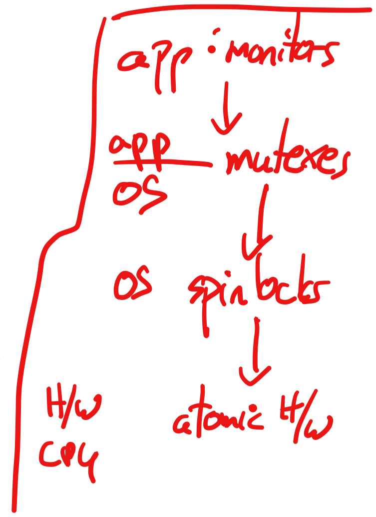
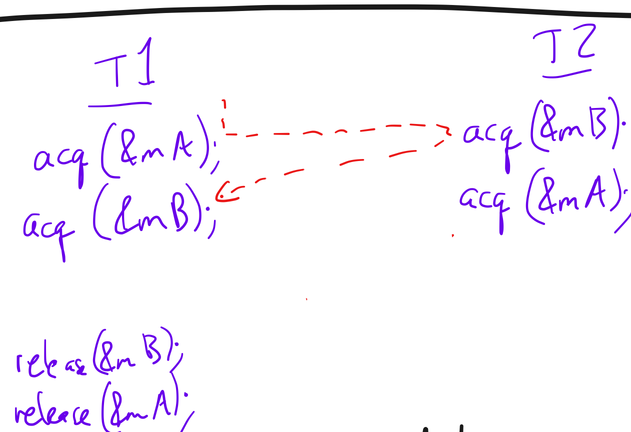
- [x] 1. Last time
- [x] 2. Implementation of locks, spinlocks, mutexes
- [x] 3. Deadlock
- [x] 4. Other progress issues
- [ ] 5. Performance issues
- [ ] 6. Programmability issues
- [ ] 7. Mutexes and interleavings
- [ ] 8. Questions

next time (items 5–8)



---

3. Deadlock



Happens when <u>all four</u> of these conditions are present:

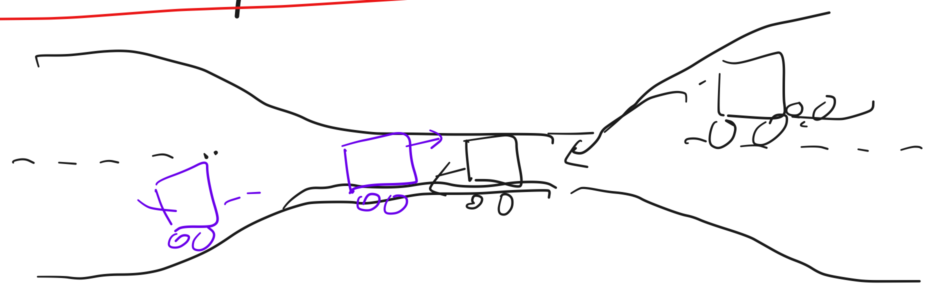
i. mutual exclusion
ii. hold and wait
iii. no pre-emption
iv. circular wait

What can we do about deadlock?

(a) ignore it [not crazy]
(b) detect + recover [can't always do that]
(c) avoid algorithmically [see text]

(d) negate one of the 4 conditions [*]

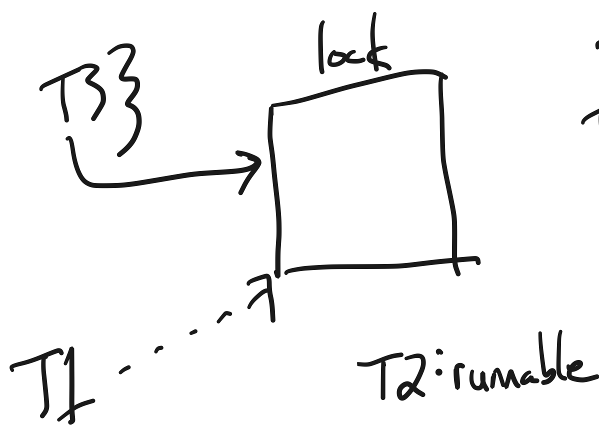
(e) static/dynamic detection tools



4. Other progress issues

Starvation

Priority inversion



T1: highest
T2: medium
T3: lowest

Assume: highest-prio runnable thread runs.

5. Performance issues + tradeoffs

- locks/mutexes have a perf cost

- spinlocks/mutexes have [illegible]
- coarse-grained locking limits parallelism...

