT` being called in `__timedwait_cp` (in `src/thread/__timedwait.c`)

Pthread Mutex Implementation

- Musl implementation of `pthread_mutex_unlock` in [src/thread/pthread_mutex_unlock.c]():

```
int __pthread_mutex_unlock(pthread_mutex_t *m) {
  int waiters = m->_m_waiters;

  /* ... */

  cont = a_swap(&m->_m_lock, new);

  if (waiters || cont<0)
        __wake(&m->_m_lock, 1, priv);
}
```

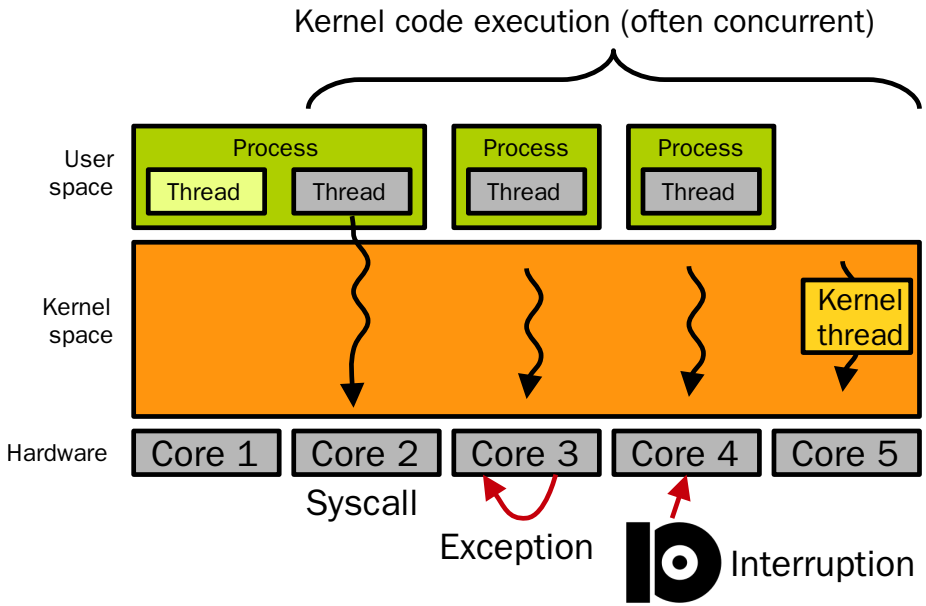
Concurrency in the Kernel

Concurrency in the Kernel

- There used to be a **big kernel lock** serialising all execution of kernel code
  - Slowly removed over time, removal finalised in v2.6.39 (2011)
- Today the **kernel is a highly concurrent, shared memory program**

Concurrency in the Kernel

- There used to be a **big kernel lock** serialising all execution of kernel code
  - Slowly removed over time, removal finalised in v2.6.39 (2011)
- Today the **kernel is a highly concurrent, shared memory program**



Sources of concurrency:
system calls, exceptions, hardware interrupts, kernel threads & more, preemption, etc.

Locking in the Kernel

- Similar to user space, in-kernel critical data sections must be protected by locks
- Different types available:

Locking in the Kernel

- Similar to user space, in-kernel critical data sections must be protected by locks
- Different types available:
  - **Mutexes**: sleep-based wait, usage count (number of entities that can hold the mutex) is 1

Locking in the Kernel

- Similar to user space, in-kernel critical data sections must be protected by locks
- Different types available:
  - **Mutexes**: sleep-based wait, usage count (number of entities that can hold the mutex) is 1
  - **Semaphores**: sleep-based wait, usage count can be >= 1

Locking in the Kernel

- Similar to user space, in-kernel critical data sections must be protected by locks
- Different types available:
  - **Mutexes**: sleep-based wait, usage count (number of entities that can hold the mutex) is 1
  - **Semaphores**: sleep-based wait, usage count can be >= 1
  - **Spinlocks**: busy waiting, usage count 1
    - Run with preemption and possibly interrupts disabled
    - **Usable when kernel execution cannot sleep** (e.g. interrupt context, preemption disabled)

