- 1. Last time
- 2. Condition variables
- 3. Monitors and standards
- 4. Advice
- 5. Practice w/ concurrent programming
- 6. (preview) Implementation of locks: spinlocks, mutexes

---

2. CVs

A. Motivation

cond_init(&cv);

B. API

```
cond_init( Cond *, .. );
cond_wait( Cond *, Mutex* m);
cond_signal( Cond *, ... );
cond_broadcast( Cond *, ... );
```

C. Important points

- Must use "while" not "if"

```
while (_ _ _) {
    signal();
}
```

wait(); atomically { rel, sleep }
...
acquire();

- Cond.wait() releases mutex and goes into waiting state atomically; why?

alternative:

producer:
```
:
— release (&m);
— cond_wait(&cv);
— acquire (&m);
```

consumer:
```
acquire (&m);
:
cond_signal(&cv);
```
nonfull

- Can we replace signal() w/ broadcast()? Yes
- Can we replace broadcast() w/ signal()? No

Monitors and standards

3. Moni…

Monitor = <u>one</u> mutex + one or more CVs

```
M::f()
{
    acquire(&m);

    ...

    release(&m);
}
```

```
class M {
  private:
    Mutex
    CV cv1;
       cv2;
    ...
  public
    f();
    g();
}
```

```
M::g()
{
    acquire(&m);

    ...

    release(&m);
}
```

Commandments:

Rule: acquire/release at beginning/end of method or function.

Rule: hold lock when doing CV operations

Rule: a thread in wait() must be prepared to be restarted any time, not just when another thread calls signal(); ⟹ while (not safe to proceed) wait();

Rule: don't call sleep();

---

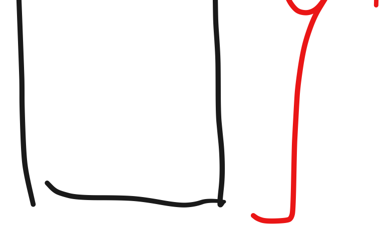
alloc/free example

alloc(1KB) wait()

alloc(1KB)

wait()

free



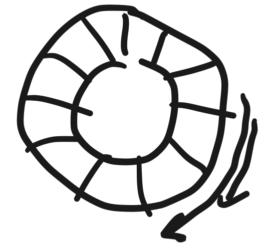
T. Hoare

B. Hansen

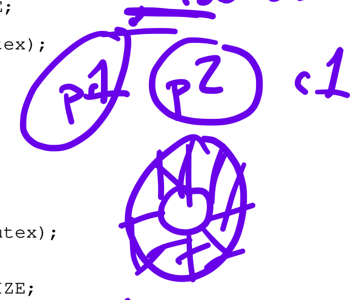
```
CS 202, Fall 2021
Handout 4 (Class 5)

The handout from the last class gave examples of race conditions. The following
panels demonstrate the use of concurrency primitives (mutexes, etc.). We are
using concurrency primitives to eliminate race conditions (see items 1
and 2a) and improve scheduling (see item 2b).

1. Protecting the linked list......

        Mutex list_mutex;

        insert(int data) {
            List_elem* l = new List_elem;
            l->data = data;

            acquire(&list_mutex);

            l->next = head;
            head = l;

            release(&list_mutex);
        }
```



```
2.  Producer/consumer revisited [also known as bounded buffer]

   2a. Producer/consumer [bounded buffer] with mutexes

     Mutex mutex;

     void producer (void *ignored) {
         for (;;) {
             /* next line produces an item and puts it in nextProduced */
             nextProduced = means_of_production();

             acquire(&mutex);
             while (count == BUFFER_SIZE) {
                release(&mutex);
                yield(); /* or schedule() */
                acquire(&mutex);
             }

             buffer [in] = nextProduced;
             in = (in + 1) % BUFFER_SIZE;
             count++;
             release(&mutex);
         }
     }

     void consumer (void *ignored) {
         for (;;) {

             acquire(&mutex);
             while (count == 0) {
                release(&mutex);
                yield(); /* or schedule() */
                acquire(&mutex);
             }

             nextConsumed = buffer[out];
             out = (out + 1) % BUFFER_SIZE;
             count--;
             release(&mutex);

             /* next line abstractly consumes the item */
             consume_item(nextConsumed);
         }
     }
```



```
2b. Producer/consumer [bounded buffer] with mutexes and condition variables

    Mutex mutex;
    Cond nonempty;
    Cond nonfull;

    void producer (void *ignored) {
        for (;;) {
            /* next line produces an item and puts it in nextProduced */
            nextProduced = means_of_production();

            acquire(&mutex);
            while (count == BUFFER_SIZE)
               cond_wait(&nonfull, &mutex);

            buffer [in] = nextProduced;
            in = (in + 1) % BUFFER_SIZE;
            count++;
            cond_signal(&nonempty, &mutex);
            release(&mutex);
        }
    }

    void consumer (void *ignored) {
        for (;;) {

            acquire(&mutex);
            while (count == 0)
               cond_wait(&nonempty, &mutex);

            nextConsumed = buffer[out];
            out = (out + 1) % BUFFER_SIZE;
            count--;
            cond_signal(&nonfull, &mutex);
            release(&mutex);

            /* next line abstractly consumes the item */
            consume_item(nextConsumed);
        }
    }


Question: why does cond_wait need to both release the mutex and
sleep? Why not:

    while (count == BUFFER_SIZE)  {
        release(&mutex);
        cond_wait(&nonfull);
        acquire(&mutex);
    }
```




```
2c.  Producer/consumer [bounded buffer] with semaphores

    Semaphore mutex(1);             /* mutex initialized to 1 */
    Semaphore empty(BUFFER_SIZE);   /* start with BUFFER_SIZE empty slots */
    Semaphore full(0);              /* 0 full slots */

    void producer (void *ignored) {
        for (;;) {
            /* next line produces an item and puts it in nextProduced */
            nextProduced = means_of_production();

           /*
            * next line diminishes the count of empty slots and
            * waits if there are no empty slots
            */
            sem_down(&empty);
            sem_down(&mutex);   /* get exclusive access */

            buffer [in] = nextProduced;
            in = (in + 1) % BUFFER_SIZE;

            sem_up(&mutex);
            sem_up(&full);   /* we just increased the # of full slots */
        }
    }

     void consumer (void *ignored) {
         for (;;) {

             /*
              * next line diminishes the count of full slots and
              * waits if there are no full slots
              */
             sem_down(&full);
             sem_down(&mutex);

             nextConsumed = buffer[out];
             out = (out + 1) % BUFFER_SIZE;

             sem_up(&mutex);
             sem_up(&empty);   /* one further empty slot */

             /* next line abstractly consumes the item */
             consume_item(nextConsumed);
         }
     }

    Semaphores *can* (not always) lead to elegant solutions (notice
    that the code above is fewer lines than 2b) but they are much
    harder to use.

    The fundamental issue is that semaphores make implicit (counts,
    conditions, etc.) what is probably best left explicit. Moreover,
    they *also* implement mutual exclusion.

    For this reason, you should not use semaphores. This example is
    here mainly for completeness and so you know what a semaphore
    is. But do not code with them. Solutions that use semaphores in
    this course will receive no credit.
```