∴ fine-grained locking leads to complexity, bugs

CS 202, Spring 2022
Handout 5 (Class 6)

The previous handout demonstrated the use of mutexes and condition variables. This handout demonstrates the use of monitors (which combine mutexes and condition variables).

1. The bounded buffer as a monitor

```
    // This is pseudocode that is inspired by C++.
    // Don't take it literally.

    class MyBuffer {
      public:
        MyBuffer();
        ~MyBuffer();
        void Enqueue(Item);
        Item = Dequeue();
      private:
        int count;
        int in;
        int out;
        Item buffer[BUFFER_SIZE];
        Mutex* mutex;
        Cond* nonempty;
        Cond* nonfull;
    }

    void
    MyBuffer::MyBuffer()
    {
        in = out = count = 0;
        mutex = new Mutex;
        nonempty = new Cond;
        nonfull = new Cond;
    }

    void
    MyBuffer::Enqueue(Item item)
    {
        mutex.acquire();
        while (count == BUFFER_SIZE)
            cond_wait(&nonfull, &mutex);

        buffer[in] = item;
        in = (in + 1) % BUFFER_SIZE;
        ++count;
        cond_signal(&nonempty, &mutex);
        mutex.release();
    }

    Item
    MyBuffer::Dequeue()
    {
        mutex.acquire();
        while (count == 0)
            cond_wait(&nonempty, &mutex);

        Item ret = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        --count;
        cond_signal(&nonfull, &mutex);
        mutex.release();
        return ret;
    }
```



```
    int main(int, char**)
    {
        MyBuffer buf;
        int dummy;
        tid1 = thread_create(producer, &buf);
        tid2 = thread_create(consumer, &buf);

        // never reach this point
        thread_join(tid1);
        thread_join(tid2);
        return -1;
    }

    void producer(void* buf)
    {
        MyBuffer* sharedbuf = reinterpret_cast<MyBuffer*>(buf);
        for (;;) {
            /* next line produces an item and puts it in nextProduced */
            Item nextProduced = means_of_production();
            sharedbuf->Enqueue(nextProduced);
        }
    }

    void consumer(void* buf)
    {
        MyBuffer* sharedbuf = reinterpret_cast<MyBuffer*>(buf);
        for (;;) {
            Item nextConsumed = sharedbuf->Dequeue();

            /* next line abstractly consumes the item */
            consume_item(nextConsumed);
        }
    }
```

Key point: *Threads* (the producer and consumer) are separate from *shared object* (MyBuffer). The synchronization happens in the shared object.

2. This monitor is a model of a database with multiple readers and writers. The high-level goal here is (a) to give a writer exclusive access (a single active writer means there should be no other writers and no readers) while (b) allowing multiple readers. Like the previous example, this one is expressed in pseudocode.

```
    // assume that these variables are initialized in a constructor
    state variables:
        AR = 0;  // # active readers
        AW = 0;  // # active writers
        WR = 0;  // # waiting readers
        WW = 0;  // # waiting writers

        Condition okToRead = NIL;
        Condition okToWrite = NIL;
        Mutex mutex = FREE;

    Database::read() {
        startRead();  // first, check self into the system
        Access Data
        doneRead();   // check self out of system
    }

    Database::startRead() {
        acquire(&mutex);
        while((AW + WW) > 0){
            WR++;
            wait(&okToRead, &mutex);
            WR--;
        }
        AR++;
        release(&mutex);
    }

   Database::doneRead() {
        acquire(&mutex);
        AR--;
        if (AR == 0 && WW > 0) { // if no other readers still
          signal(&okToWrite, &mutex);   // active, wake up writer
        }
        release(&mutex);
    }

    Database::write(){  // symmetrical
        startWrite();  // check in
        Access Data
        doneWrite();  // check out
    }

    Database::startWrite() {
        acquire(&mutex);
        while ((AW + AR) > 0) { // check if safe to write.
                                // if any readers or writers, wait
            WW++;
            wait(&okToWrite, &mutex);
            WW--;
        }
        AW++;
        release(&mutex);
    }

    Database::doneWrite() {
        acquire(&mutex);
        AW--;
        if (WW > 0) {
            signal(&okToWrite, &mutex); // give priority to writers
        } else if (WR > 0) {
            broadcast(&okToRead, &mutex);
        }
        release(&mutex);
    }

    NOTE: what is the starvation problem here?
```


