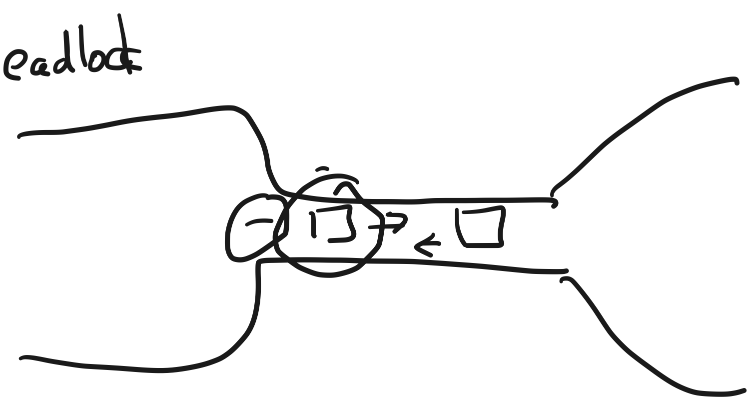
1. Last time
2. Deadlock
3. Other progress issues
4. Performance issues
5. Programmability issues
6. Mutexes and interleavings
7. Questions

Thrac-25

---

2. Deadlock



Happens when all four 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

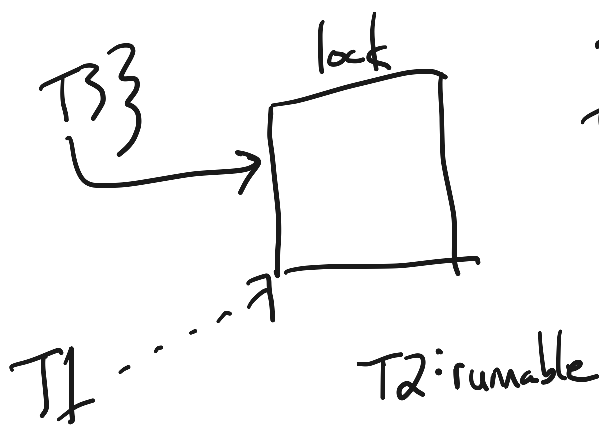
---

3. Other progress issues

Starvation

Priority inversion

T1: highest
T2: medium
T3: lowest



Assume: highest-prio runnable thread runs.

---

4. Performance issues + tradeoffs

spinlocks/mutexes have a perf cost

- spinlocks/mutexes have ...

- coarse-grained locking limits parallelism ...

BKL

... - fine-grained locking leads to complexity, bugs

---

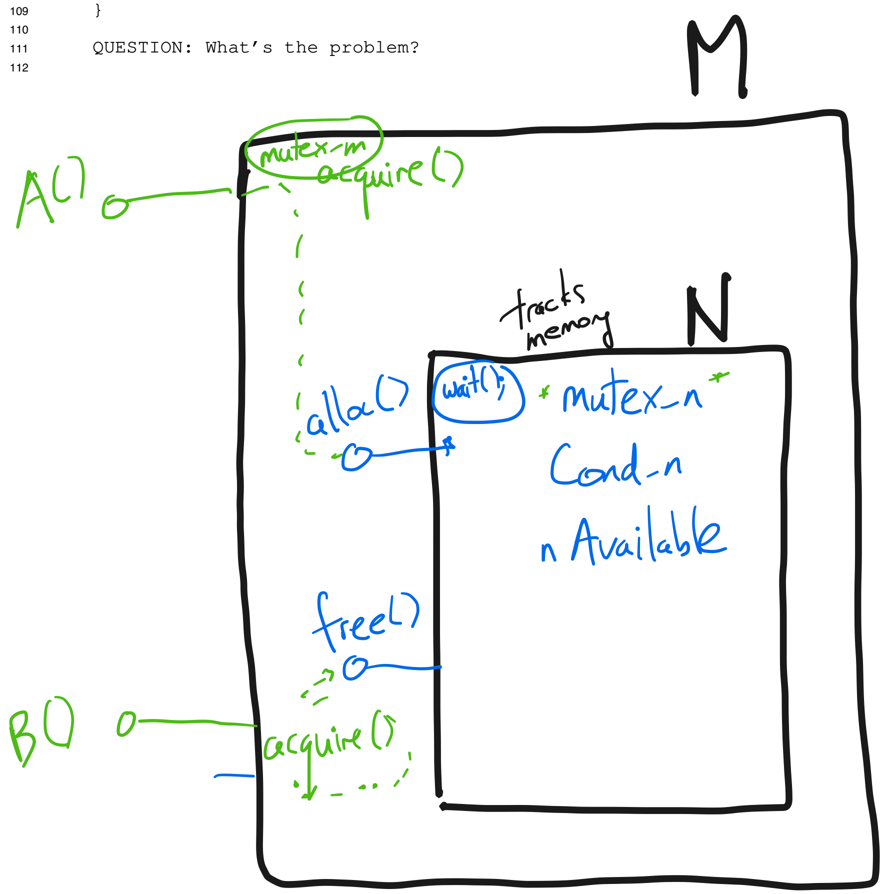
5. Programmability issues

```
CS 202, Fall 2021
Handout 6 (Class 8)

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?
```



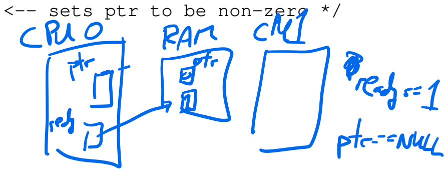

```
3. Locking brings a performance vs. complexity trade-off

/*
 *      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/
```