CSc 360
Operating Systems
Mutual Exclusion

Wenjun Yang

Fall 2025

1st Practice Exam Released!
Solution will release this Thur.

Plz join the shared teams!
Emails me bonus request and cc TAs: Johnathan and Clark.

Review: threads

- Thread vs process
  - easy to share info btw threads in one process
    - share and protect!
- Create and terminate threads
  - start routine, argument passing
    - threads are executed “in parallel”
- Join and detach threads
  - synchronization
    - “wait”

Shared or not shared?

- E.g., increment a **counter** (shared variable)
  - read the counter (from memory)
  - increment by one (at CPU)
  - write the counter
- How about two threads?
  - *sharing* only one counter
    - non-deterministic result: $R_1W_1R_2W_2$; $R_1R_2W_1W_2$
- "There is something not to be (always) shared"

* producer-consumer problem with a buffer and counter

What will be the final answer?

```
pthread_t tid[2];
int counter = 0;
void* doSomething(void *arg) {
  unsigned long i = 0;
  for (int i = 0 ; i < 1000 ; i++) {
    counter += 1;
  }
  return NULL;
}
int main(void) {
  for(int i = 0; i < 2; i++) {
    pthread_create(&(tid[i]), NULL, &doSomething, NULL);
  }
  pthread_join(tid[0], NULL);
  pthread_join(tid[1], NULL);
  printf("Counter %d \n", counter);
  return 0;
}
```

```
 matei@laptop> gcc concurrency.c -o
concurrency –pthread
```

```
 matei@laptop> ./concurrency
```

```
Counter 2000
```

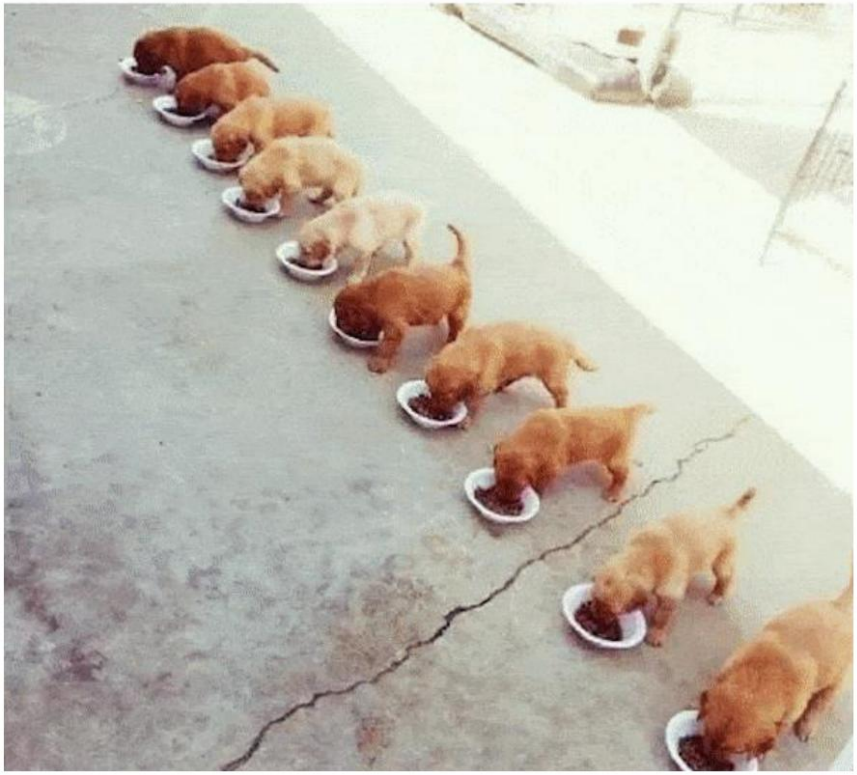
```
 matei@laptop> ./concurrency
```

```
Counter 1937
```

```
 matei@laptop> ./concurrency
```

```
Counter 1899
```

Multi-threaded programming in theory:



Multi-threaded programming in reality:

Why?

- Protection is at the process level.
- Threads not isolated.
  - Share an address space and data (e.g., counter).

So interleaving of threads is non-deterministic, and thus inconsistent outputs.

