

Synchronization II

- Lock implementation
 - interrupts
 - spinlock
 - yield()
 - queues
 - futex
- cache coherency
- Lock Free Data Structures
 - [Lockless patterns: more read-modify-write operations \[LWN.net\]](#)
 - C11 Atomic operations library
 - [Atomic operations library](#)
 - [memory_order - cppreference.com](#)
 - slides: [Memory barriers in C](#)
 - linux kernel memory barriers: [Linux kernel documentation on memory barriers](#) [An introduction to lockless algorithms \[LWN.net\]](#)
<https://www.scs.stanford.edu/23wi-cs212/sched/readings/why-memory-barriers.pdf>
- RCU
- Deadlock
- Transactions

Reminder

- thread executions are **interleaved!**
- **non-preemptive** threads
 - 1 thread executes exclusively
- **preemptive** threads
 - may switch to another thread between instructions
- **Multiple CPU** is inherently **preemptive**

Program A: Can both critical sections run?

```
int flag1 = 0, flag2 = 0;
void p1(void *ignored) {
    flag1 = 1;
    if (!flag2) {
        critical_section_1();
    }
}
void p2(void *ignored) {
    flag2 = 1;
    if (!flag1) {
        critical_section_2();
    }
}
int main() {
    tid id = thread_create(p1, NULL);
    p2();
    thread_join(id);
}
```

```
int data = 0;
int ready = 0;
void p1(void *ignored) {
    data = 2000;
    ready = 1;
}
void p2(void *ignored) {
    while (!ready)
        ;
    use(data);
}
int main() { ... }
```

Program B:
Can `use()` be called with value 0:
`use(0)`?

Program C: If p1–3 run concurrently, can use be called with value 0?

```
int a = 0;
int b = 0;
void p1(void *ignored) { a = 1; }

void p2(void *ignored) {
    if (a == 1) b = 1;
}

void p3(void *ignored) {
    if (b == 1) use(a);
}
```

Answers

We do not know!

- It depends on **what machine** you use
- If a system provides **sequential consistency**,
 - then answers all No
- But not all hardware provides **sequential consistency**

Why doesn't all hardware support sequential consistency?
→ many of the compiler and processor optimizations
would be illegal!

Sequential consistency(SC): The result of execution is as if all operations were executed in some sequential order, and the operations of each processor occurred in the order specified by the program. – Lamport

SC prevents hardware optimizations

- Complicates write buffers
 - E.g., read $\text{flag}(n)$ before $\text{flag}(3 - n)$ written through in Program A
- Can't re-order overlapping write operations
 - Concurrent writes to different memory modules
 - Coalescing writes to same cache line
- Complicates non-blocking reads
 - E.g., speculatively prefetch data in Program B
- Makes cache coherence more expensive
 - Must delay write completion until invalidation/update (Program B)
 - Can't allow overlapping updates if no globally visible order (Program C)

SC prevents hardware optimizations

- Code motion
- Caching value in register
 - Collapse multiple loads/stores of same address into one operation
- Common subexpression elimination
 - Could cause memory location to be read fewer times
- Loop blocking
 - Re-arrange loops for better cache performance
- Software pipelining
 - Move instructions across iterations of a loop to overlap instruction latency with branch cost

x86 consistency [intel 3a, §8.2]

[Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1](#)

- x86 supports multiple consistency/caching models
 - Memory Type Range Registers (MTRR) specify consistency for ranges of physical memory (e.g., frame buffer)
 - Page Attribute Table (PAT) allows control for each 4K page
- Choices include:
 - WB: Write-back caching (the default)
 - WT: Write-through caching (all writes go to memory)
 - UC: Uncacheable (for device memory)
 - WC: Write-combining – weak consistency & no caching (used for frame buffers, when sending a lot of data to GPU)
- Some instructions have weaker consistency
 - String instructions (written cache-lines can be re-ordered)
 - Special “non-temporal” store instructions (movnt*) that bypass cache and can be re-ordered with respect to other writes

- Old x86s (e.g, 486, Pentium 1) had almost SC
 - Exception: A read could finish before an earlier write to a different location
 - Which of Programs A, B, C might be affected?
 - just A
- Newer x86s also let a CPU read its own writes early

```
volatile int flag1;

int p1 (void)
{
    register int f, g;
    flag1 = 1;
    f = flag1;
    g = flag2;
    return 2*f + g;
}
```

```
volatile int flag2;

int p2 (void)
{
    register int f, g;
    flag2 = 1;
    f = flag2;
    g = flag1;
    return 2*f + g;
}
```

- E.g., both p1 and p2 can return 2:
- Older CPUs would wait at “f = ...” until store complete

Assuming sequential
consistency

Peterson's solution

```
int flag[2] = {false, false}; /*flag[i] indicates that Pi wants to enter critical section (it's
ready)*/

int turn = 0; /*indicates which process has the priority (lock) to enter in its CS*/
// P0                                     // P1

while (true){                               while (true){

    // wants to enter                       // wants to enter
    flag[0] = true;                         flag[1] = true;
    turn = 1;                               turn = 0;

    while (flag[1] && turn == 1){           while (flag[0] && turn == 0) {
        ;
    }
    /* critical section */
    flag[0] = false;

    /* remainder section */
}
```

This will not work in modern architectures:
For multithreaded programs, reordering of the statements cause inconsistency!

example

<https://preshing.com/20120515/memory-reordering-caught-in-the-act/>

Peterson expensive, only works for 2 processes

- Can generalize to n , but for some fixed n

Must adapt to machine memory model if not SC

- If you need machine-specific barriers anyway, might as well take advantage of other instructions helpful for synchronization

→ **Want to insulate programmer from implementing synchronization primitives**

◆ **thread library packages**

Libraries for User Apps

Pthread and other libraries

In System Programming and OOP courses, we have seen that there are libraries that allow us to write multithreaded programs.

Thread packages typically provide mutexes:

```
void mutex_init (mutex_t *m, ...);  
void mutex_lock (mutex_t *m);  
int mutex_trylock (mutex_t *m);  
void mutex_unlock (mutex_t *m);
```

- Only one thread acquires m at a time, others wait

Pthread library (Posix thread library) is such an example that provides an API for multithreaded user programs. In the pthread library, we have seen two different synchronization tools: mutex and condition variables.

Mutexes

```
//m1 is a mutex variable  
mutex_lock(m1); //acquire lock  
critical_section  
mutex_unlock(m1); //release lock
```

- You can consider lock as **“a mic among participants that controls who has the right to speak”**,
 - i.e whoever has the mic (m1) has the right to speak (in our case do ops on the memory shared among the participants).

Monitors(Monitors = cond var + mutex)

//two threads: e.g. producer/consumer

//Condition variables: c1 is for one condition, c2 is for another condition

//Locks: m1 to control the access to the critical section.

```
//1st thread
mutex_lock(m1);
/*critical section entry*/
while(need_to_wait_1 ){
    cond_wait(c1, m1);
}

/*critical section exit*/
cond_signal(c2) //or broadcast
mutex_unlock(m1);
```

```
// 2nd thread
mutex_lock(m1);
/*critical section entry*/
while(need_to_wait_2){
    cond_wait(c2, m1);
};

/*critical section exit*/
cond_signal(c1) //or broadcast
mutex_unlock(m1);
```


Monitors = cond var + mutex

Always acquire lock before accessing shared data

– Use condition variables to wait inside critical section

→ Three Operations: Wait(), Signal(), and Broadcast()

- Monitors represent the logic of the program
 - Wait if necessary
 - Signal when change something so any waiting threads can proceed

```
//1st thread
mutex_lock(m1);
/*critical section entry*/
while(need_to_wait_1 ){
    cond_wait(c1, m1);
}
/*critical section exit*/
cond_signal(c2) //or broadcast
mutex_unlock(m1);
```

```
// 2nd thread
mutex_lock(m1);
/*critical section entry*/
while(need_to_wait_2){
    cond_wait(c2, m1);
};
/*critical section exit*/
cond_signal(c1) //or broadcast
mutex_unlock(m1);
```

Semaphores

```
sem_t s;  
sem_init(&s, 0, 10);  
//a thread that wants to occupy a chair  
sem_wait(&s); //down the value of s by 1  
  
//After done with the chair  
sem_post(&s); //up the value of s by 1
```

- If the return value of `sem_wait` is negative the thread waits as done in the mutex locks.
 - This happens when the value of `s` before `sem_wait` is 0.
- `sem_wait` and `sem_post` can be called from different processes/threads.

In Linux kernel

- Pthread API is implemented by using NPTL ([Native POSIX Thread Library - Wikipedia](#)).
 - It uses the system calls such as **clone** and **futex**, and atomic operations to create a library in glibc ([The GNU C Library](#))
 - (see [pthread_create.c source code \[glibc/nptl/pthread_create.c\] - Codebrowser](#)).

There are also alternative/different pthread implementations for Linux.

In the kernel space,

- similarly to processes, you can also spawn threads by using kernel threads(kthreads).
- There is also similar lock mechanism you can use(see [locking — The Linux Kernel documentation](#))

Lock implementation

both user/kernel need
synchronization!

Goals:

- **Correctness**
 - Mutual exclusion: only one thread in critical section at a time
 - Progress (deadlock-free): if several simultaneous requests, must allow one to proceed
 - Bounded wait (starvation-free): must eventually allow each waiting thread to enter
- **Fairness**: each thread waits for same amount of time
 - Also, threads acquire locks in the same order as requested
- **Performance**: CPU time is used efficiently

- Locks are variables in shared memory
 - Two main operations: **acquire()** and **release()**
 - Also called **lock()** and **unlock()**

- To check if locked,
 - read variable and check value
- To acquire,
 - write “locked” value to variable
 - Should only do this if already unlocked
 - If already locked, keep reading value until unlock observed
- To release,
 - write “unlocked” value to variable

Implementing as a straightforward data structure?

```
typedef struct mutex {  
    bool is_locked;           /* true if locked */  
    thread_id_t owner;        /* thread holding lock, if locked */  
    thread_list_t waiters;    /* threads waiting for lock */  
    lower_level_lock_t lk;    /* Protect above fields */  
};
```

★ Fine, so long as we avoid data races on the mutex itself

→ Need lower-level lock `lk` for mutual exclusion

◆ Internally, `mutex_*` functions bracket code with

`lock(&mutex->lk) . . . unlock(&mutex->lk)`

→ Otherwise, data races! (E.g., two threads manipulating `waiters`)

How to implement `lower_level_lock_t lk;`?

- Could use Peterson's algorithm,
 - typically a bad idea
 - too slow
 - and don't know maximum number of threads

Two approaches

1. Disable interrupts
 - a. works only in kernel
2. Spinlocks

Approach 1: Disable interrupts

Naïve use of Interrupt Enable/Disable

- How can we build multi-instruction atomic operations?
 - Recall: dispatcher gets control in two ways.
 - Internal: Thread does something to relinquish the CPU
 - External: Interrupts cause dispatcher to take CPU
 - On a *uniprocessor*, can avoid context-switching by:
 - Avoiding internal events (although virtual memory tricky)
 - Preventing external events by disabling interrupts

- Consequently, naïve Implementation of locks:

```
LockAcquire {  
    disable interrupts;  
}
```

```
LockRelease {  
    enable interrupts;  
}
```

- Problems with this approach:

- Can't let user do this!

```
LockAcquire () ;  
while (true) { ; }
```



- Real-Time system—no guarantees on timing!
 - Critical Sections might be arbitrarily long
- What happens with I/O or other important events?
 - “Reactor about to meltdown. Help?”