Locking in the Kernel

- Similar to user space, in-kernel critical data sections must be protected by locks
- Different types available:
  - **Mutexes**: sleep-based wait, usage count (number of entities that can hold the mutex) is 1
  - **Semaphores**: sleep-based wait, usage count can be >= 1
  - **Spinlocks**: busy waiting, usage count 1
    - Run with preemption and possibly interrupts disabled
    - **Usable when kernel execution cannot sleep** (e.g. interrupt context, preemption disabled)
  - **Completion variables** (~condition variables)

Locking in the Kernel

- Similar to user space, in-kernel critical data sections must be protected by locks
- Different types available:
  - **Mutexes**: sleep-based wait, usage count (number of entities that can hold the mutex) is 1
  - **Semaphores**: sleep-based wait, usage count can be >= 1
  - **Spinlocks**: busy waiting, usage count 1
    - Run with preemption and possibly interrupts disabled
    - **Usable when kernel execution cannot sleep** (e.g. interrupt context, preemption disabled)
  - **Completion variables** (~condition variables)
  - **Reader-writer spinlocks**
  - **Sequential locks**

Spinlocks in the Kernel

Interrupt handler for the i8042 mouse/keyboard driver in `drivers/input/serio/i8042.c`:

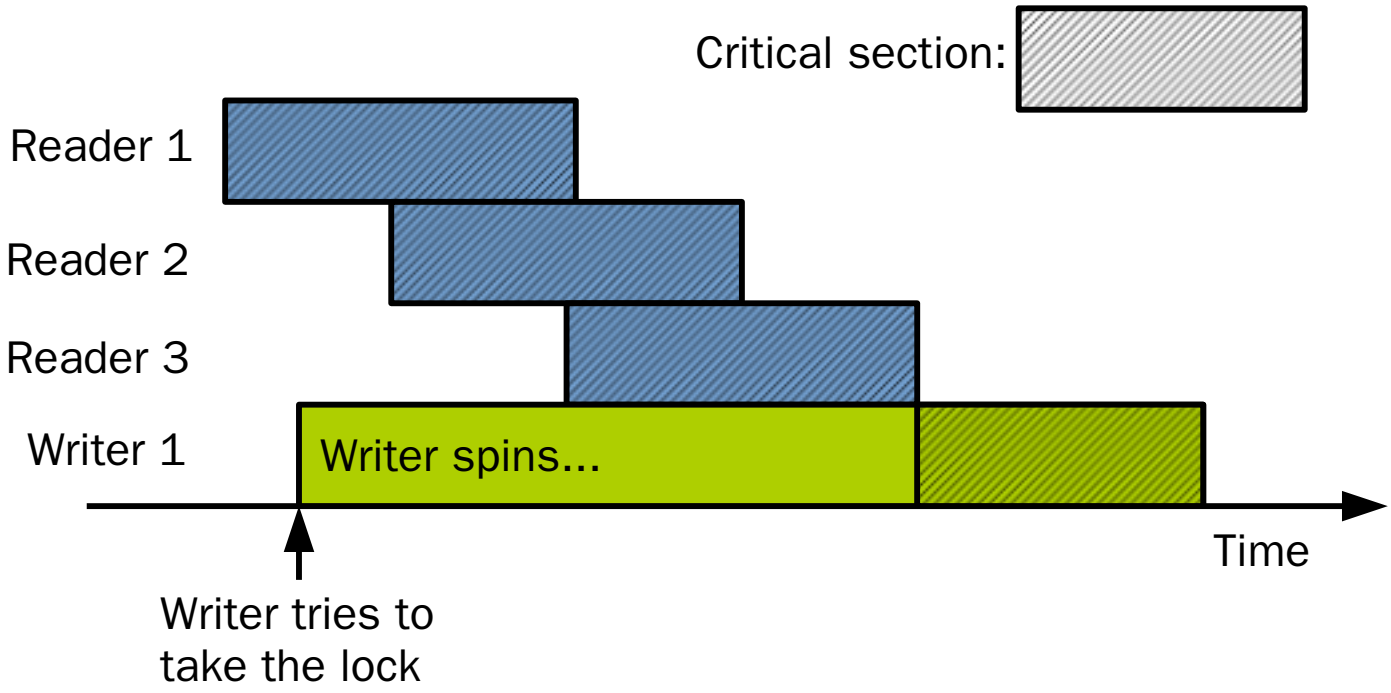
```
static irqreturn_t i8042_interrupt(int irq, void *dev_id) {

  spin_lock_irqsave(&i8042_lock, flags);
  /* read data from device */
  spin_unlock_irqrestore(&i8042_lock, flags);
}
```

