CS202 (003): Operating Systems
Concurrency II

Instructor: Jocelyn Chen

Most of the materials covered in this slide come from the lecture notes of Mike Walfish's CS202

Last time

Managing Concurrency: the Key Problem

How do we avoid multiple **threads** accessing **a shared resource** at the **same time**?

A piece of code that access a shared resource and must not be concurrently executed by more than one thread is called a
Critical Section

How do we *protect* Critical Sections from concurrent execution?

Three (ideal) Properties of the Solution

Mutual Exclusion/Atomicity
Only one thread can be in critical section at a time

Progress
If no thread is executing in critical section, then one of the threads trying to enter a given critical section will eventually get in

Bounded Waiting
Once a thread T starts trying to enter the critical section, there is a bound on the number of other threads that may enter the critical section before T enters

So, what is the solution?

Key Idea

Once the thread of execution is *executing inside the critical section*, **no other** thread of execution is executing there

```
lock()/unlock()
enter()/leave()
acquire()/release()
```

They all illustrate the same idea!

```
mutex_init(mutex_t* m)
mutex_lock(mutex_t* m)
mutex_unlock(mutex_t* m)
```

Mutex (mutual exclusion objects)

```
pthread_mutex_init(…)
pthread_mutex_lock(…)
pthread_mutex_unlock(…)
```

POSIX Thread (pthread) Functions

How to implement these solutions?

“Easy” Implementation (on uniprocessor)
enter( ) -> disable interrupts
leave ( ) -> re-enable interrupts

This prevents CPU from switching to another thread when the current thread is exciting its critical section

We will study other implementation later!

Look at your new handout!

```
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);
}
```

Look at your new handout!

```
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);
    }
}
```

Use of Mutex

Once we have mutex, we don’t have to worry about arbitrary interleaving

Because mutex allows us maintain certain **type of invariants:**

LinkedList	*Only one thread can be modifying the head of the list*
Producer/Consumer	*The 'count' accurately represents the number of items in the buffer*

Going back to the Producer/Consumer example

What is the problem of using mutex?

Producer/Consumer keep checking the buffer state when it is full/empty

Two types of synchronization

Mutual Exclusion	updating the count variable
Scheduling Constraint: Wait for some other thread to do sth	waiting the buffer to have/empty something