CS 202, Fall 2021
Handout 5 (Class 6)

The previous handout demonstrated the use of mutexes and condition variables. This handout demonstrates the use of monitors (which combine mutexes and condition variables).

1. The bounded buffer as a monitor

```
    // This is pseudocode that is inspired by C++.
    // Don't take it literally.

    class MyBuffer {
      public:
        MyBuffer();
        ~MyBuffer();
        void Enqueue(Item);
        Item = Dequeue();
      private:
        int count;
        int in;
        int out;
        Item buffer[BUFFER_SIZE];
        Mutex* mutex;
        Cond* nonempty;
        Cond* nonfull;
    }

    void
    MyBuffer::MyBuffer()
    {
        in = out = count = 0;
        mutex = new Mutex;
        nonempty = new Cond;
        nonfull = new Cond;
    }

    void
    MyBuffer::Enqueue(Item item)
    {
        mutex.acquire();
        while (count == BUFFER_SIZE)
            cond_wait(&nonfull, &mutex);

        buffer[in] = item;
        in = (in + 1) % BUFFER_SIZE;
        ++count;
        cond_signal(&nonempty, &mutex);
        mutex.release();
    }

    Item
    MyBuffer::Dequeue()
    {
        mutex.acquire();
        while (count == 0)
            cond_wait(&nonempty, &mutex);

        Item ret = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        --count;
        cond_signal(&nonfull, &mutex);
        mutex.release();
        return ret;
    }
```



```
    int main(int, char**)
    {
        MyBuffer buf;
        int dummy;
        tid1 = thread_create(producer, &buf);
        tid2 = thread_create(consumer, &buf);

        // never reach this point
        thread_join(tid1);
        thread_join(tid2);
        return -1;
    }

    void producer(void* buf)
    {
        MyBuffer* sharedbuf = reinterpret_cast<MyBuffer*>(buf);
        for (;;) {
            /* next line produces an item and puts it in nextProduced */
            Item nextProduced = means_of_production();
            sharedbuf->Enqueue(nextProduced);
        }
    }

    void consumer(void* buf)
    {
        MyBuffer* sharedbuf = reinterpret_cast<MyBuffer*>(buf);
        for (;;) {
            Item nextConsumed = sharedbuf->Dequeue();

            /* next line abstractly consumes the item */
            consume_item(nextConsumed);
        }
    }
```

Key point: *Threads* (the producer and consumer) are separate from *shared object* (MyBuffer). The synchronization happens in the shared object.