- Spinlock needed, interrupt context cannot sleep (not a schedulable entity)
- `spin_lock_irqsave` takes the lock and disable interrupts if they are not already disabled
- `spin_unlock_irqrestore` releases the lock and restore interrupt if they were enabled when `spin_lock_irqsave` was called

Reader-Writer Spinlocks

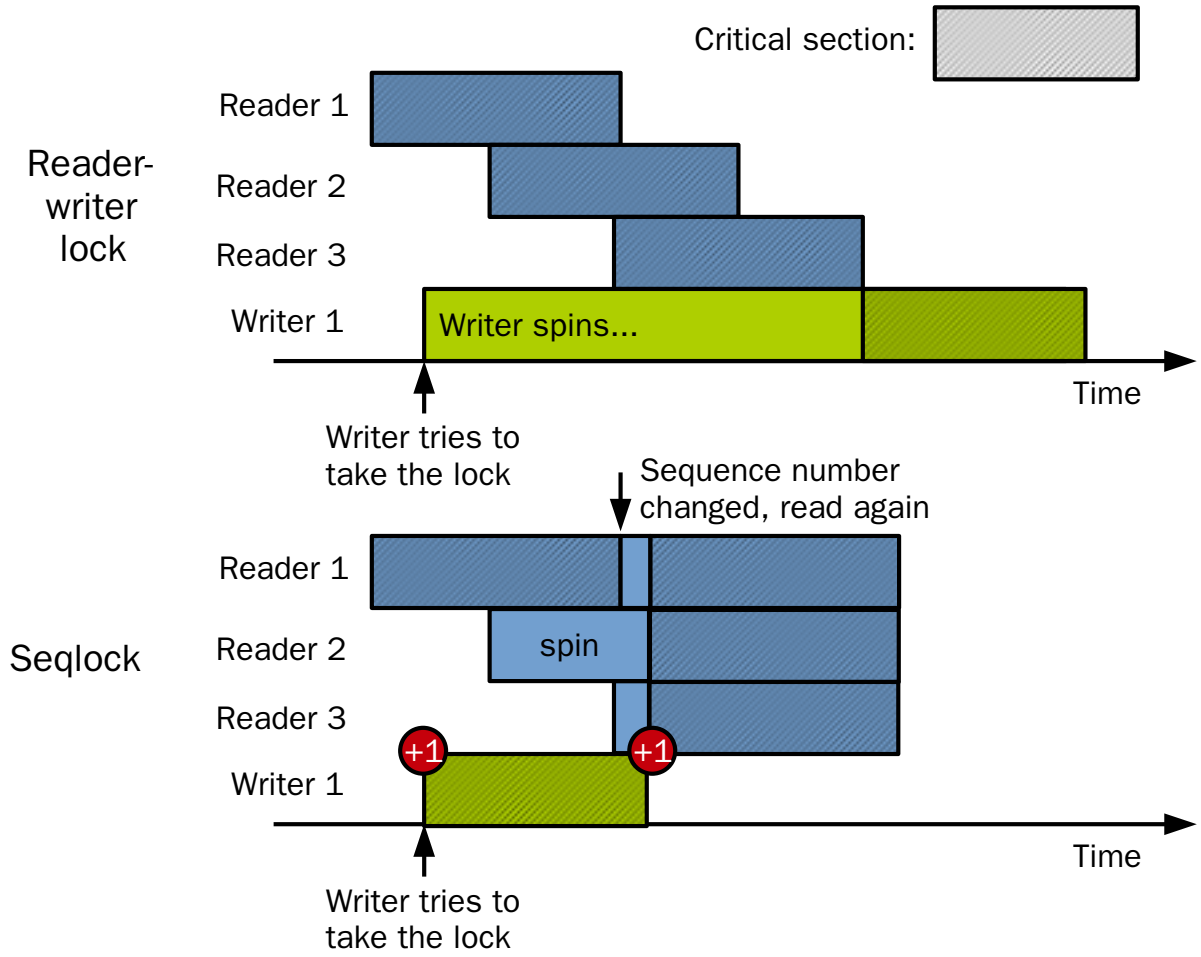
- Serialises write accesses with other (read/write accesses), but allows concurrent readers



