// C++

```
// from the header file
class Singleton {
public:
static Singleton* instance();
  ...
private:
  static Singleton* pInstance;
};

// from the implementation file
Singleton* Singleton::pInstance = 0;

Singleton* Singleton::instance() {
  if (pInstance == 0) {
    pInstance = new Singleton;
  }
  return pInstance;
}
```

Save

Singleton. instance()

T1

T2

lock

unlock

```
Singleton* Singleton::instance() {
  if (pInstance == 0) {           // 1st test
    Lock lock;
    if (pInstance == 0) {         // 2nd test
      pInstance = new Singleton;
    }
  }
  return pInstance;
}
```

T2

T1

① malloc mem

② init mem

③ assign pInstance

➔

```
pInstance =                          // Step 3
  operator new(sizeof(Singleton)); // Step 1
new (pInstance) Singleton;           // Step 2
```

T1 T2

①

③

②

BUG

Example borrowed from "C++ and the Perils of Double-Checked Locking"
https://www.aristeia.com/Papers/DDJ_Jul_Aug_2004_revised.pdf

Week 6.b
CS 5600
02/15 2023
https://naizhengtan.github.io/23spring/

1. Concurrency, continued ←
3. Deadlock
3. Other concurrency issues
4. Other synchronization mechanisms

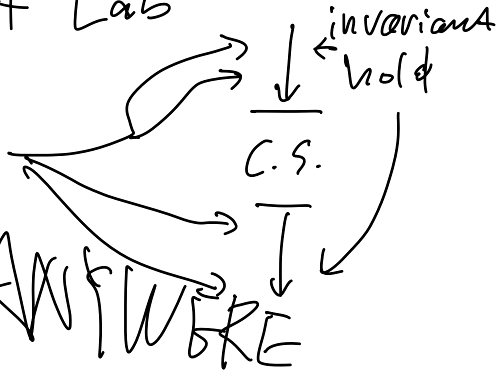
- midterm. 02/27.
- review. 02/22. ⇐
- Lab3.

Lecture + Lab
↓ ↓
review + Lab

- "terms"
"concepts"

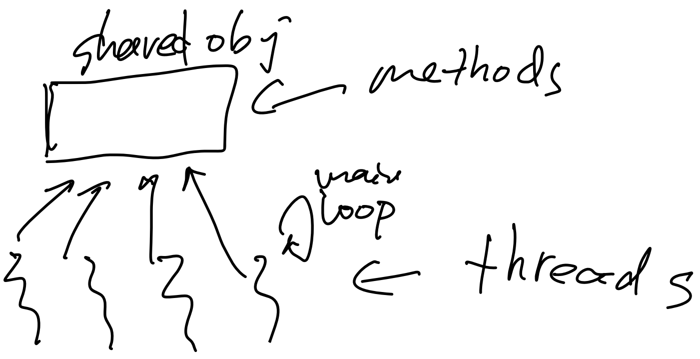
- What is "invariant"?

assert(count == items(buffer))

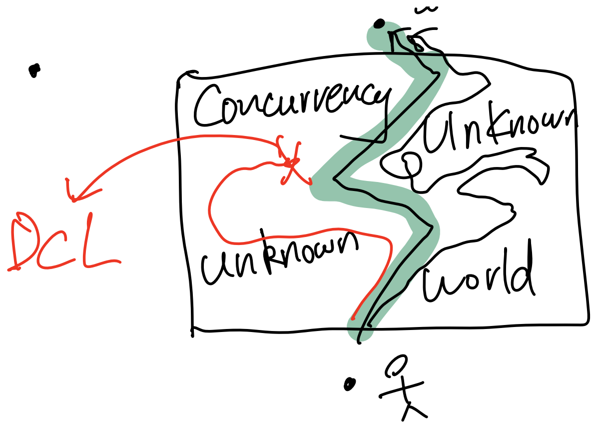


- mutex
- C.V. { cond_wait(&cv, &m)
signal(&cv) ← at least 1
broadcast(&cv) ← all