Definitions

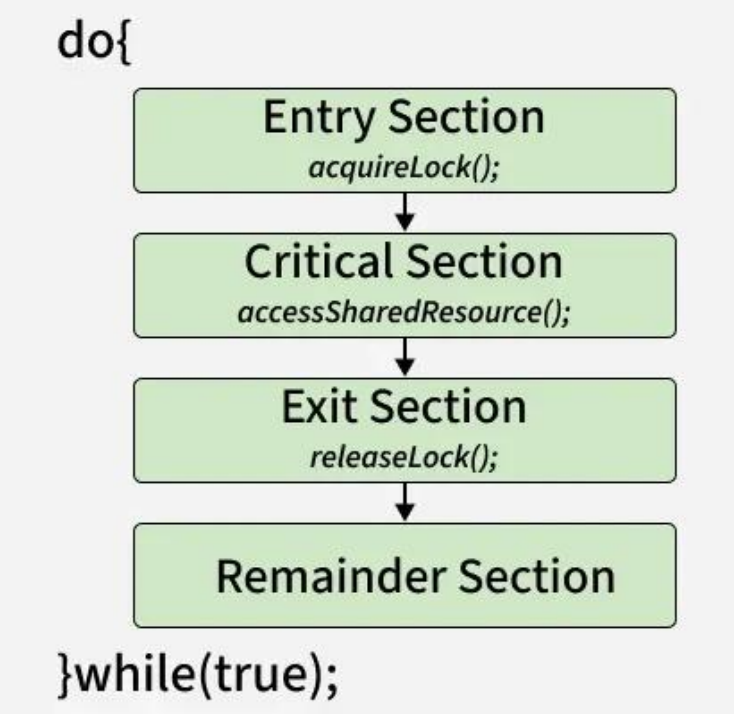
- Synchronization
  - Using atomic operations to ensure cooperation between threads
- Critical Section
  - Piece of code that only one thread can execute at once. Only one thread at a time will get into this section of code
- Mutual Exclusion
  - Ensuring that only one thread does a particular thing at a time

Atomic Operations

- An operation that always runs to completion, or not at all
- It is indivisible: it cannot be stopped in the middle, and state cannot be modified by someone else in the middle
- Fundamental building block
  - If no atomic operations, then have no way for threads to work together
- On most machines, memory references and assignments (i.e. loads and stores) of words are atomic
- Many instructions are not atomic, e.g., double-precision floating point store

Critical section

- Critical section
  - code section accessing shared data
  - only one thread executing in critical section
  - choose the **right** (size of) critical section!



Critical section

- Approach: exclusion (lock)
  - Prevents someone from doing something
  - **Lock()** before entering critical section and before accessing shared data
  - **Unlock()** when leaving, after accessing shared data
  - **Wait** if locked

Mutual exclusion

- Mutual exclusion (mutex)
  - only two states
    - unlocked: there is no thread in critical section
    - locked: there is one thread in critical section
  - state change is atomic
    - if it is unlocked, it can be locked by **at most one** thread when entering the critical section
    - if it is locked, it can “only” be unlocked by the locking thread when leaving the critical section

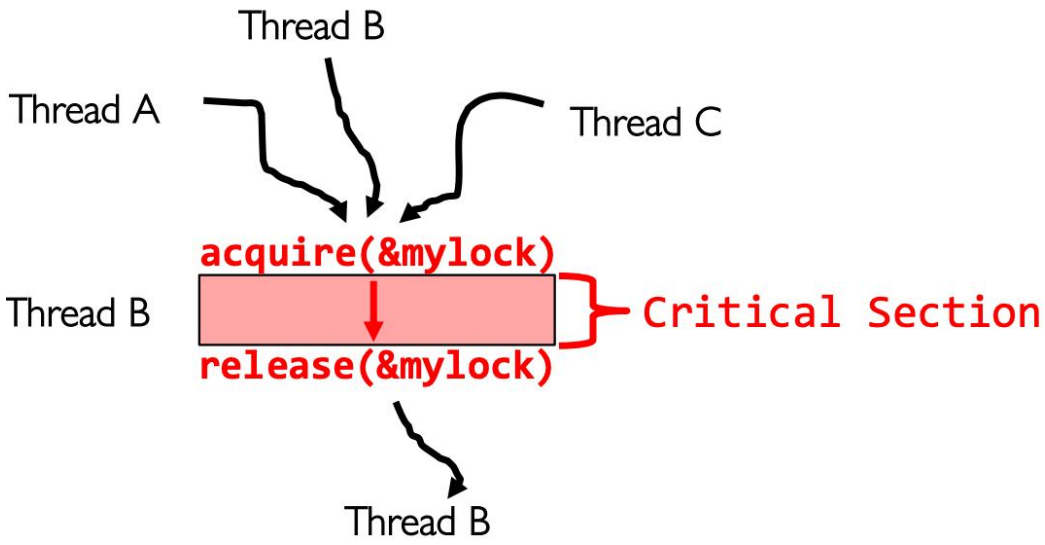
Example: Bank deposit with locks

- Identify critical sections (atomic instruction sequences) and add locking

```
Deposit(acctId, amount) {
  acquire(&mylock)          // Wait if someone else in critical section!
  acct = GetAccount(actId);
  acct->balance += amount;
  StoreAccount(acct);
  release(&mylock)          // Release someone into critical section
}
```

Critical Section

Example: Bank deposit with locks



Threads serialized by lock through critical section

Only one thread at a time

Example:“Too Much Milk”

Time	Person A	Person B
3:00	Look in Fridge. Out of milk	
3:05	Leave for store	
3:10	Arrive at store	Look in Fridge. Out of milk
3:15	Buy milk	Leave for store
3:20	Arrive home, put milk away	Arrive at store
3:25		Buy milk
3:30		Arrive home, put milk away

Too Much Milk: Correctness Properties

What are the correctness properties for the “Too much milk” problem???