Better Implementation of Locks by Disabling Interrupts

- Key idea: maintain a lock variable and impose mutual exclusion only during operations on that variable

```
int value = FREE;
```



```
Acquire() {  
    disable interrupts;  
    if (value == BUSY) {  
        put thread on wait queue;  
        Go to sleep();  
        // Enable interrupts?  
    } else {  
        value = BUSY;  
    }  
    enable interrupts;  
}
```

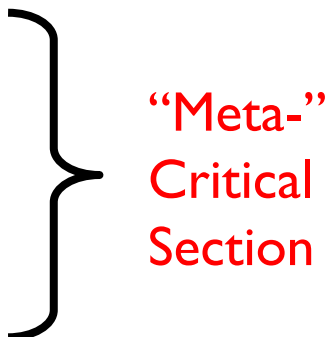
```
Release() {  
    disable interrupts;  
    if (anyone on wait queue) {  
        take thread off wait queue  
        Place on ready queue;  
    } else {  
        value = FREE;  
    }  
    enable interrupts;  
}
```

- Really only works in kernel – why?

Why do we need to disable interrupts at all?

- Avoid interruption between checking and setting lock value.
- Prevent switching to other thread that might be trying to acquire lock!
- Otherwise two threads could think that they both have lock!

```
Acquire() {  
    disable interrupts;  
    if (value == BUSY) {  
        put thread on wait queue;  
        Go to sleep();  
        // Enable interrupts?  
    } else {  
        value = BUSY;  
    }  
    enable interrupts;  
}
```



“Meta-”
Critical
Section

- Note: unlike previous solution, this “meta-”critical section is very short
 - User of lock can take as long as they like in their own critical section: doesn’t impact global machine behavior
 - Critical interrupts taken in time!

What about re-enabling ints when going to sleep?

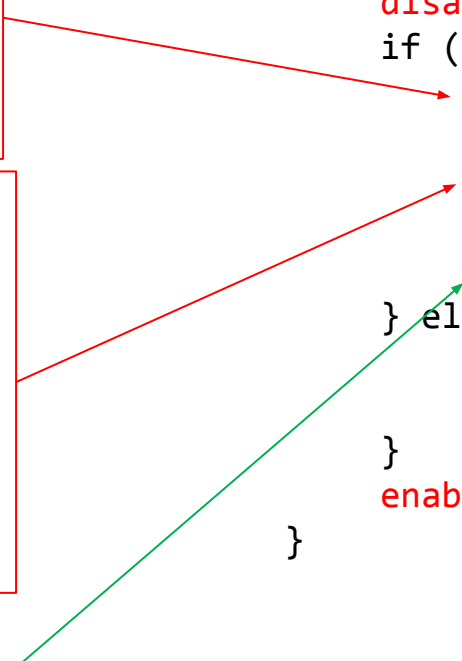
- Before putting thread on the wait queue?

- Release can check the queue and not wake up thread

- After putting the thread on the wait queue?

- Release puts the thread on the ready queue, but the thread still thinks it needs to go to sleep
- Misses wakeup and still holds lock (deadlock!)

```
Acquire() {  
    disable interrupts;  
    if (value == BUSY) {  
        put thread on wait queue;  
        Go to sleep();  
    } else {  
        value = BUSY;  
    }  
    enable interrupts;  
}
```

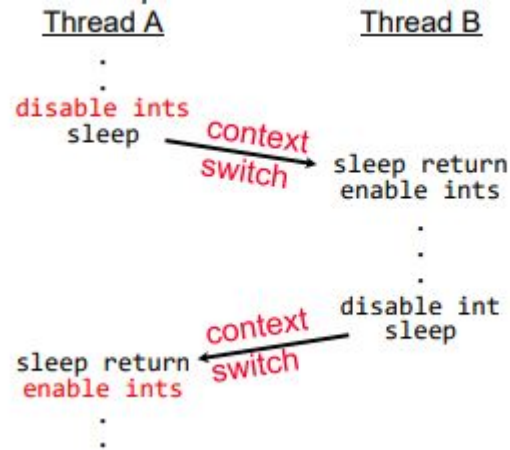


- Want to put it after sleep(). But – how?

How to Re-enable After Sleep()?

In scheduler, since interrupts are disabled when you call sleep:

- Responsibility of the next thread to re-enable ints
- When the sleeping thread wakes up, returns to acquire and re-enables interrupts



Problems with interrupt based locks

Can't give lock implementation to users

Doesn't work well on multiprocessor

- Disabling interrupts on all processors requires messages and would be very time consuming

But sometimes most efficient solution for uniprocessors

for apps with $n > 1$ threads (1 kthread)

Cannot take advantage of multiprocessors

But sometimes most efficient solution for uniprocessors

Typical setup:

- periodic timer signal caught by thread scheduler
- Have per-thread “do not interrupt” (DNI) bit

lock (lk):

- sets thread’s DNI bit

If timer interrupt arrives

- Check interrupted thread’s DNI bit
- If DNI clear,
 - preempt current thread
- If DNI set,
 - set “interrupted” (I) bit
 - & resume current thread

unlock (lk):

- clears DNI bit and checks (I) bit
- If I bit is set, immediately yields the CPU

Approach 2: Spinlocks

- Idea is to implement something like this:

```
bool lock = false; // shared variable
void acquire(bool *lock) {
    while (*lock) /* wait */
        ;
    *lock = true;
}
void release(bool *lock) { *lock = false; }
```

This does not work!

- Checking and writing of the lock value in acquire() need to happen atomically.

Spinlocks

Most CPUs support atomic **read-[modify-]write**

- **Test and Set**
- **Fetch and Add**
- **Compare and Swap (CAS)**
- **Load Linked / Store Conditional**

Hardware is responsible for implementing this correctly

Example: `int test_and_set (int *lockp) ;`

- atomically sets ***lockp = 1**
- and returns **old value**

Special instruction

- no way to implement in portable C99
- C11 supports with explicit `atomic_flag_test_and_set` function
- C11 [Atomic operations library](https://ericniebler.com/2014/05/27/atomic-operations-library/)

<https://www.scs.stanford.edu/24wi-cs212/notes/concurrency.pdf>

Synchronization on x86

x86 xchg instruction, exchanges reg with mem

_test_and_set :

```
movl 4(% esp), % edx # % edx = lockp
```

```
movl $1, % eax # % eax = 1
```

```
xchgl % eax, (% edx) # swap(% eax, *lockp)
```

```
ret
```

// Implementation in x86 :

```
int TAS(volatile int *addr, int newval) {  
    int result = newval;  
    asm volatile("lock; xchg %0, %1"  
                 : "+m" (*addr), "=r" (result)  
                 : "1" (newval)  
                 : "cc");  
    return result;  
}
```

CPU locks memory system around read and write

- xchgl always acts like it has implicit lock prefix
- Prevents other uses of the bus (e.g., DMA)

Usually runs at memory bus speed, not CPU speed

- Much slower than cached read/buffered write

recall: using in critical section problem

```
volatile int lock = 0;

void critical() {
    while (test_and_set(&lock) == 1); /*spinlock*/

    /* critical section */

    lock = 0; /* release lock when finished CS*
}
```

Use spinlocks to implement mutex's lower_level_lock_t

```
typedef struct mutex {  
    bool is_locked;  
    thread_id_t owner;  
    thread_list_t waiters;  
    lower_level_lock_t lk;  
};  
  
#define lock(lockp) while (test_and_set (lockp))  
#define trylock(lockp) (test_and_set (lockp) == 0)  
#define unlock(lockp) *lockp = 0
```

Can you use spinlocks instead of mutexes?

- Wastes CPU, especially if thread holding lock not running
- Mutex functions have short C.S., less likely to be preempted
- On multiprocessor, sometimes good to spin for a bit, then yield

Problem: Busy-Waiting for Lock



- Positives for this solution
 - Machine can receive interrupts
 - User code can use this lock
 - Works on a multiprocessor
- Negatives
 - This is very inefficient as thread will consume cycles waiting
 - Waiting thread may take cycles away from thread holding lock (no one wins!)
 - **Priority Inversion**: If busy-waiting thread has higher priority than thread holding lock \Rightarrow no progress!
- Priority Inversion problem with original Martian rover
- For higher-level synchronization primitives (e.g. semaphores or monitors), waiting thread may wait for an arbitrary long time!
 - Thus even if busy-waiting was OK for locks, definitely not ok for other primitives
 - Homework/exam solutions should avoid busy-waiting!

Kernel Synchronization

Should kernel use locks or disable interrupts?

Old UNIX had 1 CPU, non-preemptive threads, no mutexes

- Interface designed for single CPU, so count++ etc. not data race
- ... Unless memory shared with an interrupt handler

```
int x = splhigh (); /* bsd Disable interrupts, preempt_disable() in Linux */
/* touch data shared with interrupt handler ... */
splx (x); /* bsd Restore previous state, preempt_enable in Linux */
```

- Used arbitrary pointers like condition variables

```
int [t]sleep (void *ident, int priority, ...);
```

put thread to sleep; will wake up at priority (~cond_wait)

```
int wakeup (void *ident);
```

wake up all threads sleeping on ident (~cond_broadcast)

<https://www.scs.stanford.edu/24wi-cs212/notes/concurrency.pdf>

Kernel Locks

Nowadays, should design for multiprocessors

- Even if first version of OS is for uniprocessor
- Someday may want multiple CPUs and need preemptive threads
- That's why Pintos uses sleeping locks (sleeping locks means mutexes, as opposed to spinlocks)

Multiprocessor performance needs fine-grained locks

- Want to be able to call into the kernel on multiple CPUs

If kernel has locks, should it ever disable interrupts?

Kernel Locks

If kernel has locks, should it ever disable interrupts?

- Yes! Can't sleep in interrupt handler, so can't wait for lock
- So even modern OSes have support for disabling interrupts
- Often uses DNI trick when cheaper than masking interrupts in hardware

Improving spinlock performance

Kernel support for userspace
sleeping locks

Cache Coherence

Recall: Where are we going with synchronization?