3. Shared locks

```
    struct sharedlock {
      int i;
      Mutex mutex;
      Cond c;
    };

    void AcquireExclusive (sharedlock *sl) {
      acquire(&sl->mutex);
      while (sl->i) {
        wait (&sl->c, &sl->mutex);
      }
      sl->i = -1;
      release(&sl->mutex);
    }

    void AcquireShared (sharedlock *sl) {
      acquire(&sl->mutex);
      while (sl->i < 0) {
        wait (&sl->c, &sl->mutex);
      }
      sl->i++;
      release(&sl->mutex);
    }

    void ReleaseShared (sharedlock *sl) {
      acquire(&sl->mutex);
      if (!--sl->i)
        signal (&sl->c, &sl->mutex);
      release(&sl->mutex);
    }

    void ReleaseExclusive (sharedlock *sl) {
      acquire(&sl->mutex);
      sl->i = 0;
      broadcast (&sl->c, &sl->mutex);
      release(&sl->mutex);
    }
```

QUESTIONS:

A. There is a starvation problem here. What is it? (Readers can keep writers out if there is a steady stream of readers.)

B. How could you use these shared locks to write a cleaner version of the code in the prior item? (Though note that the starvation properties would be different.)

Implementation of spinlocks and mutexes

1. Here is a BROKEN spinlock implementation:

```
        struct Spinlock {
          int locked;
        }

        void acquire(Spinlock *lock) {
          while (1) {
            if (lock->locked == 0) { // A
              lock->locked = 1;      // B
              break;
            }
          }
        }

        void release (Spinlock *lock) {
          lock->locked = 0;
        }
```

What’s the problem? Two acquire()s on the same lock on different CPUs might both execute line A, and then both execute B. Then both will think they have acquired the lock. Both will proceed. That doesn’t provide mutual exclusion.

2. Correct spinlock implementation

Relies on atomic hardware instruction. For example, on the x86-64, doing

```
            "xchg addr, %rax"
    does the following:

    (i)   freeze all CPUs’ memory activity for address addr
    (ii)  temp <-- *addr
    (iii) *addr <-- %rax
    (iv)  %rax <-- temp
    (v)   un-freeze memory activity

/* pseudocode */
int xchg_val(addr, value) {
    %rax = value;
    xchg (*addr), %rax
}

/* bare-bones version of acquire */
void acquire (Spinlock *lock) {
  pushcli();    /* what does this do? */
  while (1) {
    if (xchg_val(&lock->locked, 1) == 0)
      break;
  }
}

void release(Spinlock *lock){
   xchg_val(&lock->locked, 0);
   popcli();    /* what does this do? */
}


/* optimization in acquire; call xchg_val() less frequently */
void acquire(Spinlock* lock) {
    pushcli();
    while (xchg_val(&lock->locked, 1) == 1) {
        while (lock->locked) ;
    }
}
```

The above is called a *spinlock* because acquire() spins. The bare-bones version is called a "test-and-set (TAS) spinlock"; the other is called a "test-and-test-and-set spinlock".

The spinlock above is great for some things, not so great for others. The main problem is that it *busy waits*: it spins, chewing up CPU cycles. Sometimes this is what we want (e.g., if the cost of going to sleep is greater than the cost of spinning for a few cycles waiting for another thread or process to relinquish the spinlock). But sometimes this is not at all what we want (e.g., if the lock would be held for a while: in those cases, the CPU waiting for the lock would waste cycles spinning instead of running some other thread or process).

NOTE: the spinlocks presented here can introduce performance issues when there is a lot of contention. (This happens even if the programmer is using spinlocks correctly.) The performance issues result from cross-talk among CPUs (which undermines caching and generates traffic on the memory bus). If we have time later, we will study a remediation of this issue (search the Web for "MCS locks").

ANOTHER NOTE: In everyday application-level programming, spinlocks will not be something you use (use mutexes instead). But you should know what these are for technical literacy, and to see where the mutual exclusion is truly enforced on modern hardware.

```
3. Mutex implementation

    The intent of a mutex is to avoid busy waiting: if the lock is not
    available, the locking thread is put to sleep, and tracked by a
    queue in the mutex. The next page has an implementation.
```



