```
2. 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.]
```

3. 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 acquisition 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 barriers (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/

```
CS 202, Spring 2024
Handout 8 (Class 9)

Therac-25

1. Software problem #1 (our best guess)

    A. Three threads:

        --Hand: sets the collimator/turntable position

        --Treat: sets a bunch of other parameters. Part of its job takes
        eight seconds, during which time it's ignoring everything else.

        --Vtkbp (keyboard handler): invoked when user types. It parses
        the input, and writes to a two-byte shared variable, "MEOS" (mode/energy
        offset)
            --"Treat" reads top byte, sets current and energy
            --"Hand" reads bottom byte, sets the collimator/turntable position

    B. Pseudocode:

        Vtkbp (gets and parses keyboard input):

            data_completion_flag = 0

            while (1) {
                wait_for_keyboard_activity();
                /* there was some keyboard activity; let's check it */
                if (cursor_in_bottom_right) {
                    parse_the_input();
                    set the MEOS variable
                    set data_completion_flag = 1;
                    signal hand thread
                    signal treat thread
                } else {
                    /* operator still typing */
                    data_completion_flag = 0;
                }
                yield();
            }


        Hand (sets the turntable position):

            while (1) {
                wait until signalled
                read bottom byte of MEOS variable
                /* next line executes quickly */
                set turntable position
                yield();
            }

        Treat (sets a bunch of parameters and delivers treatment):

            dataent() { /* this is a subroutine that was called */

                while (1) {
                    wait until signalled
                    read top byte of MEOS variable
                    set_energy_and_current();
                    set_bending_magnets(); /* this takes eight seconds */
                    if (data_completion_flag == 1)
                        break;
                }
                /*
                 * now we leave the subroutine and progress to a state in
                 * which the machine will accept a "beam on" command
                 */
                return;
            }

```



```
2. Software problem #2 (simplified)

   [Simplifying here and condensing to one thread of control; in
   reality, the functions below are spread over two different threads,
   but that is not actually the problem, despite what the paper
   sometimes says. The problem appears to be given by the following
   simplified description.]

    class3 = 0;

    while (1) {

        if (in field light position) {
            increment class3;
        }

        check whether operator pressed "set"

        if (operator pressed set) {
            if (class3 != 0) {
                move turntable out of field light position;
            }
            break;
        }
    }

    What's the issue here? (Hint: class3 is only one byte.)

```