Programs	Shared Programs
Higher-level API	Locks Semaphores Monitors Send/Receive
Hardware	Load/Store Disable Ints Test&Set Compare&Swap

- We are going to implement various higher-level synchronization primitives using atomic operations
 - Everything is pretty painful if only atomic primitives are load and store
 - Need to provide primitives useful at user-level

Evaluating our lock Implementation with TAS

```
typedef struct __lock_t {
    int flag;
} lock_t;

void init(lock_t *lock) { lock->flag = 0; }

void acquire(lock_t *lock) {
    while (test_and_set(&lock->flag, 1) == 1)
        ; // spin-wait (do nothing)
}

void release(lock_t *lock) { lock->flag = 0; }
```

Evaluating our lock Implementation with TAS

1) Mutual exclusion: only one thread in critical section at a time

2) Progress (deadlock-free): if several simultaneous requests, must allow one to proceed

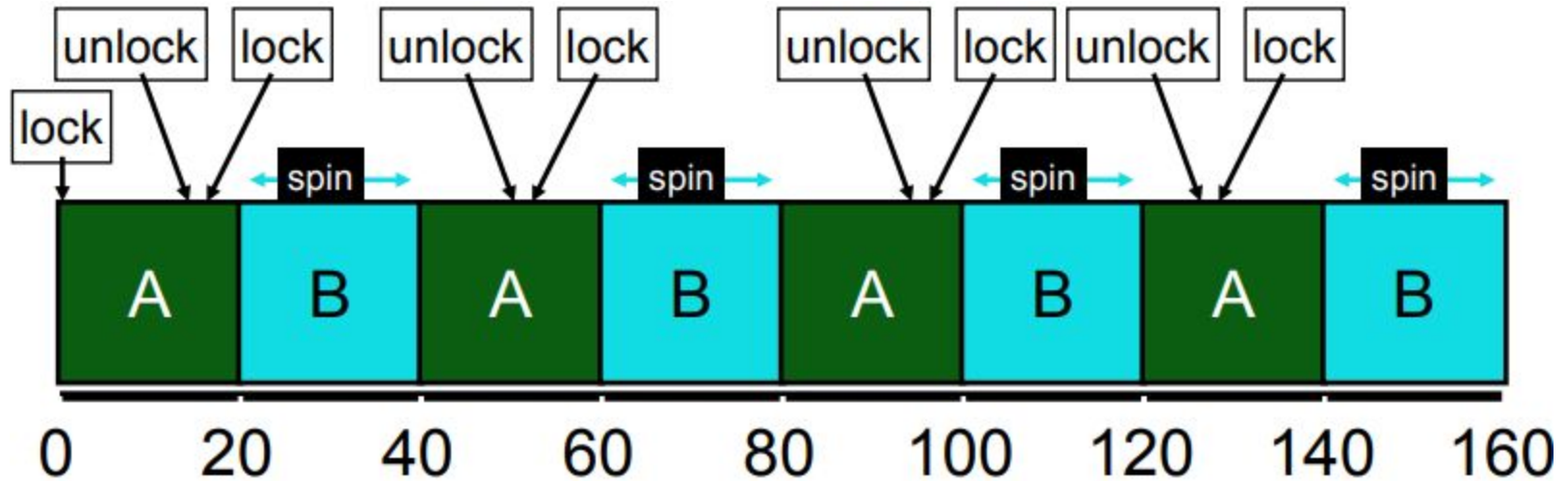
3) Bounded wait: must eventually allow each waiting thread to enter

4) Fairness: threads acquire lock in the order of requesting

5) Performance: CPU time is used efficiently

**3, 4, 5 may NOT be
satisfied in practice!**

our spinlock is not fair!



Scheduler is independent of locks/unlocks

- **Busy-Waiting:** thread consumes cycles while waiting
 - For multiprocessors: every test&set() is a write, which makes value ping-pong around in cache (using lots of network BW)

Fairness and Bounded Wait

Use Ticket Locks

Idea: reserve each thread's turn to use a lock.

- Each thread spins until their turn.

Use new atomic primitive: **fetch-and-add**

```
// Semantic
int fetch_and_add(int *ptr) {
    int old = *ptr;
    *ptr = old + 1;
    return old;
}

// example implementation
// GCC's built-in atomic function
__sync_fetch_and_add(ptr, 1)
```

ticket-lock implementation

```
typedef struct {
    int ticket;
    int turn;
} lock_t;

void lock_init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn = 0;
}

void acquire(lock_t *lock) {
    int myturn = fetch_and_add(&lock->ticket);
    while (lock->turn != myturn)
        ; // spin
}

void release(lock_t *lock) {
    lock->turn += 1;
}
```

Busy-waiting(spining) performance

Good when...

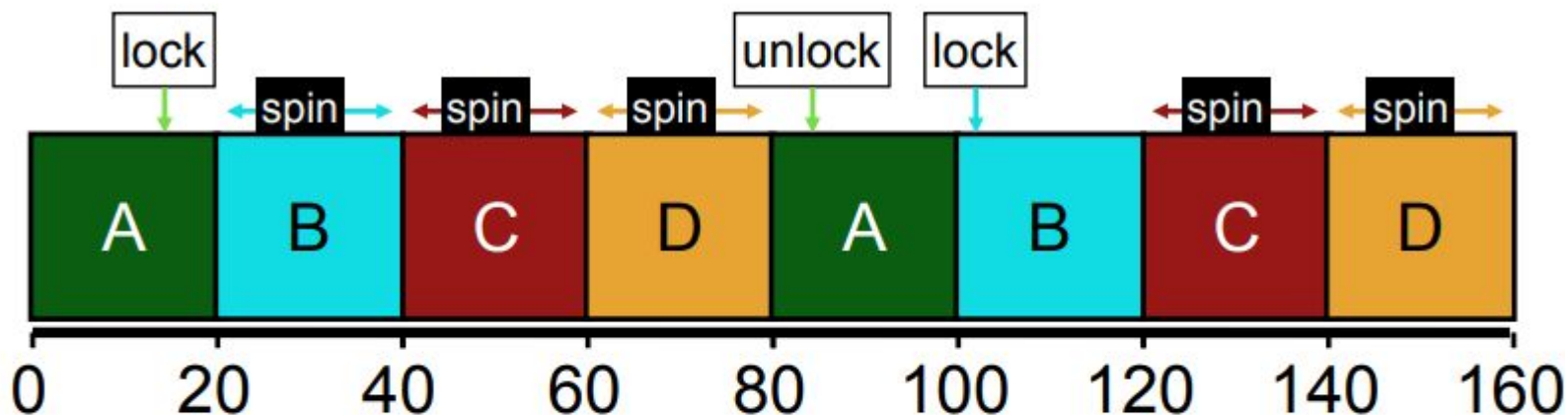
- many CPUs
- locks held a short time
- advantage: avoid context switch

Awful when...

- one CPU
- locks held a long time
- disadvantage: spinning is wasteful

CPU Scheduler Is Ignorant

busy-waiting (spinning) locks



CPU scheduler may run **B** instead of **A**
even though **B** is waiting for **A**

Ticket Lock with yield() (see Linus Torvalds comment)

```
typedef struct {
    int ticket;
    int turn;
} lock_t;

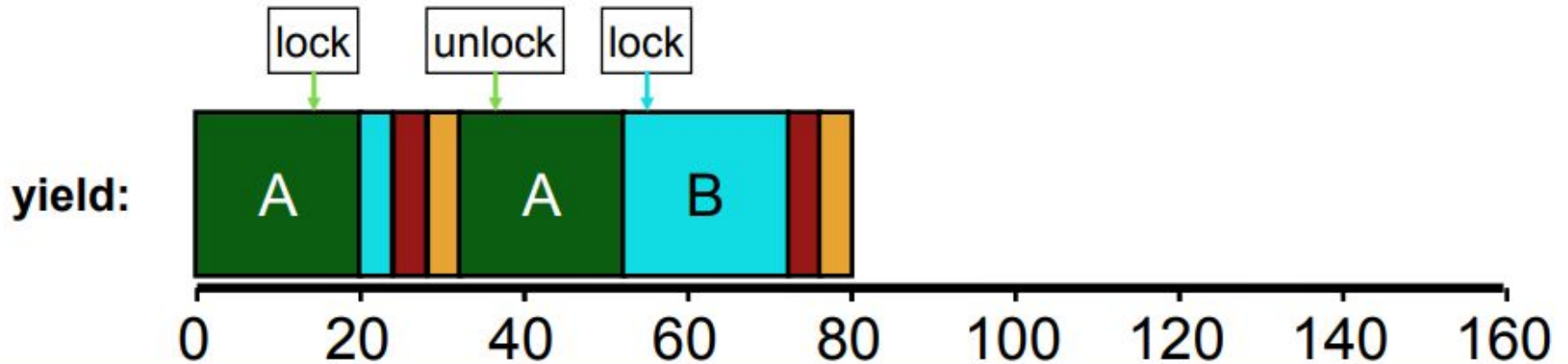
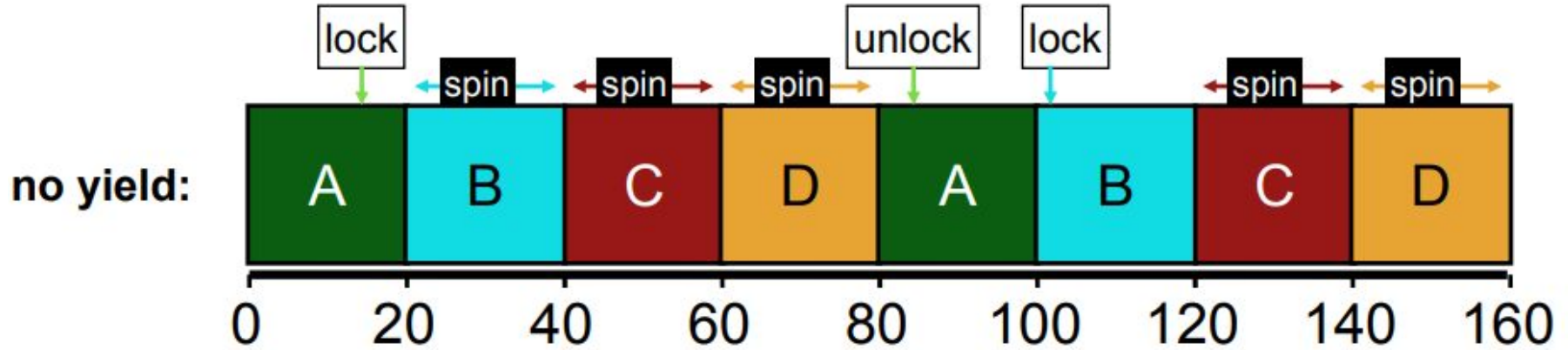
void acquire(lock_t *lock) {
    int myturn = fetch_and_add(&lock->ticket);
    while (lock->turn != myturn) sched_yield();
}

void release(lock_t *lock) { lock->turn += 1; }
```


yielding instead of spinning

Wasted time

- Without yield: $O(\text{threads} \times \text{time_slice})$
- With yield: $O(\text{threads} \times \text{context_switch_time})$



Evaluating Ticket Lock

5) Performance: CPU time is used efficiently

→ 5 (even with yielding, too much overhead)

So even with yield, spinning is slow with high thread contention

Next improvement: instead of spinning, block and put thread on a wait queue

Blocking Locks with queues

`acquire()` removes waiting threads from run queue using special system call

`release()` returns waiting threads to run queue using special system call

Better Locks using test&set

- Can we build test&set locks without busy-waiting?
 - Mostly. Idea: only busy-wait to atomically check lock value

```
int guard = 0; // Global Variable!  
int mylock = FREE; // Interface: acquire(&mylock);  
                        //                        release(&mylock);
```



```
acquire(int *thelock) {  
    // Short busy-wait time  
    while (test_and_set(guard));  
    if (*thelock == BUSY) {  
        put thread on wait queue;  
        go to sleep() & guard = 0;  
        // guard == 0 on wakeup!  
    } else {  
        *thelock = BUSY;  
        guard = 0;  
    }  
}
```

```
release(int *thelock) {  
    // Short busy-wait time  
    while (test_and_set(guard));  
    if anyone on wait queue {  
        take thread off wait queue  
        Place on ready queue;  
    } else {  
        *thelock = FREE;  
    }  
    guard = 0;
```

- Note: sleep has to be sure to reset the guard variable
 - Why can't we do it just before or just after the sleep?

