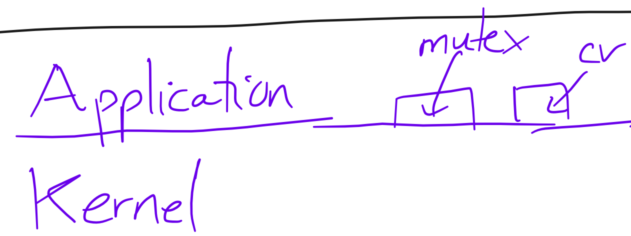
- [x] 1. Last time
- [x] 2. Condition variables
- [x] 3. Monitors and standards
- [x] 4. Advice
- [ ] 5. Practice with concurrent programming

ONE HANDOUT

---

2. Condition variables



A. Motivation

B. API

```
cond_init (Cond *, ...);

cond_wait (Mutex *m, Cond *);

cond_signal (Mutex *m, Cond *);

cond_broadcast (Mutex *m, Cond *);
```

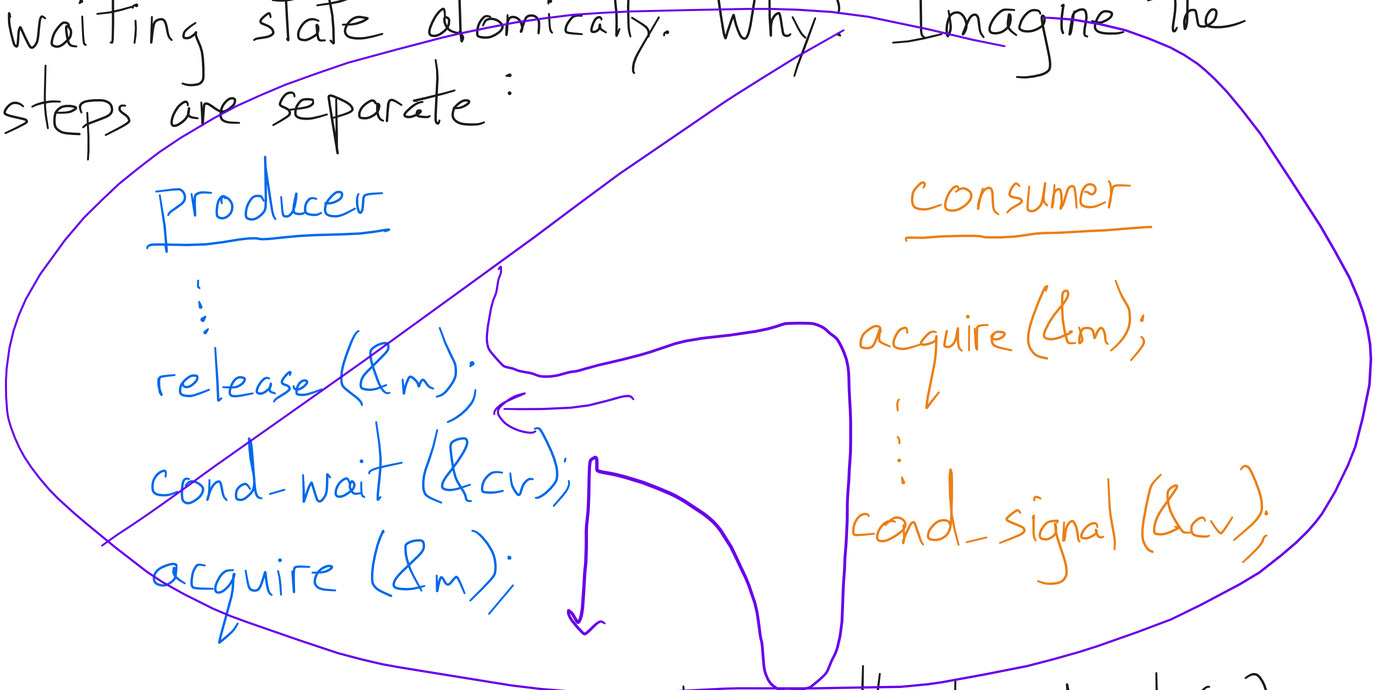
C. Most important point about usage:

MUST [illegible] "[illegible]" [illegible] "if" [illegible]

MUST use while not if when waiting.

D. Other aspects

(1) cond_wait() releases mutex and goes into waiting state atomically. Why? Imagine the steps are separate:



(2) Can we replace signal() with broadcast()?

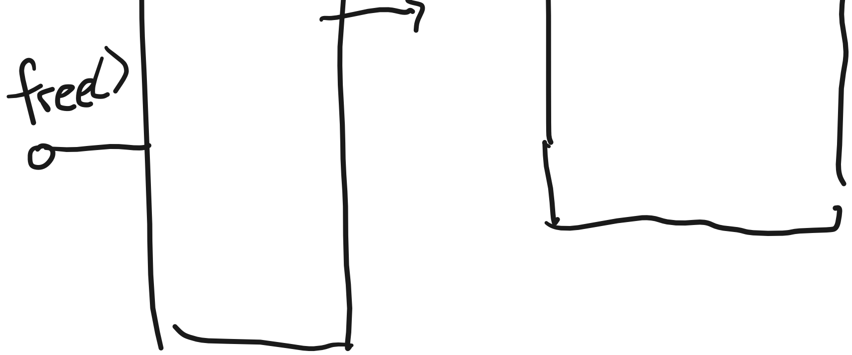
(3) Can we replace broadcast() with signal()?

---

alloc/free example

alloc:

while (not enuf mem)
    wait();



free:
    signal() ?

3. Monitors and standards

Monitor = one mutex + one or more CVs.

The pattern:

class Mon {
  private:
    Mut...

Mon :: f()
{
    acquire(&m);

```
Mutex m;
Cond cv1;
Cond cv2;
  :
public:
  f();
  g();
}
```

```
acquire(&m);
  :
  release(&m);
}

Mon::g()
{
  acquire(&m);
    :
  release(&m);
}
```

Example: see handout

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();

---

4. Advice

1. Getting started
   - 1a. identify units of concurrency
   - 1b. identify chunks of state
   - 1c. write down high-level main loop of each thread

   separate threads from objects

2. write down the synchronization constraints, and the kind (mutual exclusion or scheduling)

   Ex:
   - ~~sync~~ mutual excl. constraint: only one thr. interacts w/ shared state
   - Consumer cannot proceed if buf is empty
   - Producer " " if buf is full

3. create a lock or CV for each constraint
   - mutex
   - Cond notempty

· Cond not full

4. write the methods, using the locks and CVs

---

5. Practice

Example:

- workers interact with a database
- readers never modify
- writers read and modify
- single mutex would be too restrictive
- instead, want:
  - many readers at once OR
  - only one writer (and no readers)

Let's follow the advice:

a. units of concurrency?

b. shared chunks of state?

c. what does main function look like?

read()

check in .... wait until no writers

access DB()

access_DB()
check out ... wake waiting writers, if any

write()
check in ... wait until no one else
access_DB()
check out ... wake up waiting readers or writers

2. and 3. synchronization constraints and synchronization objects

4. write the methods

int AR = 0; // active readers

```
int AR = 0;  // active readers
int AW = 0;  // active writers
int WR = 0;  // waiting readers
int WW = 0;  // waiting writers
```

CS 202, Fall 2024
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 2024
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.)