Sequential Locks

- Lock has a **sequence number** associated:
  - Incremented each time a writer acquires the lock
  - Incremented each time a writer releases the lock
- Concurrent readers are allowed, they check the number at the beginning and end of their critical section
  - If it has changed, a writer started/finished during teh reader's critical section
    - Need to read again: reader restarts its critical section
- Seqlocks scales to many readers like reader-writer locks, but **favour writers**

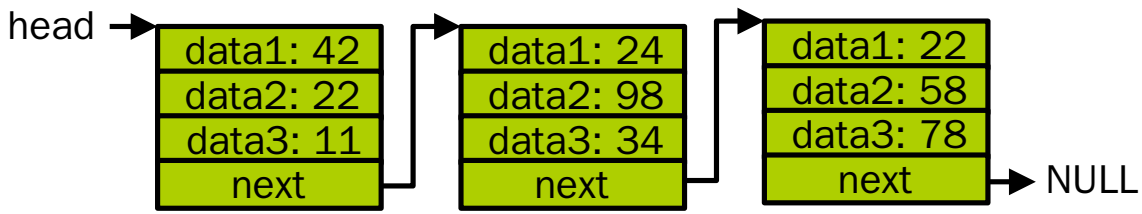
Sequential Locks



Read-Copy-Update

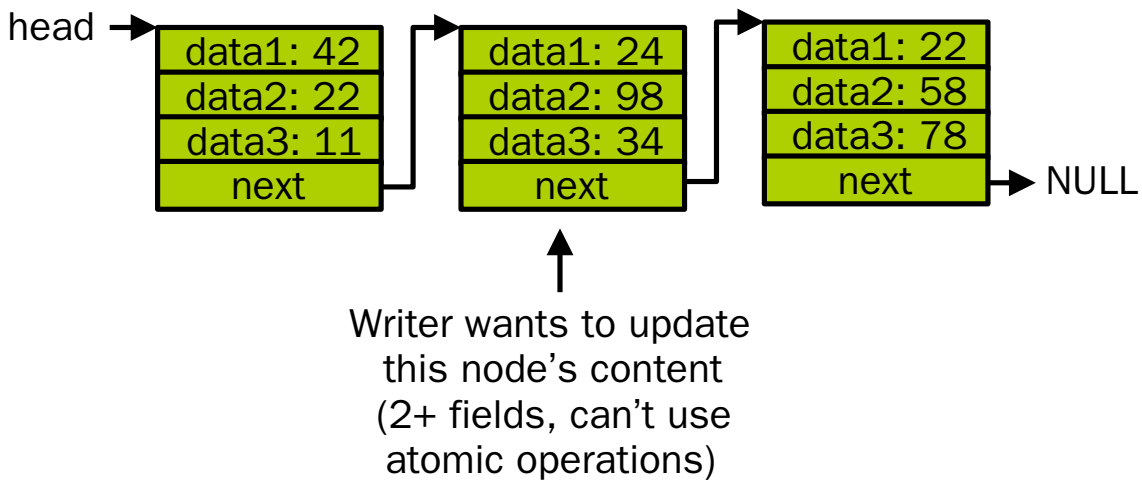
- RCU protects a critical section and **allows concurrent lockless readers**
- For situations in which **it's OK for concurrent readers not to see the same state for a given piece of data**, as long as **all see a consistent state**
- Extensively used in the kernel