Kernel support for sleeping locks

Sleeping locks must interact with scheduler

- For processes or kernel threads, must go into kernel (expensive)
- Common case is you can acquire lock—how to optimize?
- **Idea: never enter kernel for uncontested lock**

futex abstraction solves the problem

- Ask kernel to sleep only if memory location hasn't changed
- **void futex (int *uaddr, FUTEX_WAIT, int val. . .);**
 - Go to sleep only if ***uaddr == val**
 - Extra arguments allow timeouts, etc.
- **void futex (int *uaddr, FUTEX_WAKE, int val. . .);**
 - Wake up at most **val** threads sleeping on **uaddr**
- **uaddr** is translated down to offset in VM object
 - So works on memory mapped file at different virtual addresses in different processes

Recap: Locks using interrupts

```
int mylock=0;
```

```
acquire(&mylock);
```

```
...
```

```
critical section;
```

```
...
```

```
release(&mylock);
```

```
acquire(int *thelock) {  
    disable interrupts;  
}
```

```
release(int *thelock)  
{  
    enable interrupts;  
}
```

```
acquire(int *thelock) {  
    // Short busy-wait time  
    disable interrupts;  
    if (*thelock == 1) {  
        put thread on wait-queue;  
        go to sleep() //??  
    } else {  
        *thelock = 1;  
        enable interrupts;  
    }  
}
```

```
release(int *thelock) {  
    // Short busy-wait time  
    disable interrupts;  
    if anyone on wait queue {  
        take thread off wait-queue  
        Place on ready queue;  
    } else {  
        *thelock = 0;  
    }  
    enable interrupts;  
}
```

If one thread in critical section, no other activity (including OS) can run!

Lock argument not used!

Recap: Locks using test & set

```
int mylock=0;
acquire(&mylock);
...
critical section;
...
release(&mylock);
```

```
int mylock = 0;
acquire(int *thelock) {
    while(test&set(thelock));
}
```

```
release(int *thelock) {
    *thelock = 0;
}
```

Threads waiting to enter
critical section
busy-wait

```
int guard = 0; // global!
```

```
acquire(int *thelock) {
    // Short busy-wait time
    while(test&set(guard));
    if (*thelock == 1) {
        put thread on wait-queue;
        go to sleep() & guard = 0;
        // guard == 0 on wakeup
    } else {
        *thelock = 1;
        guard = 0;
    }
}
```

```
release(int *thelock) {
    // Short busy-wait time
    while (test&set(guard));
    if anyone on wait queue {
        take thread off wait-queue;
        Place on ready queue;
    } else {
        *thelock = 0;
    }
    guard = 0;
}
```


Linux futex: Fast Userspace Mutex

```
#include <linux/futex.h>
#include <sys/time.h>

int futex(int *uaddr, int futex_op, int val,
          const struct timespec *timeout );
```

uaddr points to a 32-bit value in user space

futex_op

- FUTEX_WAIT – if *val* == **uaddr* sleep till FUTEX_WAIT
 - **Atomic** check that condition still holds after we disable interrupts (in kernel!)
- FUTEX_WAKE – wake up at most *val* waiting threads
- FUTEX_FD, FUTEX_WAKE_OP, FUTEX_CMP_REQUEUE: More interesting operations!

timeout

- ptr to a *timespec* structure that specifies a timeout for the op

- Interface to the kernel `sleep()` functionality!
 - Let thread put themselves to sleep – conditionally!
- **futex is not exposed in libc; it is used within the implementation of pthreads**
 - Can be used to implement locks, semaphores, monitors, etc...

Example: First try: T&S and futex

```
int mylock = 0; // Interface: acquire(&mylock);  
                //                release(&mylock);
```

```
acquire(int *thelock) {  
    while (test&set(thelock)) {  
        futex(thelock, FUTEX_WAIT, 1);  
    }  
}
```

```
release(int *thelock) {  
    thelock = 0; // unlock  
    futex(&thelock, FUTEX_WAKE, 1);  
}
```

- Properties:
 - Sleep interface by using futex – no busywaiting
- No overhead to acquire lock
 - Good!
- Every unlock has to call kernel to potentially wake someone up – even if none
 - Doesn't quite give us no-kernel crossings when uncontended...!

Example: Try #2: T&S and futex

```
bool maybe_waiters = false;
int mylock = 0; // Interface:
acquire(&mylock, &maybe_waiters);
//
release(&mylock, &maybe_waiters);

acquire(int *thelock, bool *maybe) {
    while (test&set(thelock)) {
        // Sleep, since lock busy!
        *maybe = true;
        futex(thelock, FUTEX_WAIT, 1);

        // Make sure other sleepers not stuck
        *maybe = true;
    }
}
```

```
release(int *thelock, bool *maybe) {
    thelock = 0;
    if (*maybe) {
        *maybe = false;
        // Try to wake up someone
        futex(&value, FUTEX_WAKE, 1);
    }
}
```

- This is syscall-free in the uncontended case
 - Temporarily falls back to syscalls if multiple waiters, or concurrent acquire/release
- But it can be considerably optimized!
 - See “[Futexes are Tricky](https://www.eecs.berkeley.edu/~cs162/sp22/static/lectures/8.pdf)” by Ulrich Drepper

Try #3: Better, using more atomics

- Much better: Three (3) states:
 - **UNLOCKED**: No one has lock
 - **LOCKED**: One thread has lock
 - **CONTESTED**: Possibly more than one (with someone sleeping)
- Clean interface!
- Lock grabbed cleanly by either
 - **compare_and_swap()**
 - First **swap()**
- No overhead if uncontested!
- Could build semaphores in a similar way!

no one has to be woken up, and so no syscalls at all.

```
typedef enum { UNLOCKED, LOCKED, CONTESTED } Lock;
Lock mylock = UNLOCKED; // Interface:
acquire(&mylock);
//
release(&mylock);

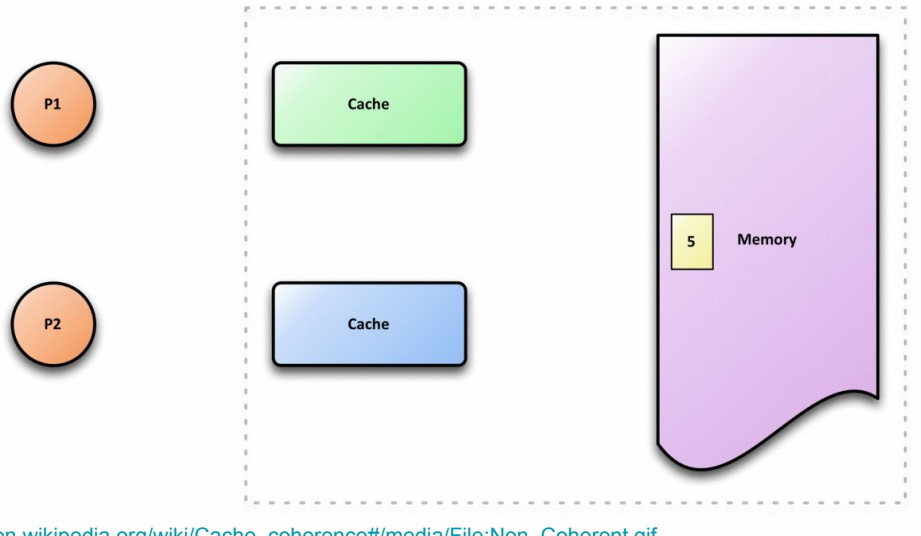
acquire(Lock *thelock) {
    // If unlocked, grab lock!
    if (compare&swap(thelock, UNLOCKED, LOCKED))
        return;

    // Keep trying to grab lock, sleep in futex
    while (swap(mylock, CONTESTED) != UNLOCKED)
        // Sleep unless someone releases heard!
        futex(thelock, FUTEX_WAIT, CONTESTED);
}

release(Lock *thelock) {
    // If someone sleeping,
    if (swap(thelock, UNLOCKED) == CONTESTED)
        futex(thelock, FUTEX_WAKE, 1);
}
```

- See also [futex_demo.c](#) in [futex\(2\) - Linux manual page](#)

Cache coherence



<https://www.scs.stanford.edu/23wi-cs212/notes/synchronization1.pdf>

For detailed lecture see:

https://booksite.elsevier.com/9780123973375/powerpoint/chapter_07.ppt

https://en.wikipedia.org/wiki/Cache_coherence#/media/File:Non_Coherent.gif

For the remaining slides, enough to know “concept” (what it is used for), you are NOT responsible from the details for the exam.

Important memory system properties

Coherence – concerns accesses to a single memory location

- There is a total order on all updates
- Must obey program order if access from only one CPU
- There is bounded latency before everyone sees a write

Consistency – concerns ordering across memory locations

- Even with coherence, different CPUs can see the same write happen at different times
- Sequential consistency is what matches our intuition

(As if operations from all CPUs interleaved on one CPU)

- Many architectures offer weaker consistency
- Yet well-defined weaker consistency can still be sufficient to implement thread API

Multicore cache coherence

Performance requires caches

- Divided into chunks of bytes called lines (e.g., 64 bytes)
- Caches create an opportunity for cores to disagree about memory

Bus-based approaches

- “Snoopy” protocols, each CPU listens to memory bus
- Use write-through and invalidate when you see a write bits
- Bus-based schemes limit scalability

Modern CPUs use networks (e.g., hypertransport, infinity fabric, QPI, UPI)

- CPUs pass each other messages about cache lines

MESI coherence protocol

Modified (M)

Exactly one cache has a valid copy

The copy in the current cache is *dirty* - (needs to be written back to memory)

Must invalidate all copies in other caches before entering this state

Exclusive (E)

Same as modified except the copy in the current cache is *clean* (it matches main memory).

Shared (S)

One or more caches and memory have a valid copy

Invalid (I)

Indicates that this cache line is invalid (**unused**).