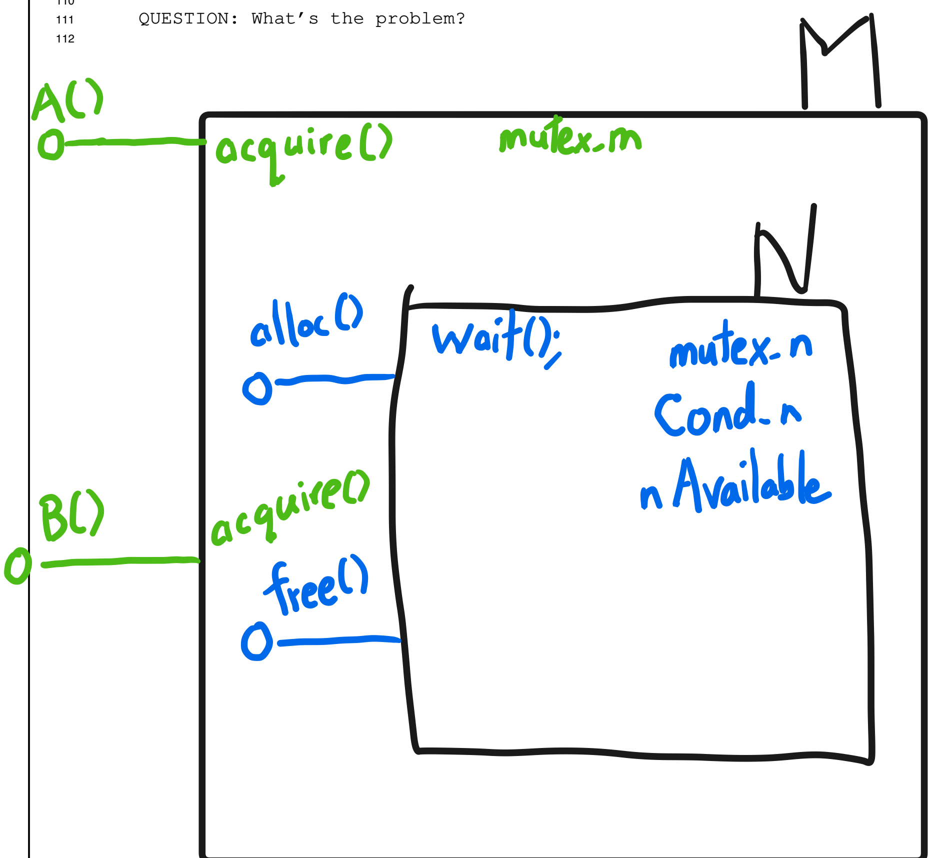
```
1  #include <sys/queue.h>
2  
3  typedef struct thread {
4      // ... Entries elided.
5      STAILQ_ENTRY(thread_t) qlink; // Tail queue entry.
6  } thread_t;
7  
8  struct Mutex {
9      // Current owner, or 0 when mutex is not held.
10     thread_t *owner;
11 
12     // List of threads waiting on mutex
13     STAILQ(thread_t) waiters;
14 
15     // A lock protecting the internals of the mutex.
16     Spinlock splock;   // as in item 1, above
17 };
18 
19 void mutex_acquire(struct Mutex *m) {
20 
21     acquire(&m->splock);
22 
23     // Check if the mutex is held; if not, current thread gets mutex and returns
24     if (m->owner == 0) {
25         m->owner = id_of_this_thread;
26         release(&m->splock);
27     } else {
28         // Add thread to waiters.
29         STAILQ_INSERT_TAIL(&m->waiters, id_of_this_thread, qlink);
30 
31         // Tell the scheduler to add current thread to the list
32         // of blocked threads. The scheduler needs to be careful
33         // when a corresponding sched_wakeup call is executed to
34         // make sure that it treats running threads correctly.
35         sched_mark_blocked(&id_of_this_thread);
36 
37         // Unlock spinlock.
38         release(&m->splock);
39 
40         // Stop executing until woken.
41         sched_swtch();
42 
43         // When we get to this line, we are guaranteed to hold the mutex. This
44         // is because we can get here only if context-switched-TO, which itself
45         // can happen only if this thread is removed from the waiting queue,
46         // marked "unblocked",/ and set to be the owner (in mutex_release()
47         // below). However, we might have held the mutex in lines 39-42
48         // (if we were context-switched out after the spinlock release(),
49         // followed by being run as a result of another thread's release of the
50         // mutex). But if that happens, it just means that we are
51         // context-switched out an "extra" time before proceeding.
52     }
53 }
54 
55 void mutex_release(struct Mutex *m) {
56     // Acquire the spinlock in order to make changes.
57     acquire(&m->splock);
58 
59     // Assert that the current thread actually owns the mutex
60     assert(m->owner == id_of_this_thread);
61 
62     // Check if anyone is waiting.
63     m->owner = STAILQ_GET_HEAD(&m->waiters);
64 
65     // If so, wake them up.
66     if (m->owner) {
67         sched_wakeone(&m->owner);
68         STAILQ_REMOVE_HEAD(&m->waiters, qlink);
69     }
70 
71     // Release the internal spinlock
72     release(&m->splock);
73 }
```

```
CS 202, Spring 2022
Handout 6 (Class 7)

1. Simple deadlock example

    T1:
        acquire(mutexA);
        acquire(mutexB);

        // do some stuff

        release(mutexB);
        release(mutexA);

    T2:
        acquire(mutexB);
        acquire(mutexA);

        // do some stuff

        release(mutexA);
        release(mutexB);
```



```
2. More subtle deadlock example

    Let M be a monitor (shared object with methods protected by mutex)
    Let N be another monitor

    class M {
        private:
            Mutex mutex_m;

            // instance of monitor N
            N another_monitor;

            // Assumption: no other objects in the system hold a pointer
            // to our "another_monitor"

        public:
            M();
            ~M();
            void methodA();
            void methodB();
    };

    class N {
        private:
            Mutex mutex_n;
            Cond cond_n;
            int navailable;

        public:
            N();
            ~N();
            void* alloc(int nwanted);
            void  free(void*);
    }

    int
    N::alloc(int nwanted) {
        acquire(&mutex_n);
        while (navailable < nwanted) {
            wait(&cond_n, &mutex_n);
        }

        // peel off the memory

        navailable -= nwanted;
        release(&mutex_n);
    }

    void
    N::free(void* returning_mem) {

        acquire(&mutex_n);

        // put the memory back

        navailable += returning_mem;

        broadcast(&cond_n, &mutex_n);

        release(&mutex_n);
    }
```

```
    void
    M::methodA() {