RCU Example: Linked List Update

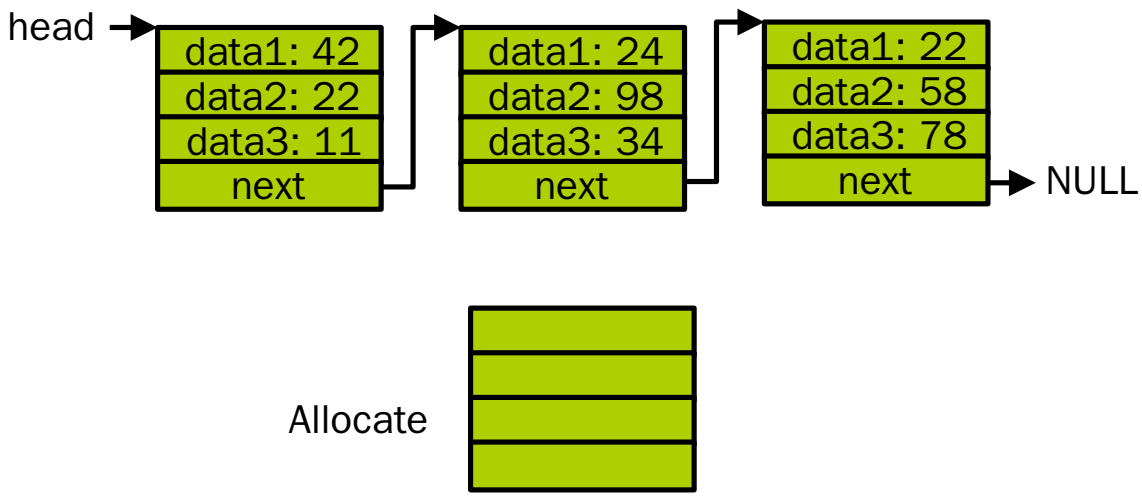


- Singly linked list
- Readers traverse the list to read data from certain nodes
- Writers traverse the list to update the data contained in certain nodes
- Writers also add/delete elements