Owned (for enhanced “MOESI” protocol)

has exclusive right to change, others can read but not write

For any given pair of caches, the permitted states of a given cache line are as follows:

	M	E	S	I
M	✗	✗	✗	✓
E	✗	✗	✗	✓
S	✗	✗	✓	✓
I	✓	✓	✓	✓

Core and Bus Actions

Actions performed by CPU core

- Read
- Write
- Evict (modified? must write back)

Transactions on bus (or interconnect)

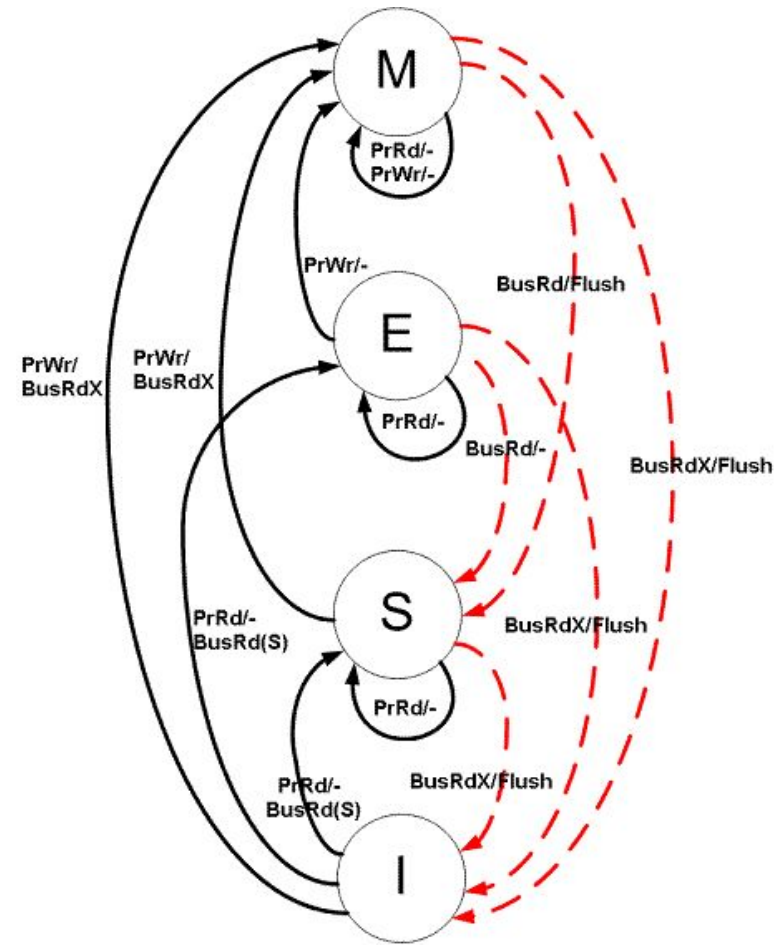
- Read: without intent to modify, data can come from memory or another cache
- Read-exclusive: with intent to modify, must invalidate all other

cache copies

- Writeback: contents put on bus and memory is updated

State diagram for MESI

1. PrRd: The processor requests to **read** a Cache block.
2. PrWr: The processor requests to **write** a Cache block
3. BusRd: Snooped request that indicates there is a **read** request to a Cache block requested by another processor
4. BusRdX: Snooped request that indicates there is a **write** request to a Cache block requested by another processor that **doesn't already have the block**.
5. BusUpgr: Snooped request that indicates that there is a write request to a Cache block requested by another processor that already has that **cache block residing in its own cache**.
6. Flush: Snooped request that indicates that an entire cache block is written back to the main memory by another processor.
7. FlushOpt: Snooped request that indicates that an entire cache block is posted on the bus in order to supply it to another processor (Cache to Cache transfers).



https://en.wikipedia.org/wiki/MESI_protocol

cc-NUMA = cache-coherent NUMA

Old machines used dance hall architectures

- Any CPU can “dance with” any memory equally

An alternative: Non-Uniform Memory Access (NUMA)

- Each CPU has fast access to some “close” memory
- Slower to access memory that is farther away
- Use a directory to keep track of who is caching what

Originally for esoteric machines with many CPUs

- But AMD and then intel integrated memory controller into CPU
- Faster to access memory controlled by the local socket (or even local die in a multi-chip module)

cc-NUMA = cache-coherent NUMA

- Rarely see non-cache-coherent NUMA (BBN Butterfly 1, Cray T3D)

Real World Coherence Costs

See [David] for a great reference. Xeon results:

- 3 cycle L1, 11 cycle L2, 44 cycle LLC, 355 cycle local RAM

If another core in same socket holds line in modified state:

- load: 109 cycles (LLC + 65)
- store: 115 cycles (LLC + 71)
- atomic CAS: 120 cycles (LLC + 76)

LLC: non-inclusive last-level cache

If a core in a different socket holds line in modified state:

- NUMA load: 289 cycles
- NUMA store: 320 cycles
- NUMA atomic CAS: 324 cycles

But only a partial picture

- Could be faster because of out-of-order execution
- Could be slower if interconnect contention or multiple hops

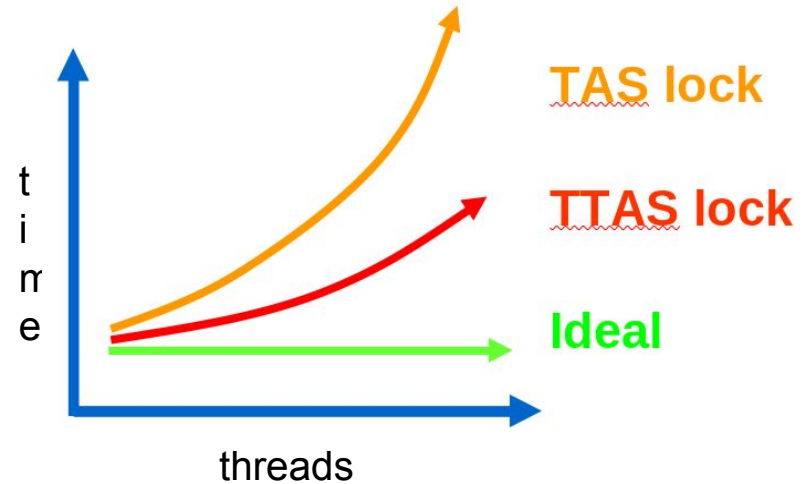
NUMA and spinlocks

Test-and-set (TAS) spinlock advantages

- Simple to implement and understand
- One memory location for arbitrarily many CPUs

Test-and-set (TAS) spinlock Disadvantages

- Lots of traffic over memory interconnect (especially w. > 1 spinner)
- Not necessarily fair (lacks bounded waiting)
- Even less fair on a NUMA machine



Better alternative: Test-and-test-and-set Lock

- TTAS performs much better than TAS
- Neither approaches ideal

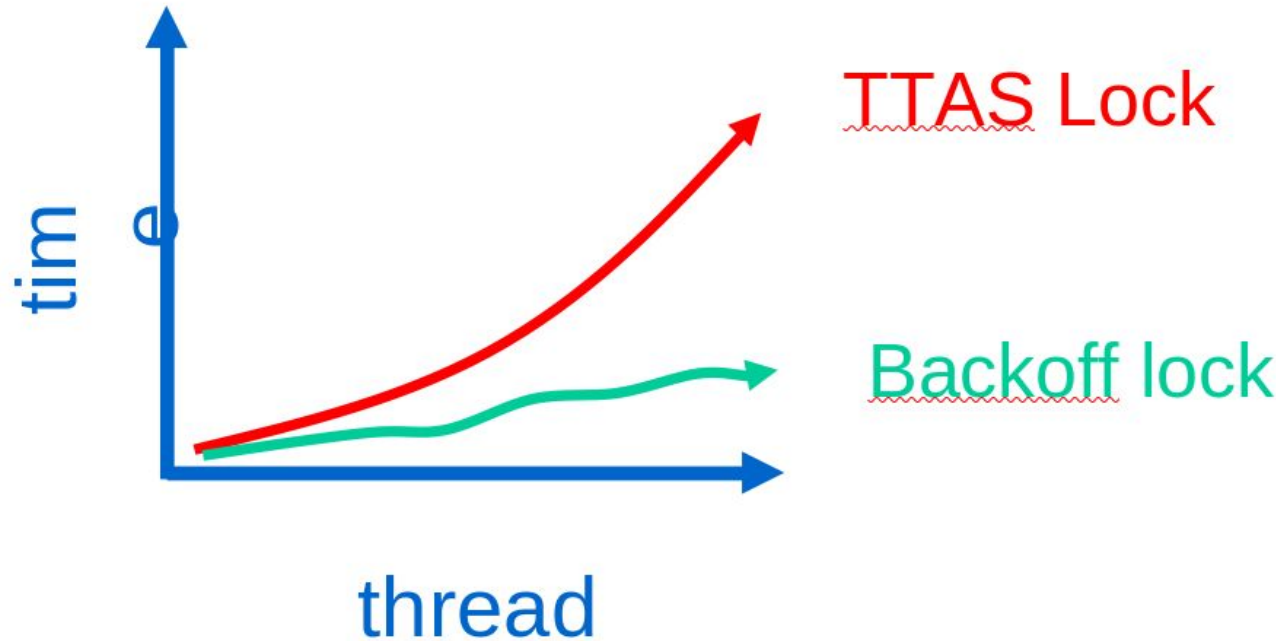
introduce waiting: Backoff lock

Good

- Easy to implement
- Beats TTAS lock

Bad

- Almost same
- Must choose parameters carefully
- Not portable across platform



NUMA and spinlocks (More Advanced Topics)

- **Idea 1:** Avoid spinlocks altogether (lock free data structures)
- **Idea 2:** Reduce interconnect traffic with better spinlocks
 - ◆ Design lock that spins only on local memory
 - ◆ Also gives better fairness

For the remaining slides, enough to know “concept” (what it is used for), you are NOT responsible from the details for the exam.

Useful macros

https://en.cppreference.com/w/c/atomic/memory_order

Atomic compare and swap: CAS (mem, old, new)

- If `*mem == old`,
 - then swap `*mem ↔ new`
 - and return true,
- else false
- On x86, can implement using locked `cmpxchg` instruction
- In C11, use `atomic_compare_exchange_strong`

(note: C atomics version sets `old = *mem` if `*mem != old`)

Atomic swap: XCHG (mem, new)

- Atomically exchanges `*mem ↔ new`
- Implement w. C11 `atomic_exchange`, or `xchg` on x86

Atomic fetch and add: FADD (mem, val)

- Atomically sets `*mem += val` and returns old value of `*mem`
- Implement w. C11 `atomic_fetch_add`, lock add on x86

Atomic fetch and subtract: FSUB (mem, val)

Note: atomics return previous value (like `x++`, not `++x`)

All behave like sequentially consistent fences

- In C11, weaker `_explicit` versions take a `memory_order` argument

MCS Lock

Build a better spinlock

- Lock designed by [Mellor-Crummey and Scott](#)

- Goal:

- reduce bus traffic on cc machines,
- improve fairness

Each CPU has a qnode structure in local memory

```
typedef struct qnode {  
    __Atomic(struct qnode *) next;  
    atomic_bool locked;  
} qnode;
```

- Local can mean local memory in NUMA machine
- Or just its own cache line that gets cached in exclusive mode

While waiting, spin on your local `locked` flag

A lock is a qnode pointer: `typedef __Atomic (qnode *) lock;`

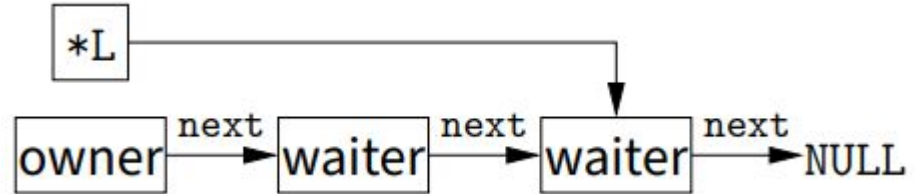
- Construct list of CPUs holding or waiting for lock
- lock itself points to tail of list (or NULL when unlocked)

see also pg.151 in https://booksite.elsevier.com/9780123973375/powerpoint/chapter_07.ppt, there are also clh lock etc.