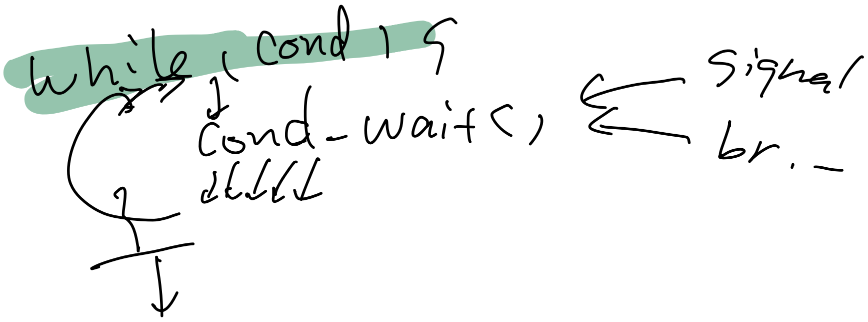
- monitor
\+
3 → 6 Rules. (☆)
\+
4-step



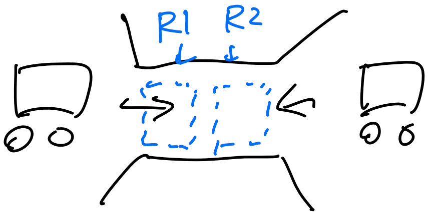
dst

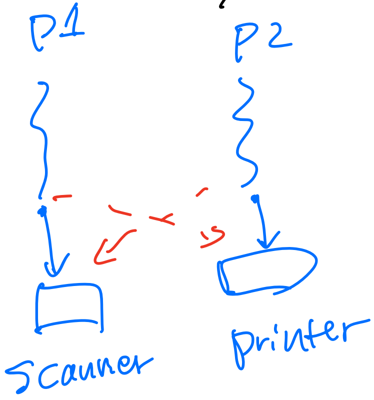


- Signal vs. broadcast

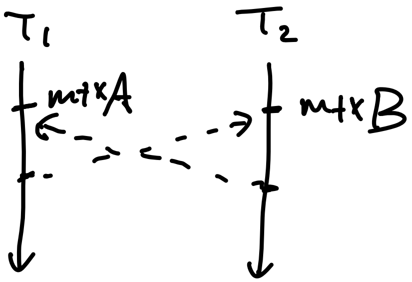


- Deadlock.





```
CS5600, Handout week 6.b

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();
    };
```

//app datastore

// mem allocator

```
    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?
```

// mem alloc

aval mem.

wait c.v.

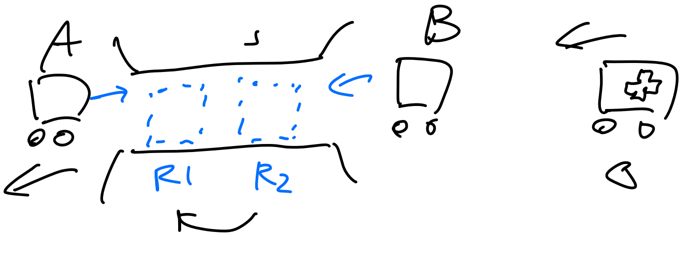
// signal

T2
methodB
↳ lock m ?

T1
methodA()
↳ lock of m mtx_m
↳ alloc()
↳ ~~lock~~ mtx_n
↳ wait c.v.

deadlock

1. mutual exclusion
2. hold-and-wait
3. no preemption
4. circular wait

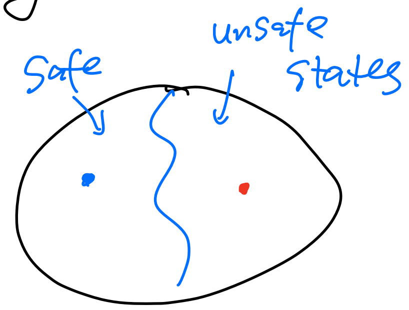


Solutions:

(a) ignore

(b) detect & recovery

(c) avoid algo



- banker's algorithm

```
void request(...) {
    acquire(&mutex);
    assert(system in a safe state);
    while (state that would result from giving
           resource to thread is not safe) {
        wait(&cv, &mutex);
    }
    update state by giving resource to thread
    assert(system in a safe state);
    release(&mutex);
}
```

--an example:

-- a bank has 10 coins

-- three customers (max coins needed):

A (9), B (4), and C (7)

-- current status:

	has	max
A	3+1	9
B	2	4
C	2	7

bank: 3-1 =2

A→ +1 Reject

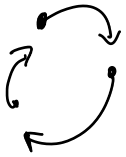
Q: is this state safe?

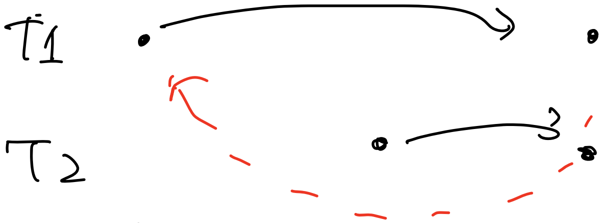
$B \rightarrow C \rightarrow A$

(d) negate one of the four

1. mutual exclusion
2. hold-and-wait → OCC
3. no preemption
4. ¬ circular wait

Lock1 → Lock2 → Lock3





(e) static & dynamic detection

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
```

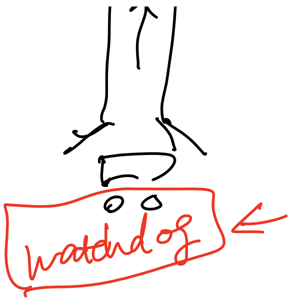
```
*   ->i_mmap_rwsem (truncate->unmap_mapping_range)
*
* ->mmap_sem
*   ->i_mmap_rwsem
*     ->page_table_lock or pte_lock (various, mainly in memory.c)
*       ->i_pages lock (arch-dependent flush_dcache_mmap_lock)
*
* ->mmap_sem
*   ->lock_page (access_process_vm)
*
* ->i_mutex (generic_perform_write)
*   ->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.]