RCU Example: Linked List Update

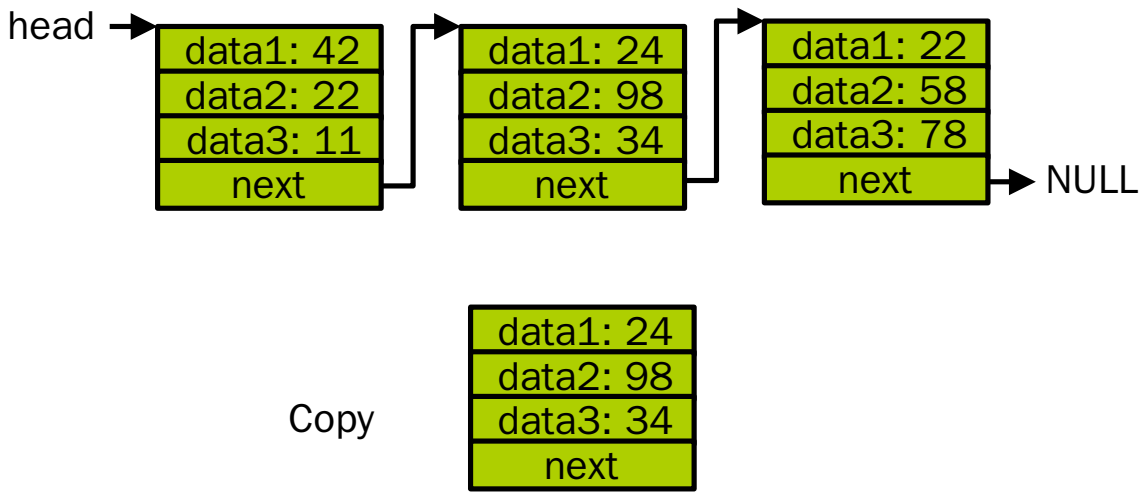


RCU Example: Linked List Update



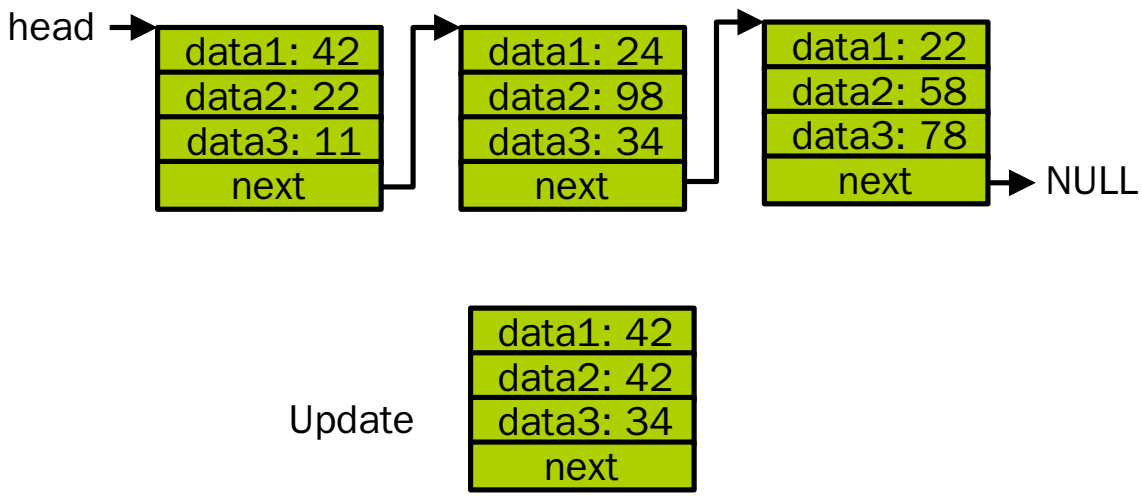
1. Allocate a new node

RCU Example: Linked List Update



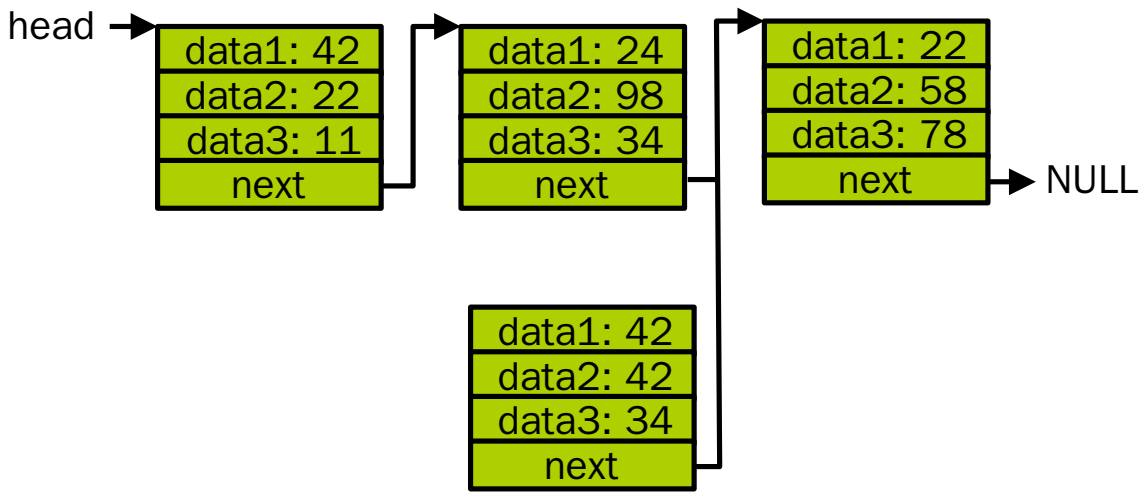
1. Allocate a new node
2. Copy target node data