MCS acquire

- If unlocked, L is NULL
- If locked, no waiters, L is owner's qnode
- If waiters, *L is tail of waiter list:

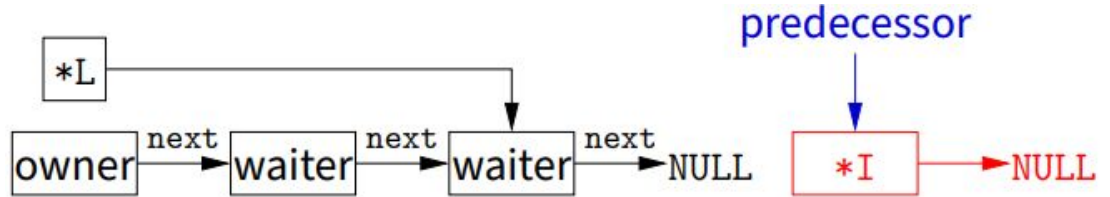
```
acquire(lock *L, qnode *I) {  
    I->next = NULL;  
    qnode *predecessor = I;  
    XCHG(*L, predecessor);  
    if (predecessor != NULL) {  
        I->locked = true;  
        predecessor->next = I;  
        while (I->locked)  
            ;  
    }  
}
```



MCS acquire

- If unlocked, L is NULL
- If locked, no waiters, L is owner's qnode
- If waiters, *L is tail of waiter list:

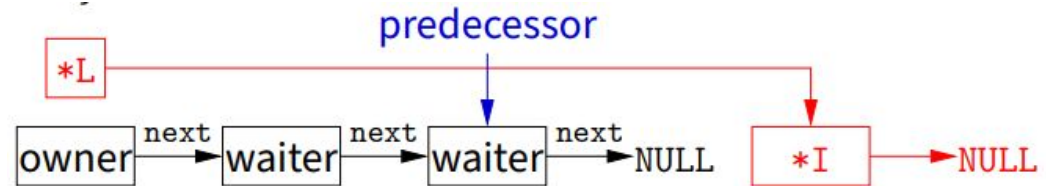
```
acquire(lock *L, qnode *I) {  
    I->next = NULL;  
    qnode *predecessor = I;  
    XCHG(*L, predecessor);  
    if (predecessor != NULL) {  
        I->locked = true;  
        predecessor->next = I;  
        while (I->locked)  
            ;  
    }  
}
```



MCS acquire

- If unlocked, L is NULL
- If locked, no waiters, L is owner's qnode
- If waiters, *L is tail of waiter list:

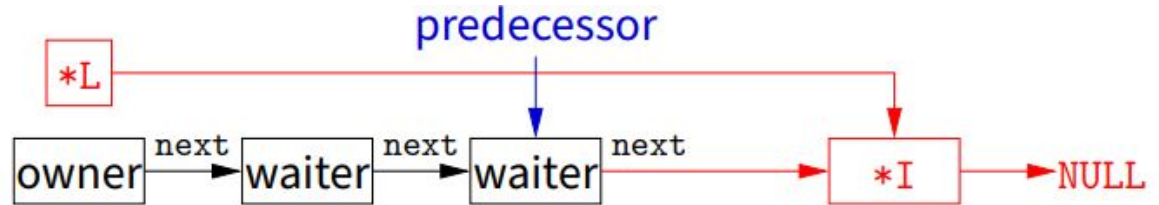
```
acquire(lock *L, qnode *I) {  
    I->next = NULL;  
    qnode *predecessor = I;  
    XCHG(*L, predecessor);  
    if (predecessor != NULL) {  
        I->locked = true;  
        predecessor->next = I;  
        while (I->locked)  
            ;  
    }  
}
```



MCS acquire

- If unlocked, L is NULL
- If locked, no waiters, L is owner's qnode
- If waiters, *L is tail of waiter list:

```
acquire(lock *L, qnode *I) {  
    I->next = NULL;  
    qnode *predecessor = I;  
    XCHG(*L, predecessor);  
    if (predecessor != NULL) {  
        I->locked = true;  
        predecessor->next = I;  
        while (I->locked)  
            ;  
    }  
}
```

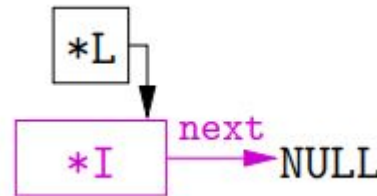


MCS Release with CAS

```
release(lock *L, qnode *I) {  
    if (!I->next)  
        if (CAS(*L, I, NULL)) return;  
    while (!I->next)  
        ;  
    I->next->locked = false;  
}
```

If $I \rightarrow \text{next}$ NULL and $*L == I$

- No one else is waiting for lock, OK to set $*L = \text{NULL}$

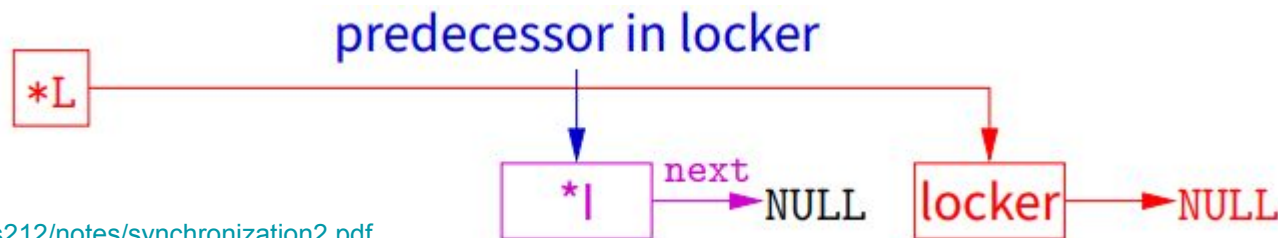


MCS Release with CAS

```
release(lock *L, qnode *I) {  
    if (!I->next)  
        if (CAS(*L, I, NULL)) return;  
    while (!I->next)  
        ;  
    I->next->locked = false;  
}
```

If $I \rightarrow \text{next}$ NULL and $*L \neq I$

- Another thread is in the middle of **acquire**
- Just wait for $I \rightarrow \text{next}$ to be **non-NULL**

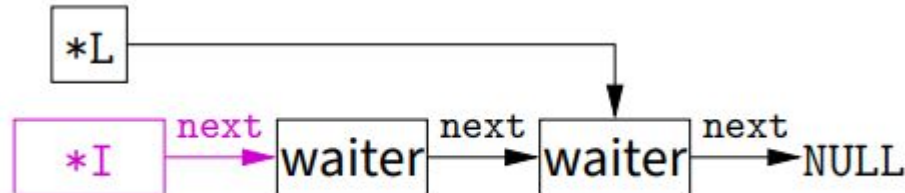


MCS Release with CAS

```
release(lock *L, qnode *I) {  
    if (!I->next)  
        if (CAS(*L, I, NULL)) return;  
    while (!I->next)  
        ;  
    I->next->locked = false;  
}
```

If $I \rightarrow \text{next}$ is non-NULL

- $I \rightarrow \text{next}$ oldest waiter, wake up with $I \rightarrow \text{next} \rightarrow \text{locked} = \text{false}$



MCS Release without CAS

```
release(lock *L, qnode *I) {  
    if (I->next)  
        I->next->locked = false;  
    else {  
        qnode *old_tail = NULL;  
        XCHG(*L, old_tail);  
        if (old_tail == I) return;  
        /* old_tail != I? CAS would have failed, so undo XCHG */  
        qnode *userper = old_tail;  
        XCHG(*L, userper);  
        while (I->next == NULL)  
            ;  
        if (userper) /* someone changed *L between 2 XCHGs */  
            userper->next = I->next;  
        else  
            I->next->locked = false;  
    }  
}
```

1. Atomically swap NULL into *L
 - If old value of *L was I, no waiters and we are done
2. Atomically swap old *L value back into *L
 - If *L unchanged, same effect as CAS

Otherwise,

- Some “userper” attempted to acquire lock between 1 and 2
- Because *L was NULL, the userper succeeded (May be followed by zero or more waiters)
- Graft old list of waiters on to end of new last waiter (Sacrifice small amount of fairness, but still safe)

Lock “Free” Multithreading: Non-blocking synchronization

Atomic operations

Read-modify-write (RMW) atomic instructions

Memory barriers (see [memory-barriers.txt](#))

In Linux kernel (mb(), smp_mb(), etc.),

assembly instruction asm volatile ("mfence" : :
: "memory")

C11 atomic library, Linux system calls

RCU

For the remaining slides, enough to know “concept” (what it is used for), you are NOT responsible from the details for the exam.

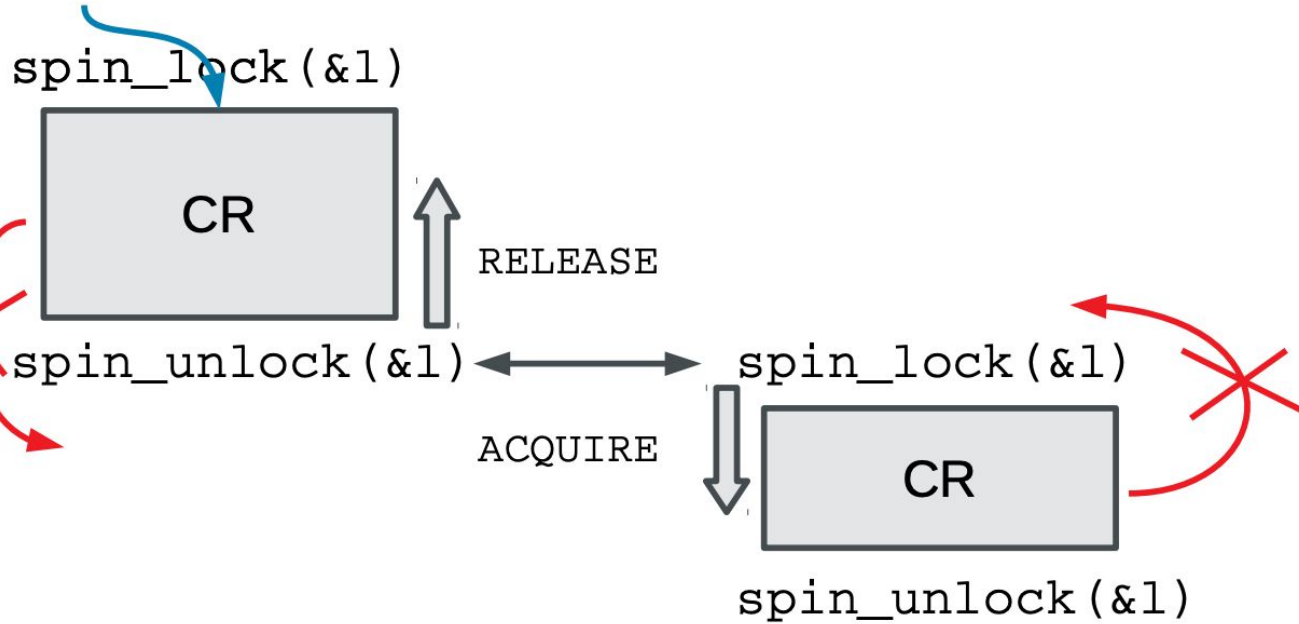
Acquire/release semantics

Passing information reliably between threads about a variable.