- Never more than one person buys
- Someone buys if needed

First attempt: Restrict ourselves to use only atomic load and store operations as building blocks

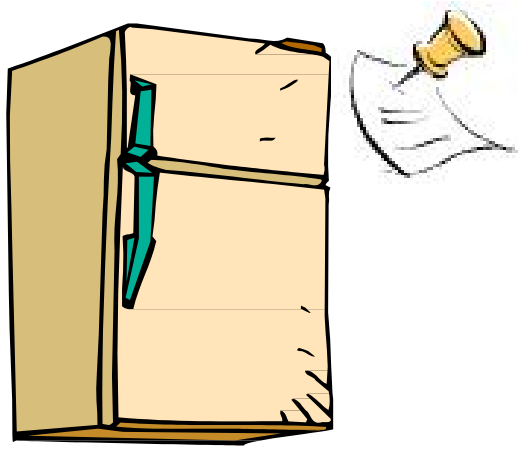
Too Much Milk: Solution #1

Use a note to avoid buying too much milk:

- Leave a note before buying (kind of “lock”)
- Remove note after buying (kind of “unlock”)
- Don’t buy if note (wait)

Suppose a computer tries this
(remember, only memory read/write are atomic)

```
if   (noMilk)   {
     if   (noNote)
     {
          leave   Note;
          buy  milk;
          remove note;
     }
}
```



Too Much Milk: Solution #1

Thread A

```
if  (noMilk) {
    if  (noNote) {
        leave Note;

        buy Milk;
        remove Note;
}   }
```

Thread B

```
if  (noMilk) {
    if  (noNote) {

        leave Note;
        buy Milk;
        remove Note;
    }
}
```

Too Much Milk: Solution #1

Still too much milk but only occasionally!

Thread can get context switched after checking milk and note but before leaving notes!

Too Much Milk: Solution #2

Let's try to fix this by placing note first

```
leave  Note;
if  (noMilk)  {
    if  (noNote)  {
        buy  milk;
    }
}
remove Note;
```

What happens here?

– Well, with human, probably nothing bad

– With computer: no one ever buys milk

Too Much Milk: Solution #3

Recall our target lock interface:
- acquire(&milklock) – wait until lock is free, then grab
- release(&milklock) – Unlock, waking up anyone waiting
– These must be atomic operations – if two threads are waiting for the lock and both see it's free, only one succeeds to grab the lock

Then, our milk problem is easy:

```
acquire(&milklock);
if (nomilk)
    buy milk;
release(&milklock);
```

Mutex: more

- Mutex procedures
  - create a mutex variable (initially unlocked)
  - (some threads) attempt to lock the mutex
    - only one can lock the mutex
      - others may be blocked and waiting
    - the one with the mutex
      - execute the critical section
      - unlock the mutex variable eventually
  - destroy the mutex variable

Mutex with pthread

- Create mutex
  - int pthread_mutex_init (mutex, attributes);
- Attempt to lock
  - int pthread_mutex_lock (mutex);
    - if unlocked, lock and return immediately
    - if locked
      - “fast” lock: blocked until the mutex is unlocked
      - “test” lock: return immediately with error
      - “recursive” lock: “over”-lock
        - » multiple pthread_mutex_unlock() to unlock

https://computing.llnl.gov/tutorials/pthreads/

Mutex with pthread: more

- Try to lock
  - int pthread_mutex_trylock (mutex);
    - if locked, return immediately with error code
- Unlock
  - int pthread_mutex_unlock (mutex);
    - if “recursive” lock, multiple pthread_mutex_unlock necessary to fully unlock the mutex
- Destroy mutex
  - int pthread_mutex_destroy (mutex);

Condition variable

- Used **together** with mutex
  - mutex: control access to shared data
  - condition: synchronize by condition “predict”
- Wait for condition
  - pthread_cond_wait (condition, mutex);
    - automatically unlock and wait for signal
    - on signal, wake up and automatically lock
- Signal or broadcast
  - pthread_cond_signal (condition);

Example: mutex and condition

- Main thread
  - global variable
  - create mutex and condition variable

- Wait to be signaled
  - pthread_mutex_lock();
  - pthread_cond_wait();
  - … /* run me next */
  - pthread_mutex_unlock();

- Send the signal
  - pthread_mutex_lock();
  - … /* run me first */
  - pthread_cond_signal();
  - pthread_mutex_unlock();

This lecture

- Mutex and condition
  - mutex: binary access control
    - locked or unlocked
  - condition: access control by condition
    - used together with mutex
- Explore further
  - more details in

    http://computing.llnl.gov/tutorials/pthreads/

    * also a tutorial coming soon

Next half of the lecture

- CPU scheduling
  - Another chapter in the textbook