RCU Example: Linked List Update



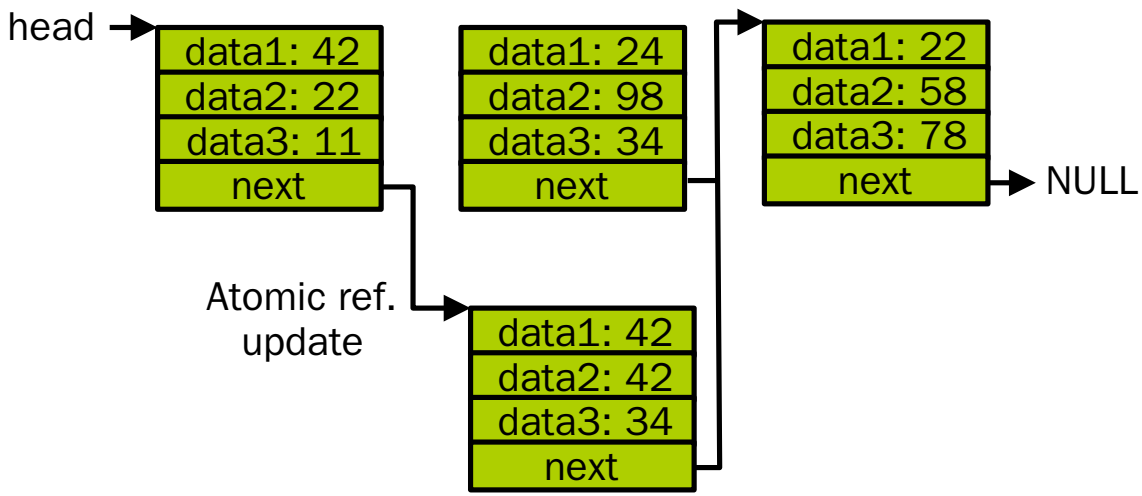
1. Allocate a new node
2. Copy target node data
3. Perform update

RCU Example: Linked List Update



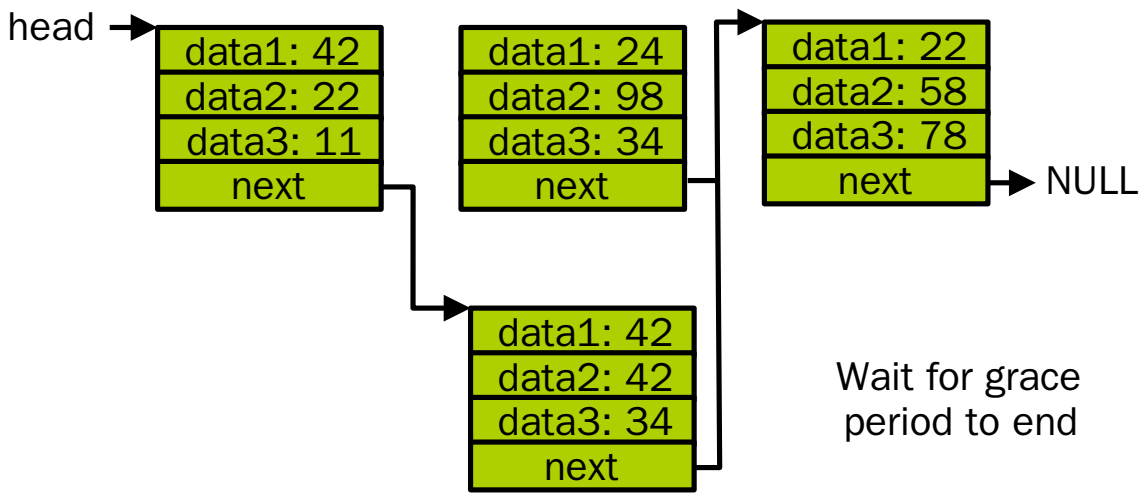
1. Allocate a new node
2. Copy target node data
3. Perform update
4. Point to next node