- Ideal in producer/consumer type situations (pairing!!).
- After an ACQUIRE on a given variable, all memory accesses preceding any prior RELEASE on that same variable are guaranteed to be visible.
- All accesses of all previous critical sections for that variable are guaranteed to have completed.
- C++11's **memory_order_acquire**, **memory_order_release** and **memory_order_relaxed** (see [Memory barriers in C](#)).

CPU0

CPU1



`smp_store_release(lock->val, 0) <-> cmpxchg_acquire(lock->val, 0, LOCKED)`

Recall: Producer/Consumer

```
/* PRODUCER */
for (;;) {
    item *nextProduced = produce_item();
    mutex_lock(&mutex);
    while (count == BUF_SIZE)
        cond_wait(&nonfull, &mutex);
    buffer[in] = nextProduced;
    in = (in + 1) % BUF_SIZE;
    count++;
    cond_signal(&nonempty);
    mutex_unlock(&mutex);
}
```

```
/* CONSUMER */
for (;;) {
    mutex_lock(&mutex);
    while (count == 0)
        cond_wait(&nonempty, &mutex);
    nextConsumed = buffer[out];
    out = (out + 1) % BUF_SIZE;
    count--;
    cond_signal(&nonfull);
    mutex_unlock(&mutex);
    consume_item(nextConsumed);
}
```

Eliminating Locks

One use of locks is to coordinate multiple updates of single piece of state

How to remove locks here?

- - Factor state so that each variable only has a single writer

In Producer/Consumer example, Assume one producer, one consumer

Why do we need count variable, written by both?

- To detect buffer full/empty

Have producer write in, consumer write out (both `_Atomic`)

- Use in/out to detect buffer state
- But note next example busy-waits, which is less good

Lock-free producer/consumer

```
atomic_int in, out;

void producer(void *ignored) {
    for (;;) {
        item *nextProduced = produce_item();
        while (((in + 1) % BUF_SIZE) == out)
            thread_yield();
        buffer[in] = nextProduced;
        in = (in + 1) % BUF_SIZE;
    }
}

void consumer(void *ignored) {
    for (;;) {
        while (in == out)
            thread_yield();
        nextConsumed = buffer[out];
        out = (out + 1) % BUF_SIZE;
        consume_item(nextConsumed);
    }
}
```

Note fences not needed because no relaxed atomics
example busy-waits, which is less good

Version with relaxed atomics

```
void producer(void *ignored) {
    for (;;) {
        item *nextProduced = produce_item();
        int slot = atomic_load_explicit(&in, memory_order_relaxed);
        int next = (slot + 1) % BUF_SIZE;
        while (atomic_load_explicit(&out, memory_order_acquire) ==
               next) // Could you use relaxed? ^^^^^^^
            thread_yield();
        buffer[slot] = nextProduced;
        atomic_store_explicit(&in, next, memory_order_release);
    }
}

void consumer(void *ignored) {
    // Use memory_order_acquire to load in (for latest buffer[myin])
    // Use memory_order_release to store out
}
```

Non-blocking synchronization

Design algorithm to avoid critical sections

- Any threads can make progress if other threads are preempted
- Which wouldn't be the case if preempted thread held a lock

Requires that hardware provide the right kind of atomics

- Simple test-and-set is insufficient
- Atomic compare and swap is good:
 - CAS (mem, old, new) If *mem == old, then swap *mem \longleftrightarrow new and return true, else false

Can implement many common data structures

- Stacks, queues, even hash tables

Can implement any algorithm on right hardware

- Need operation such as atomic compare and swap (has property called consensus number = ∞ [[Herlihy](#)])
- Entire kernels have been written without locks [[Greenwald](#)]

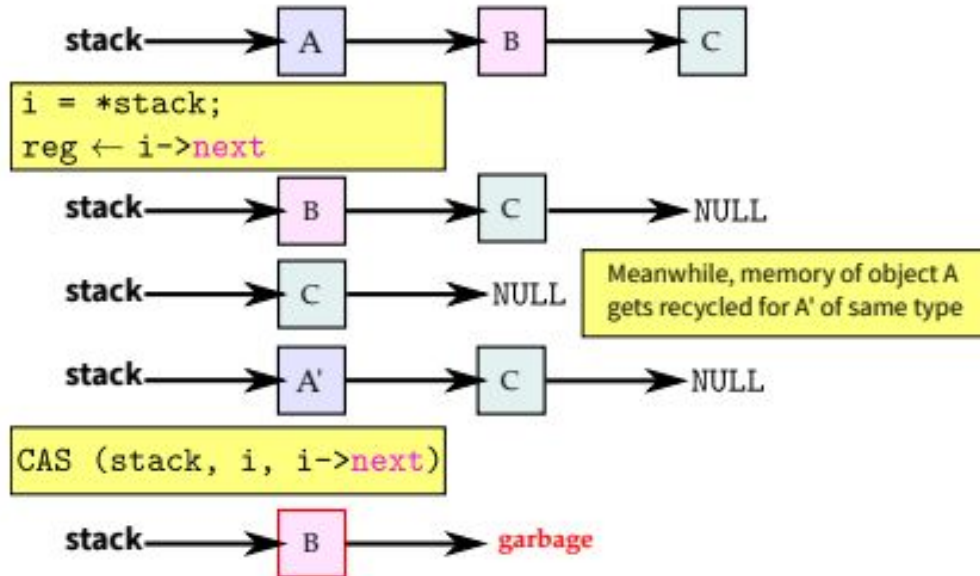
Example non-blocking stack

```
struct item {
    /* data */
    __Atomic(struct item *) next;
};

typedef __Atomic(struct item *) stack_t;
void atomic_push(stack_t *stack, item *i) {
    do {
        i->next = *stack;
    } while (!CAS(stack, i->next, i));
}
```

```
item *atomic_pop(stack_t *stack) {
    item *i;
    do {
        i = *stack;
    } while (!CAS(stack, i, i->next));
    return i;
}
```

Wait-free stack issues



“ABA” race in pop if other thread pops, re-pushes i

- - Can be solved by
 - [counters](#) (Version Counters)
 - or hazard pointers to delay re-use

Wait-free stack issues: ABA problem

Time: T1 (Thread 1)

T2 (Thread 2)

t0: Reads value A from *ptr
(gets distracted/suspended)

t1: Reads value A from *ptr
t2: Changes *ptr from A \rightarrow B
t3: Does some operations...
t4: Changes *ptr back from B \rightarrow A

t5: Wakes up, checks: "Is *ptr still A?"
YES! It's A again!
CAS succeeds: *ptr = new_value

PROBLEM: The state has changed $A \rightarrow B \rightarrow A$, but CAS thinks nothing changed!

Initial: Stack: $A \rightarrow B \rightarrow C$

Thread 1 (T1):

1. Reads `old_top = A`

2. Gets `new_top = B`

3. (Gets preempted)

4. CAS check:

"Is `*stack == A`?" YES!

Updates `*stack = B`

Thread 2 (T2):

1. Pops A successfully

2. Pops B successfully

3. Pushes A back!

Stack: $B \rightarrow C$

Stack: C

Stack: $A \rightarrow C$

(A is back on top!)

BUG: B was already popped! Stack now points to B, but B might be freed/reused!

The same memory address gets reused for different data: Node A is freed, New allocation returns same address

CAS succeeds, but points to wrong object

“Benign” races

Could also eliminate locks by having race conditions

- Maybe you think you care more about speed than correctness
 - ++hits; /* each time someone accesses web site */
- Maybe you think you can get away with the race
 - not really: <https://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html>

```
// Web server hit counter
int hits = 0;

void handle_request() {
    // "It's just a counter, wrong count is okay"
    hits++; // DATA RACE - multiple threads increment
}
```

In C/C++, Data Race = **Undefined Behavior**

“Benign” races

UB means anything can happen:

- Wrong values (what you expected)
- Crashes
- Infinite loops
- Security vulnerabilities
- Different behavior on different compilers/CPU

```
// Another "benign" race
int initialized = 0;
Data* data = NULL;

void init() {
    if (!initialized) {           // Read (race!)
        data = malloc(sizeof(Data));
        initialize_data(data);
        initialized = 1;         // Write (race!)
    }
    use(data);
}
```

What can happen:

1. Compiler reordering: May move `initialized = 1` before `initialize_data(data)`
2. CPU reordering: May execute store to `initialized` before store to `data`
3. Result: Other thread sees `initialized == 1` but `data == NULL` → CRASH

“Benign” races

```
if (!initialized) {                                // First check (UNSYNCHRONIZED)
    lock(m);                                        // Acquire lock
    if (!initialized) {                            // Second check (INSIDE LOCK)
        initialize();                             // Do the expensive initialization
        atomic_thread_fence(memory_order_release); // <-- CRITICAL!
        initialized = 1;                          // Mark as initialized
    }
    unlock(m);                                     // Release lock
}
```

But don't do this [Vyukov], [Boehm]! Not benign at all

- Again, UB really bad! Like user-after free or array overflow bad
- **If needed for efficiency, use relaxed-memory-order atomics**

<https://www.scs.stanford.edu/23wi-cs212/notes/synchronization1.pdf>

```
// BROKEN - "It looks like it should work"
Singleton* get_instance() {
    if (instance == NULL) {                // First check (unsynchronized)
        lock(mutex);
        if (instance == NULL) {            // Second check (synchronized)
            instance = new Singleton();
        }
        unlock(mutex);
    }
    return instance;
}
```

The race:

1. Thread 1: Enters, sees `instance == NULL`
2. Thread 1: Allocates memory for Singleton
3. Thread 1: Starts constructing Singleton (but not finished!)
4. Thread 2: Sees `instance != NULL` (pointer assigned but object not fully constructed)
5. Thread 2: Uses half-constructed Singleton → CRASH

This was such a common bug it has its own Wikipedia page: [Double-checked locking - Wikipedia](#)

Read Copy Update(RCU)

- achieves scalability improvements by allowing reads to occur concurrently with updates.
- supports concurrency between
 - a single updater
 - and multiple readers.
-

```
// READERS proceed without locks
// READ data (no lock needed!)

// WRITER:
1. Make a COPY of the data
2. UPDATE the copy
3. ATOMICALLY replace the pointer to new version
4. Wait for all existing readers to finish
5. Free the old version
```

- **RCU thinking:** "Let readers proceed with old data while writer prepares new data. Switch atomically. Clean up old data later."