        acquire(&mutex_m);

        void* new_mem = another_monitor.alloc(int nbytes);

        // do a bunch of stuff using this nice
        // chunk of memory n allocated for us

        release(&mutex_m);
    }

    void
    M::methodB() {

        acquire(&mutex_m);

        // do a bunch of stuff

        another_monitor.free(some_pointer);

        release(&mutex_m);
    }

    QUESTION: What's the problem?
```




```
3. Locking brings a performance vs. complexity trade-off

/*
 *      linux/mm/filemap.c
 *
 * Copyright (C) 1994-1999  Linus Torvalds
 */

/*
 * This file handles the generic file mmap semantics used by
 * most "normal" filesystems (but you don't /have/ to use this:
 * the NFS filesystem used to do this differently, for example)
 */
#include <linux/export.h>
#include <linux/compiler.h>
#include <linux/dax.h>
#include <linux/fs.h>
#include <linux/sched/signal.h>
#include <linux/uaccess.h>
#include <linux/capability.h>
#include <linux/kernel_stat.h>
#include <linux/gfp.h>
#include <linux/mm.h>
#include <linux/swap.h>
#include <linux/mman.h>
#include <linux/pagemap.h>
#include <linux/file.h>
#include <linux/uio.h>
#include <linux/hash.h>
#include <linux/writeback.h>
#include <linux/backing-dev.h>
#include <linux/pagevec.h>
#include <linux/blkdev.h>
#include <linux/security.h>
#include <linux/cpuset.h>
#include <linux/hugetlb.h>
#include <linux/memcontrol.h>
#include <linux/cleancache.h>
#include <linux/shmem_fs.h>
#include <linux/rmap.h>
#include "internal.h"

#define CREATE_TRACE_POINTS
#include <trace/events/filemap.h>

/*
 * FIXME: remove all knowledge of the buffer layer from the core VM
 */
#include <linux/buffer_head.h> /* for try_to_free_buffers */

#include <asm/mman.h>

/*
 * Shared mappings implemented 30.11.1994. It's not fully working yet,
 * though.
 *
 * Shared mappings now work. 15.8.1995  Bruno.
 *
 * finished 'unifying' the page and buffer cache and SMP-threaded the
 * page-cache, 21.05.1999, Ingo Molnar <mingo@redhat.com>
 *
 * SMP-threaded pagemap-LRU 1999, Andrea Arcangeli <andrea@suse.de>
 */

/*
 * Lock ordering:
 *
 *  ->i_mmap_rwsem              (truncate_pagecache)
 *    ->private_lock            (__free_pte->__set_page_dirty_buffers)
 *      ->swap_lock             (exclusive_swap_page, others)
 *        ->i_pages lock
 *
 *  ->i_mutex
```

```
 *    ->i_mmap_rwsem            (truncate->unmap_mapping_range)
 *
 *  ->mmap_sem
 *    ->i_mmap_rwsem
 *      ->page_table_lock or pte_lock   (various, mainly in memory.c)
 *        ->i_pages lock        (arch-dependent flush_dcache_mmap_lock)
 *
 *  ->mmap_sem
 *    ->lock_page               (access_process_vm)
 *
 *  ->i_mutex                   (generic_perform_write)
 *    ->mmap_sem                (fault_in_pages_readable->do_page_fault)
 *
 *  bdi->wb.list_lock
 *    sb_lock                   (fs/fs-writeback.c)
 *    ->i_pages lock            (__sync_single_inode)
 *
 *  ->i_mmap_rwsem
 *    ->anon_vma.lock           (vma_adjust)
 *
 *  ->anon_vma.lock
 *    ->page_table_lock or pte_lock     (anon_vma_prepare and various)
 *
 *  ->page_table_lock or pte_lock
 *    ->swap_lock               (try_to_unmap_one)
 *    ->private_lock            (try_to_unmap_one)
 *    ->i_pages lock            (try_to_unmap_one)
 *    ->zone_lru_lock(zone)     (follow_page->mark_page_accessed)
 *    ->zone_lru_lock(zone)     (check_pte_range->isolate_lru_page)
 *    ->private_lock            (page_remove_rmap->set_page_dirty)
 *    ->i_pages lock            (page_remove_rmap->set_page_dirty)
 *    bdi.wb->list_lock         (page_remove_rmap->set_page_dirty)
 *    ->inode->i_lock           (page_remove_rmap->set_page_dirty)
 *    ->memcg->move_lock        (page_remove_rmap->lock_page_memcg)
 *    bdi.wb->list_lock         (zap_pte_range->set_page_dirty)
 *    ->inode->i_lock           (zap_pte_range->set_page_dirty)
 *    ->private_lock            (zap_pte_range->__set_page_dirty_buffers)
 *
 * ->i_mmap_rwsem
 *   ->tasklist_lock            (memory_failure, collect_procs_ao)
 */

static int page_cache_tree_insert(struct address_space *mapping,
                                  struct page *page, void **shadowp)
{
        struct radix_tree_node *node;
        .....
```

