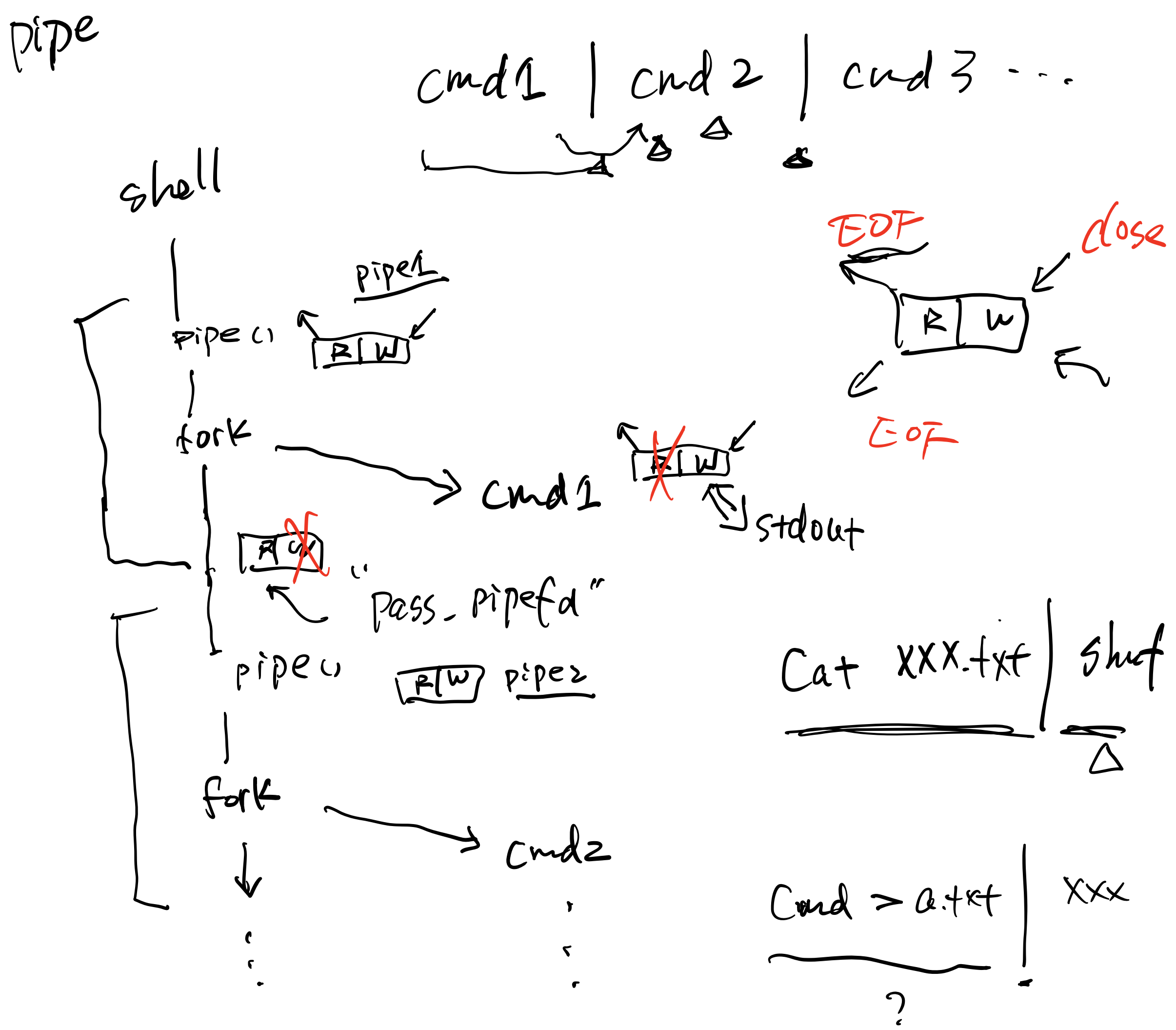
1. Lab2
2. Condition variables
3. Semaphores
4. Monitors and standards
5. Advice for concurrent programming



mutex ≈ lock

C.V.

cond_init (cond *)

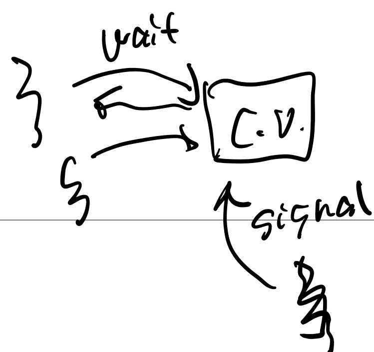
cond_wait (cond *c , mutex *m)

① unlock mutex (release) , ② wait/sleep on C signal ③ acquire mutex. ④ if succ, resume


```
CS5600 Week5.b

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);
   }
```

cond_signal(cond*)

cond_broadcast(cond*)



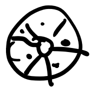

```
2. Producer/consumer revisited [also known as bounded buffer]

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);
      }
    }
```

1

100

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);
    }
```

buffer full

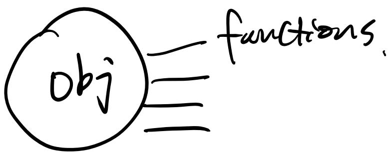
Consumer
acquire(mutex)
signal (nonfull)
//finish

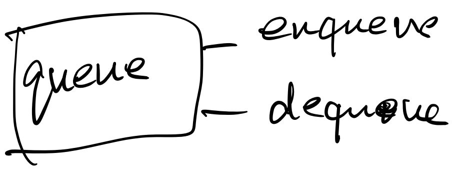
wait
FOREVER

X/OO

Monitor = mutex + C.V.

Monitor
6 Rules
4 steps.

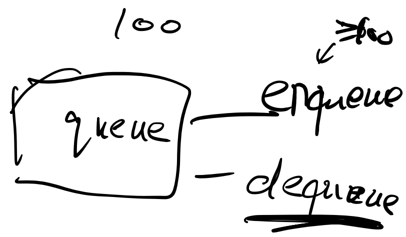




0 C.V.

↳ limited-queue.

Java. Synchronized



1 C.V.

⇓

2 C.V.

CS 5600, week 6.b

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.

Rules

--acquire/release at beginning/end of methods

--hold lock when doing condition variable operations

--always use "while" to check invariants, not "if"

-- …

while (cond)
    cond_wait(&c.v, m);

signal/broadcast

pr/con

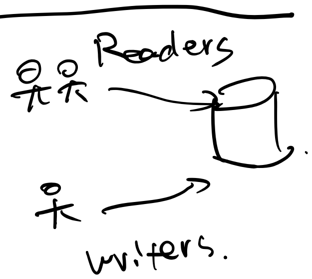
1. Getting started:
   1a. Identify units of concurrency. ← Producer Consumer
   1b. Identify shared chunks of state. buffer
   1c. Write down the high-level main loop of each thread. ←
2. Write down the synchronization constraints on the solution.
3. Create a lock or condition variable corresponding to each constraint
4. Write the methods, using locks and condition variables for coordination

producer
check buffer
if no full:
enqueue

2. ① ~~mutual~~ ex → pr/con
② ~~buffer~~ full → pr should wait
③ empty → Con . . .

3. ① → multex ② → nonfull ③ → nonempty

4.



R-R V
W-W X
R-W X

R: access_db1
W: access_db2

R:
Check if writer
if no writer:
access

W:
check if W, R
if nobody:
access

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();
  }

  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);
    }
    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?

handout w6b 2/22/22. 9:52 PM

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.)