- [x] 1. Last time
- [x] 2. Deadlock, continued
- [x] 3. Other progress issues
- [x] 4. Performance issues
- [x] 5. Programmability issues
- [x] 6. Mutexes and interleaving

ONE HANDOUT

---

2. Deadlock

T1	T2
acq(&mA);	acq(&mB);
acq(&mB);	acq(&mA);
release(&mB);	
release(&mA);	

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
(b) detect + recover
(c) avoid algorithmically
(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

T3 → lock
T1 ....→

T2: runnable

Assume: highest-prio runnable thread runs.

4. Performance issues and tradeoffs

(a) Invoking spinlocks/mutexes can be expensive

MCS locks

(b) Coarse locks limit available parallelism...

(c) ... but fine-grained locking leads to complexity and hence bugs

---

5. Programmability issues

- loss of modularity

printf("...");

- what's the fundamental problem?

---

6. mutexes and interleavings

{ops}

mem barrier

{ops}

≡A

```
release() <  fence
  B
```

"A stuff" happens before "B stuff"
although, within A and B, the "stuff"
can happen out of order from the perspective
of other cores, assuming they could view the
operations separately.

```
1   CS 202, Spring 2024
2   Handout 7 (Class 8)
3
4   1. More subtle deadlock example (last time, saw simpler example)
5
6       Let M be a monitor (shared object with methods protected by mutex)
7       Let N be another monitor
8
9       class M {
10          private:
11              Mutex mutex_m;
12
13              // instance of monitor N
14              N another_monitor;
15
16              // Assumption: no other objects in the system hold a pointer
17              // to our "another_monitor"
18
19          public:
20              M();
21              ~M();
22              void methodA();
23              void methodB();
24      };
25
26      class N {
27          private:
28              Mutex mutex_n;
29              Cond cond_n;
30              int navailable;
31
32          public:
33              N();
34              ~N();
35              void* alloc(int nwanted);
36              void  free(void*);
37      }
38
39      int
40      N::alloc(int nwanted) {
41          acquire(&mutex_n);
42          while (navailable < nwanted) {
43              wait(&cond_n, &mutex_n);
44          }
45
46          // peel off the memory
47
48          navailable -= nwanted;
49          release(&mutex_n);
50      }
51
52      void
53      N::free(void* returning_mem) {
54
55          acquire(&mutex_n);
56
57          // put the memory back
58
59          navailable += returning_mem;
60
61          broadcast(&cond_n, &mutex_n);
62
63          release(&mutex_n);
64      }
65
```



```
66      void
67      M::methodA() {
68
69          acquire(&mutex_m);
70
71          void* new_mem = another_monitor.alloc(int nbytes);
72
73          // do a bunch of stuff using this nice
74          // chunk of memory n allocated for us
75
76          release(&mutex_m);
77      }
78
79      void
80      M::methodB() {
81
82          acquire(&mutex_m);
83
84          // do a bunch of stuff
85
86          another_monitor.free(some_pointer);
87
88          release(&mutex_m);
89      }
90
91      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
```



```
74   *    ->i_mmap_rwsem            (truncate->unmap_mapping_range)
75   *
76   *  ->mmap_sem
77   *    ->i_mmap_rwsem
78   *      ->page_table_lock or pte_lock   (various, mainly in memory.c)
79   *        ->i_pages lock        (arch-dependent flush_dcache_mmap_lock)
80   *
81   *  ->mmap_sem
82   *    ->lock_page               (access_process_vm)
83   *
84   *  ->i_mutex                   (generic_perform_write)
85   *    ->mmap_sem                (fault_in_pages_readable->do_page_fault)
86   *
87   *  bdi->wb.list_lock
88   *    sb_lock                   (fs/fs-writeback.c)
89   *    ->i_pages lock            (__sync_single_inode)
90   *
91   *  ->i_mmap_rwsem
92   *    ->anon_vma.lock           (vma_adjust)
93   *
94   *  ->anon_vma.lock
95   *    ->page_table_lock or pte_lock     (anon_vma_prepare and various)
96   *
97   *  ->page_table_lock or pte_lock
98   *    ->swap_lock               (try_to_unmap_one)
99   *    ->private_lock            (try_to_unmap_one)
100  *    ->i_pages lock            (try_to_unmap_one)
101  *    ->zone_lru_lock(zone)     (follow_page->mark_page_accessed)
102  *    ->zone_lru_lock(zone)     (check_pte_range->isolate_lru_page)
103  *    ->private_lock            (page_remove_rmap->set_page_dirty)
104  *    ->i_pages lock            (page_remove_rmap->set_page_dirty)
105  *    bdi.wb->list_lock         (page_remove_rmap->set_page_dirty)
106  *    ->inode->i_lock           (page_remove_rmap->set_page_dirty)
107  *    ->memcg->move_lock        (page_remove_rmap->lock_page_memcg)
108  *    bdi.wb->list_lock         (zap_pte_range->set_page_dirty)
109  *    ->inode->i_lock           (zap_pte_range->set_page_dirty)
110  *    ->private_lock            (zap_pte_range->__set_page_dirty_buffers)
111  *
112  * ->i_mmap_rwsem
113  *   ->tasklist_lock            (memory_failure, collect_procs_ao)
114  */
115
116 static int page_cache_tree_insert(struct address_space *mapping,
117                                   struct page *page, void **shadowp)
118 {
119         struct radix_tree_node *node;
120         .....
121
122 [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/
```