[the point is: fine-grained locking leads to complexity.]

4. Cautionary tale

Consider the code below:

```
    struct foo {
        int abc;
        int def;
    };
    static int ready = 0;
    static mutex_t mutex;
    static struct foo* ptr = 0;

    void
    doublecheck_alloc()
    {
        if (!ready) { /* <-- accesses shared variable w/out holding mutex */

            mutex_acquire(&mutex);
            if (!ready) {
                ptr = alloc_foo(); /* <-- sets ptr to be non-zero */
                ready = 1;
            }

            mutex_release(&mutex);

        }
        return;
    }
```

This is an example of the so-called "double-checked locking pattern." The programmer’s intent is to avoid a mutex acquistion in the common case that ’ptr’ is already initialized. So the programmer checks a flag called ’ready’ before deciding whether to acquire the mutex and initialize ’ptr’. The intended use of doublecheck_alloc() is something like this:

```
    void f() {
        doublecheck_alloc();
        ptr->abc = 5;
    }

    void g() {
        doublecheck_alloc();
        ptr->def = 6;
    }
```

We assume here that mutex_acquire() and mutex_release() are implemented correctly (each contains memory barriers internally, etc.). Furthermore, we assume that the compiler does not reorder instructions.

NEVERTHELESS, on multi-CPU machines that do not offer sequential consistency, doublecheck_alloc() is broken. What is the bug?

Unfortunately, double-checked initialization (or double-checked locking as it’s sometimes known) is a common coding pattern. Even some references on threads suggest it! Still, it’s broken.

While you can fix it (in C) by adding another barrier (exercise: where?), this is not recommended, as the code is tricky to reason about. One of the points of this example is to show you why it’s so important to protect global data with a mutex, even if "all" one is doing is reading memory, and even if the shortcut looks harmless.

```
Finally, here are some references on this topic:

    --http://www.aristeia.com/Papers/DDJ_Jul_Aug_2004_revised.pdf
     explores issues with this pattern in C++

    --The "Double-Checked Locking is Broken" Declaration:
    http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html

    --C++11 provides a way to implement the pattern correctly and
    portably (again, using memory barriers):
    https://preshing.com/20130930/double-checked-locking-is-fixed-in-cpp11/
```