Other concurrency issues

(a) progress issues

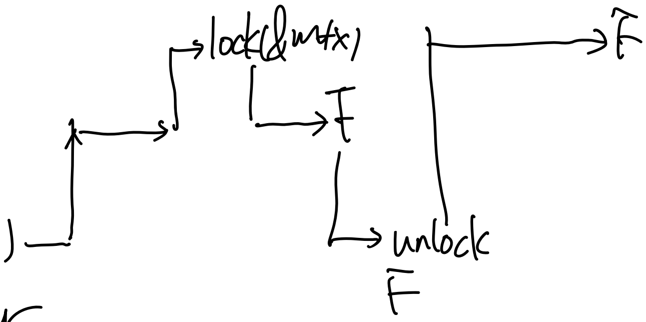
- Livelock
- Starvation
- priority inversion



management T1 (high)

Communication T2 (mid)

data gathering T3 (low) — lock(&mtx) —



Mars PathFinder

(b) other concurrency bugs

- atomicity-violation bugs

```
T1:
  if (thd->info) {
    ...
    use(thd->info);
    ...
  }

T2:
  thd->info = NULL
```

TTCTTU

- 105 bugs
  - 31 deadlock
  - 72

- order-violation bugs

```
T1:
  void init() {
    thd = createThread(...)
  }

T2:
  void main() {
    state = thd->info
  }
```

(c) performance vs. complexity

Other synchronization mechanisms

- semaphores (old)
- Barrier
- spinlocks
- reader-writer locks
- Read-copy-update (RCU)
- deterministic multithreading

mutex

lock

0: unlocked

1: locked

```
acquire() {
    while(1) {
        check if lock==0 {
            lock=1        // lock it
        } }
}
```

⇓

BROKEN IMPL

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.

```
 97  3. Mutex implementation
 98
 99     The intent of a mutex is to avoid busy waiting: if the lock is not
100     available, the locking thread is put to sleep, and tracked by a
101     queue in the mutex. The next page has an implementation.
102
103     #include <sys/queue.h>
104
105     typedef struct thread {
106         // ... Entries elided.
107         STAILQ_ENTRY(thread_t) qlink; // Tail queue entry.
108     } thread_t;
109
110     struct Mutex {
111         // Current owner, or 0 when mutex is not held.
112         thread_t *owner;
113
114         // List of threads waiting on mutex
115         STAILQ(thread_t) waiters;
116
117         // A lock protecting the internals of the mutex.
118         Spinlock splock; // as in item 1, above
119     };
120
121     void mutex_acquire(struct Mutex *m) {
122
123         acquire(&m->splock);
124
125         // Check if the mutex is held; if not, current thread gets mutex and returns
126         if (m->owner == 0) {
127             m->owner = id_of_this_thread;
128             release(&m->splock);
129         } else {
130             // Add thread to waiters.
131             STAILQ_INSERT_TAIL(&m->waiters, id_of_this_thread, qlink);
132
133             // Tell the scheduler to add current thread to the list
134             // of blocked threads. The scheduler needs to be careful
135             // when a corresponding sched_wakeup call is executed to
136             // make sure that it treats running threads correctly.
137             sched_mark_blocked(&id_of_this_thread);
138
139             // Unlock spinlock.
140             release(&m->splock);
141
142             // Stop executing until woken.
143             sched_swtch();
144
145             // When we get to this line, we are guaranteed to hold the mutex. This
146             // is because we can get here only if context-switched-TO, which itself
147             // can happen only if this thread is removed from the waiting queue,
148             // marked "unblocked",/ and set to be the owner (in mutex_release()
149             // below). However, we might actually have held the mutex in lines 141-144
150             // (if we were context-switched out after the spinlock release(),
151             // followed by being run as a result of another thread's release of the
152             // mutex). But if that happens, it just means that we are
153             // context-switched out an "extra" time before proceeding.
154         }
155     }
156
```

list

```
157     void mutex_release(struct Mutex *m) {
158         // Acquire the spinlock in order to make changes.
159         acquire(&m->splock);
160
161         // Assert that the current thread actually owns the mutex
162         assert(m->owner == id_of_this_thread);
163
164         // Check if anyone is waiting.
165         m->owner = STAILQ_GET_HEAD(&m->waiters);
166
167         // If so, wake them up.
168         if (m->owner) {
169             sched_wakeone(&m->owner);
170             STAILQ_REMOVE_HEAD(&m->waiters, qlink);
171         }
172
173         // Release the internal spinlock
174         release(&m->splock);
175     }
```