RCU Example: Linked List Update



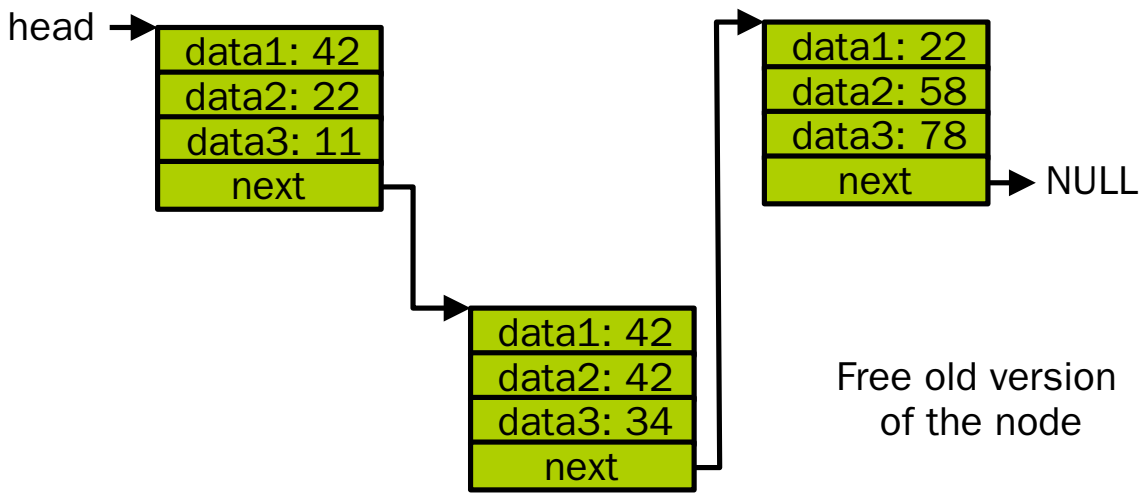
1. Allocate a new node
2. Copy target node data
3. Perform update
4. Point to next node
5. **Atomically swap previous node next pointer**

RCU Example: Linked List Update



1. Allocate a new node
2. Copy target node data
3. Perform update
4. Point to next node
5. **Atomically swap previous node next pointer**
6. Grace period: wait for outstanding readers on the old queue state to finish

RCU Example: Linked List Update



1. Allocate a new node
2. Copy target node data
3. Perform update
4. Point to next node
5. **Atomically swap previous node next pointer**
6. Grace period: wait for outstanding readers on the old queue state to finish
7. Free old node

RCU Example in Linux

From https://www.kernel.org/doc/html/next/RCU/whatisRCU.html

```
struct foo { int a; int b; int c; };
DEFINE_SPINLOCK(foo_mutex);
struct foo __rcu *gbl_foo;

void foo_write(int new_ab) {
  struct foo *new_fp, *old_fp;

  new_fp = kmalloc(sizeof(*new_fp),
      GFP_KERNEL); // allocate new data

  spin_lock(&foo_mutex); // serialise writers
  // get a ref to the data:
  old_fp = rcu_dereference_protected(gbl_foo,
      lockdep_is_held(&foo_mutex));
  *new_fp = *old_fp; // copy data
  new_fp->a = new_ab; // update data
  new_fp->b = new_ab; // update data
  // atmoic ref update:
  rcu_assign_pointer(gbl_foo, new_fp);
  spin_unlock(&foo_mutex);
  synchronize_rcu(); // wait for grace period
  kfree(old_fp);     // free old data
}

void foo_read(void) {
    struct foo *fp;
    int a, b, c;

    rcu_read_lock();
    fp = rcu_dereference(gbl_foo);
    a = fp->a;
    b = fp->b;
    c = fp->c;
    rcu_read_unlock();

  /* do something with a, b, c... */
}
```

12-os-support-for-multithreading/rcu

Wrapping Up

- **The kernel sits between multithreaded applications and the multicore hardware**
  - Involved in thread management and many synchronisation primitives
- **The kernel is itself a highly concurrent program**
  - Needs its own synchronisation primitives