[McKenney](#),

see also slides [Read-Copy Update \(RCU\)](#),

[What is RCU, Fundamentally? \[LWN.net\]](#)

[What is RCU? -- "Read, Copy, Update" — The Linux Kernel documentation](#)

RCU wikipedia analogy

Wikipedia articles:

Traditional Locking:

while editing

- Article locked
- No one can read during edits
- Readers wait for editor to finish

RCU Approach:

- Editor works on a copy of the article
- Readers continue reading the old version
- When editor finishes: swap links to new version
- Old readers finish reading old version
- Eventually delete old version

No reader is ever blocked!

Read Copy Update(RCU)

The basic idea behind RCU is to split updates into "removal" and "reclamation" phases

- **the removal phase runs concurrently with readers**

- The typical **RCU update** sequence:
 - Remove pointers to a data structure, so that subsequent readers cannot gain a reference to it.
 - **Wait for all previous readers to complete their RCU read-side critical sections (lightweight synchronization).**
 - **we can separate reclamation phase into another thread**
 - At this point, there cannot be any readers who hold references to the data structure, so it now may safely be reclaimed (e.g., `kfree()`).

[McKenney](#),

see also slides [Read-Copy Update \(RCU\)](#),

[What is RCU, Fundamentally? \[LWN.net\]](#)

[What is RCU? -- "Read, Copy, Update" — The Linux Kernel documentation](#)

Read-copy update

Some data is read way more often than written

- Routing tables consulted for each forwarded packet
- Data maps in system with 100+ disks (updated on disk failure)

Optimize for the common case of reading without lock

- E.g., global variable: `_Atomic(routing_table *) rt;`
- use without lock

```
#define RELAXED(var) atomic_load_explicit(&(var), memory_order_relaxed)
/* ... */
route = lookup(RELAXED(rt), destination);
```

Update by making copy, swapping pointer

```
/* update mutex held here, serializing updates */
routing_table *newrt = copy_routing_table(rt);
update_routing_table(newrt);
atomic_store_explicit(&rt, newrt, memory_order_release);
```

How RCU works?

```
// Global pointer to configuration
config_t *global_config = NULL;

// Initialization
global_config = create_default_config();
```

```
// Reader thread (can be MANY of these concurrently)
void process_request() {
    // 1. Get pointer to current config (no synchronization!)
    config_t *cfg = rcu_dereference(global_config);

    // 2. Use the config
    int timeout = cfg->timeout;
    int retries = cfg->retries;
    // ...

    // 3. Done - no unlock needed!
    rcu_read_unlock();
}
```

```
// Writer thread (only ONE at a time)
void update_config(int new_timeout) {
    // 1. CREATE A COPY
    config_t *new_cfg = kmalloc(sizeof(config_t));
    memcpy(new_cfg, global_config, sizeof(config_t));

    // 2. UPDATE THE COPY
    new_cfg->timeout = new_timeout;

    // 3. ATOMICALLY SWAP POINTERS
    rcu_assign_pointer(global_config, new_cfg);

    // 4. SYNCHRONIZE (wait for existing readers)
    synchronize_rcu();

    // 5. FREE OLD VERSION
    kfree(old_cfg);
}
```


Time: Writer

Readers

t0: Create new version
 Update pointer

Reader A: sees OLD pointer
Reader B: sees NEW pointer
Reader C: sees NEW pointer

t1: `synchronize_rcu()` ← WAITS

Reader A: still reading OLD
Reader B: finishes
Reader C: finishes

t2:

Reader A: FINALLY finishes

t3: Grace period ENDS!

All readers that saw OLD pointer are done

t4: Free old memory ← SAFE!

Is RCU really safe?

Consider the use of global rt with no fences:

```
lookup(RELAXED(rt), route);
```

Could a CPU read new pointer but then old contents of *rt?

- Yes on alpha, No on all other existing architectures

When can you free memory of old routing table?

- When you are guaranteed no one is using it—how to determine?
 - for more info see [Read-Copy Update \(RCU\)](https://www.scs.stanford.edu/23wi-cs212/notes/synchronization1.pdf)

Limitations of RCU

1. Write Overhead High

- Must copy data
- Must wait for grace period
- Memory allocation/free overhead

2. Not for Frequent Updates

3. Complex Error Handling

4. Memory Overhead (keeping old versions in grace periods)

Deadlock problem

```

mutex_t m1, m2;

void p1(void *ignored) {
    lock(m1);
    lock(m2);
    /* critical section */
    unlock(m2);
    unlock(m1);
}

void p2(void *ignored) {
    lock(m2);
    lock(m1);
    /* critical section */
    unlock(m1);
    unlock(m2);
}

```

This program can cease to make progress – how?
 Can you have deadlock w/o mutexes?

Deadlock conditions

1. Limited access (mutual exclusion):

- Resource can only be shared with finite users

2. No preemption:

- Once resource granted, cannot be taken away

3. Multiple independent requests (hold and wait):

- Don't ask all at once (wait for next resource while holding current one)

4. Circularity in graph of requests

All of 1–4 necessary for deadlock to occur

Two approaches to dealing with deadlock:

- Pro-active: prevention
- Reactive: detection + corrective action

Prevent by eliminating one condition

1. Limited access (mutual exclusion):

- Resource can only be shared with finite users

2. No preemption:

- Once resource granted, cannot be taken away

3. Multiple independent requests (hold and wait):

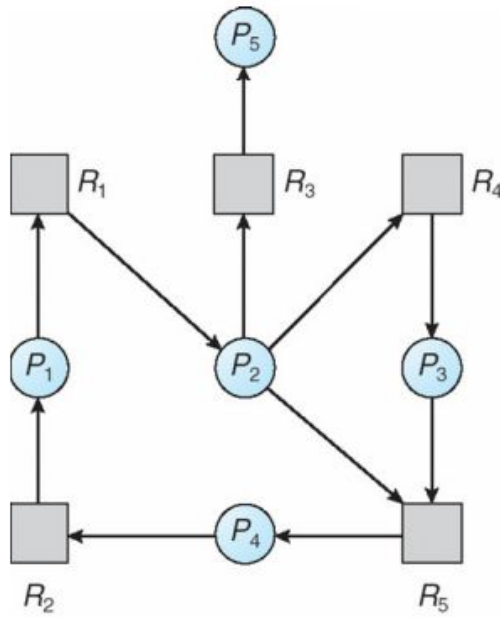
- Don't ask all at once (wait for next resource while holding current one)

4. Circularity in graph of requests

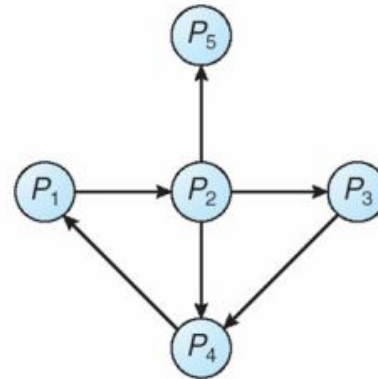
- Single lock for entire system: (problems?)
- Partial ordering of resources (next)

Detecting deadlocks

- Static approaches (hard)
- Dynamically, program grinds to a halt
 - Threads package can diagnose by keeping track of locks held:



(a)



(b)

Resource-Allocation Graph

Corresponding wait-for graph

Fixing and debugging deadlocks

Reboot system / restart application

Examine hung process with debugger

Threads package can deduce partial order

- - For each lock acquired, order with other locks held
- - If cycle occurs, abort with error
- - Detects potential deadlocks even if they do not occur

Or use transactions. . .

- - Another paradigm for handling concurrency
- - Often provided by databases, but some OSes use them

Transactions

A transaction is a collection of the properties

- **Atomicity** – all or none of actions happen
- **Consistency** – T leaves data in valid state
- **Isolation** – T's actions all appear to happen before or after every other transaction
- **Durability** – T's effects will survive reboots

ACID a set of properties of [database transactions](https://www.scs.stanford.edu/23wi-cs212/notes/synchronization2.pdf) intended to guarantee data validity despite errors, power failures, and other mishaps

Transactions typically executed concurrently

- But isolation means must appear not to
- Must roll-back transactions that use others' state
- Means you have to record all changes to undo them

- ★ When deadlock detected just abort a transaction
 - **Breaks the dependency cycle**

Transactional memory

Some modern processors support transactional memory

- ★ a promising alternative to lock-based synchronization mechanisms
 - Non-blocking

Transactional Synchronization Extensions (TSX)
[intel1§16]

- xbegin abort_handler – begins a transaction
- xend – commit a transaction
- xabort \$code – abort transaction with 8-bit code
- Note: nested transactions okay (also xtest tests if in transaction)

During transaction, processor tracks accessed memory

- Keeps read-set and write-set of cache lines
- Nothing gets written back to memory during transaction
- Transaction aborts (at xend or earlier) if any conflicts
- Otherwise, all dirty cache lines are “written” atomically
- (in practice switch to non-transactional M state of MESI)

TM system ensure atomicity by detecting and resolving any conflict arising between concurrent transactions

https://en.wikipedia.org/wiki/Transactional_memory#Available_implementations

<https://www.scs.stanford.edu/23wi-cs212/notes/synchronization2.pdf>

Using transactional memory

Idea: Use to get “free” fine-grained locking on a hash table

- E.g., concurrent inserts that don't touch same buckets are okay
 - Automatic Mutual Exclusion method

Note: Transactions are never guaranteed to commit

Can also use to poll for one of many asynchronous events

- Start transaction
- Fill cache with values to which you want to see changes
- Loop until a write causes your transaction to abort

Hardware lock elision (HLE)

concurrently executes lock critical sections as hardware transactions, but fallbacks to the original sequential lock fallback path when some hardware transaction fails.

- Begin a transaction when you acquire lock
- Other CPUs won't see lock acquired, can also enter critical section
- Okay not to have mutual exclusion when no memory conflicts!
- On conflict, abort and restart without transaction, thereby visibly acquiring lock (and aborting other concurrent transactions)

Intel support:

- Use xacquire prefix before xchgl (used for test and set)
- Use xrelease prefix before movl that releases lock
- Prefixes chosen to be noops on older CPUs (binary compatibility)

Hash table example:

- - Use xacquire xchgl in table-wide test-and-set spinlock
- - Works correctly on older CPUs (with coarse-grained lock)
- - Allows safe concurrent accesses on newer CPUs!