```
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.
```



```c
#include <sys/queue.h>

typedef struct thread {
    // ... Entries elided.
    STAILQ_ENTRY(thread_t) qlink; // Tail queue entry.
} thread_t;

struct Mutex {
    // Current owner, or 0 when mutex is not held.
    thread_t *owner;

    // List of threads waiting on mutex
    STAILQ(thread_t) waiters;

    // A lock protecting the internals of the mutex.
    Spinlock splock;   // as in item 1, above
};

void mutex_acquire(struct Mutex *m) {

    acquire(&m->splock);

    // Check if the mutex is held; if not, current thread gets mutex and returns
    if (m->owner == 0) {
        m->owner = id_of_this_thread;
        release(&m->splock);
    } else {
        // Add thread to waiters.
        STAILQ_INSERT_TAIL(&m->waiters, id_of_this_thread, qlink);

        // Tell the scheduler to add current thread to the list
        // of blocked threads. The scheduler needs to be careful
        // when a corresponding sched_wakeup call is executed to
        // make sure that it treats running threads correctly.
        sched_mark_blocked(&id_of_this_thread);

        // Unlock spinlock.
        release(&m->splock);

        // Stop executing until woken.
        sched_swtch();

        // When we get to this line, we are guaranteed to hold the mutex. This
        // is because we can get here only if context-switched-TO, which itself
        // can happen only if this thread is removed from the waiting queue,
        // marked "unblocked", and set to be the owner (in mutex_release()
        // below). However, we might have held the mutex in lines 39-42
        // (if we were context-switched out after the spinlock release(),
        // followed by being run as a result of another thread's release of the
        // mutex). But if that happens, it just means that we are
        // context-switched out an "extra" time before proceeding.
    }
}

void mutex_release(struct Mutex *m) {
    // Acquire the spinlock in order to make changes.
    acquire(&m->splock);

    // Assert that the current thread actually owns the mutex
    assert(m->owner == id_of_this_thread);

    // Check if anyone is waiting.
    m->owner = STAILQ_GET_HEAD(&m->waiters);

    // If so, wake them up.
    if (m->owner) {
        sched_wakeone(&m->owner);
        STAILQ_REMOVE_HEAD(&m->waiters, qlink);
    }

    // Release the internal spinlock
    release(&m->splock);
}
```

```
CS 202, Fall 2024
Handout 7 (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);
    }

```

```
86      void
87      M::methodA() {
88
89          acquire(&mutex_m);
90
91          void* new_mem = another_monitor.alloc(int nbytes);
92
93          // do a bunch of stuff using this nice
94          // chunk of memory n allocated for us
95
96          release(&mutex_m);
97      }
98
99      void
100     M::methodB() {
101
102         acquire(&mutex_m);
103
104         // do a bunch of stuff
105
106         another_monitor.free(some_pointer);
107
108         release(&mutex_m);
109     }
110
111     QUESTION: What's the problem?
```



```
1   2. Locking brings a performance vs. complexity trade-off
2
3   /*
4    *      linux/mm/filemap.c
5    *
6    * Copyright (C) 1994-1999  Linus Torvalds
7    */
8
9   /*
10   * This file handles the generic file mmap semantics used by
11   * most "normal" filesystems (but you don't /have/ to use this:
12   * the NFS filesystem used to do this differently, for example)
13   */
14  #include <linux/export.h>
15  #include <linux/compiler.h>
16  #include <linux/dax.h>
17  #include <linux/fs.h>
18  #include <linux/sched/signal.h>
19  #include <linux/uaccess.h>
20  #include <linux/capability.h>
21  #include <linux/kernel_stat.h>
22  #include <linux/gfp.h>
23  #include <linux/mm.h>
24  #include <linux/swap.h>
25  #include <linux/mman.h>
26  #include <linux/pagemap.h>
27  #include <linux/file.h>
28  #include <linux/uio.h>
29  #include <linux/hash.h>
30  #include <linux/writeback.h>
31  #include <linux/backing-dev.h>
32  #include <linux/pagevec.h>
33  #include <linux/blkdev.h>
34  #include <linux/security.h>
35  #include <linux/cpuset.h>
36  #include <linux/hugetlb.h>
37  #include <linux/memcontrol.h>
38  #include <linux/cleancache.h>
39  #include <linux/shmem_fs.h>
40  #include <linux/rmap.h>
41  #include "internal.h"
42
43  #define CREATE_TRACE_POINTS
44  #include <trace/events/filemap.h>
45
46  /*
47   * FIXME: remove all knowledge of the buffer layer from the core VM
48   */
49  #include <linux/buffer_head.h> /* for try_to_free_buffers */
50
51  #include <asm/mman.h>
52
53  /*
54   * Shared mappings implemented 30.11.1994. It's not fully working yet,
55   * though.
56   *
57   * Shared mappings now work. 15.8.1995  Bruno.
58   *
59   * finished 'unifying' the page and buffer cache and SMP-threaded the
60   * page-cache, 21.05.1999, Ingo Molnar <mingo@redhat.com>
61   *
62   * SMP-threaded pagemap-LRU 1999, Andrea Arcangeli <andrea@suse.de>
63   */
64
65  /*
66   * Lock ordering:
67   *
68   *  ->i_mmap_rwsem              (truncate_pagecache)
69   *    ->private_lock            (__free_pte->__set_page_dirty_buffers)
70   *      ->swap_lock             (exclusive_swap_page, others)
71   *        ->i_pages lock
72   *
73   *  ->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.
```