2. This monitor is a model of a database with multiple readers and writers. The high-level goal here is (a) to give a writer exclusive access (a single active writer means there should be no other writers and no readers) while (b) allowing multiple readers. Like the previous example, this one is expressed in pseudocode.

```
    // assume that these variables are initialized in a constructor
    state variables:
        AR = 0;  // # active readers
        AW = 0;  // # active writers
        WR = 0;  // # waiting readers
        WW = 0;  // # waiting writers

        Condition okToRead = NIL;
        Condition okToWrite = NIL;
        Mutex mutex = FREE;

    Database::read() {
        startRead();  // first, check self into the system
        Access Data
        doneRead();   // check self out of system
    }

    Database::startRead() {
        acquire(&mutex);
        while((AW + WW) > 0){
            WR++;
            wait(&okToRead, &mutex);
            WR--;
        }
        AR++;
        release(&mutex);
    }

   Database::doneRead() {
        acquire(&mutex);
        AR--;
        if (AR == 0 && WW > 0) { // if no other readers still
          signal(&okToWrite, &mutex);   // active, wake up writer
        }
        release(&mutex);
    }

    Database::write(){  // symmetrical
        startWrite();  // check in
        Access Data
        doneWrite();  // check out
    }

    Database::startWrite() {
        acquire(&mutex);
        while ((AW + AR) > 0) { // check if safe to write.
                                // if any readers or writers, wait
            WW++;
            wait(&okToWrite, &mutex);
            WW--;
        }
        AW++;
        release(&mutex);
    }

    Database::doneWrite() {
        acquire(&mutex);
        AW--;
        if (WW > 0) {
            signal(&okToWrite, &mutex); // give priority to writers
        } else if (WR > 0) {
            broadcast(&okToRead, &mutex);
        }
        release(&mutex);
    }
```

NOTE: what is the starvation problem here?

3. Shared locks

```
    struct sharedlock {
      int i;
      Mutex mutex;
      Cond c;
    };

    void AcquireExclusive (sharedlock *sl) {
      acquire(&sl->mutex);
      while (sl->i) {
        wait (&sl->c, &sl->mutex);
      }
      sl->i = -1;
      release(&sl->mutex);
    }

    void AcquireShared (sharedlock *sl) {
      acquire(&sl->mutex);
      while (sl->i < 0) {
        wait (&sl->c, &sl->mutex);
      }
      sl->i++;
      release(&sl->mutex);
    }

    void ReleaseShared (sharedlock *sl) {
      acquire(&sl->mutex);
      if (!--sl->i)
        signal (&sl->c, &sl->mutex);
      release(&sl->mutex);
    }

    void ReleaseExclusive (sharedlock *sl) {
      acquire(&sl->mutex);
      sl->i = 0;
      broadcast (&sl->c, &sl->mutex);
      release(&sl->mutex);
    }
```

QUESTIONS:

A. There is a starvation problem here. What is it? (Readers can keep writers out if there is a steady stream of readers.)

B. How could you use these shared locks to write a cleaner version of the code in the prior item? (Though note that the starvation properties would be different.)

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.

```
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
1  #include <sys/queue.h>
2
3  typedef struct thread {
4      // ... Entries elided.
5      STAILQ_ENTRY(thread_t) qlink; // Tail queue entry.
6  } thread_t;
7
8  struct Mutex {
9      // Current owner, or 0 when mutex is not held.
10     thread_t *owner;
11
12     // List of threads waiting on mutex
13     STAILQ(thread_t) waiters;
14
15     // A lock protecting the internals of the mutex.
16     Spinlock splock;   // as in item 1, above
17 };
18
19 void mutex_acquire(struct Mutex *m) {
20
21     acquire(&m->splock);
22
23     // Check if the mutex is held; if not, current thread gets mutex and returns
24     if (m->owner == 0) {
25         m->owner = id_of_this_thread;
26         release(&m->splock);
27     } else {
28         // Add thread to waiters.
29         STAILQ_INSERT_TAIL(&m->waiters, id_of_this_thread, qlink);
30
31         // Tell the scheduler to add current thread to the list
32         // of blocked threads. The scheduler needs to be careful
33         // when a corresponding sched_wakeup call is executed to
34         // make sure that it treats running threads correctly.
35         sched_mark_blocked(&id_of_this_thread);
36
37         // Unlock spinlock.
38         release(&m->splock);
39
40         // Stop executing until woken.
41         sched_swtch();
42
43         // When we get to this line, we are guaranteed to hold the mutex. This
44         // is because we can get here only if context-switched-TO, which itself
45         // can happen only if this thread is removed from the waiting queue,
46         // marked "unblocked",/ and set to be the owner (in mutex_release()
47         // below). However, we might actually have held the mutex in lines 39-42

48         // (if we were context-switched out after the spinlock release(),
49         // followed by being run as a result of another thread's release of the
50         // mutex). But if that happens, it just means that we are
51         // context-switched out an "extra" time before proceeding.
52     }
53 }
54
55 void mutex_release(struct Mutex *m) {
56     // Acquire the spinlock in order to make changes.
57     acquire(&m->splock);
58
59     // Assert that the current thread actually owns the mutex
60     assert(m->owner == id_of_this_thread);
61
62     // Check if anyone is waiting.
63     m->owner = STAILQ_GET_HEAD(&m->waiters);
64
65     // If so, wake them up.
66     if (m->owner) {
67         sched_wakeone(&m->owner);
68         STAILQ_REMOVE_HEAD(&m->waiters, qlink);
69     }
70
71     // Release the internal spinlock
72     release(&m->splock);
```

```
}
```