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!

```c
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);
    }
}
```

```c
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

Condition Variables

Warning: Condition Variable is not really a Variable!

```
void cond_init(Cond *cond, ...);
void cond_wait(Cond *cond, Mutex *mutex);
void cond_signal(Cond *cond);
void cond_broadcast(Cond *cond);
```

```
mutex_lock(&mutex);
while (!condition_is_met) {
    cond_wait(&cond, &mutex);
}
// Modify shared state
mutex_unlock(&mutex);
```

Why is this a while?

Condition Variables

Warning: Condition Variable is not really a Variable!

```
void cond_init(Cond *cond, ...);
void cond_wait(Cond *cond, Mutex *mutex);
void cond_signal(Cond *cond);
void cond_broadcast(Cond *cond);
```

```
mutex_lock(&mutex);
while (!condition_is_met) {
    cond_wait(&cond, &mutex);
}
// Modify shared state
mutex_unlock(&mutex);
```

This **MUST** be a while!

More hypothetical questions…

Why do cond_wait releases the mutexes and goes into the waiting state in one function call (see panel 2b of handout 04)?

If those two steps were separate, could get stuck waiting.

```
Producer: while (count == BUFFER_SIZE)
Producer: release()
Consumer: acquire()
Consumer: .....
Consumer: cond_signal(&nonfull)
Producer: cond_wait(&nonfull)
```

Producer never hears the signal!

More hypothetical questions…

Can we replace SIGNAL with BROADCAST, and preserve correctness*?

Yes, but it might hurt performance

```
Since while() checks the invariant,
Only thread satisfying the invariant will make progress

=> this does not affect correctness


But we make needlessly wakeup of threads

=> this might hurt performance
```

correctness*: not having race conditions, and making progress when possible

More hypothetical questions…

Can we replace BROADCAST with SIGNAL, and preserve correctness*?

No race conditions, but may never make progress

correctness*: not having race conditions, and making progress when possible

Monitor: Mutex + Conditional Variables (but in OOP)

All method calls of a class are protected by a **mutex**

Synchronization happens with condition variables whose associated mutex is the **mutex that protects the method calls**

"Monitor" can be used to refer to either a *programming convention* or a *method in certain programming languages**

* https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html

What does monitor enable us to do?

Encapsulation!

Separation of program logic inside threads from the shared object

The monitor handles all synchronization internally so threads don't need to worry about locking, unlocking or conditional signaling

Look at the first page of handout05!

Producer/Consumer w/ Monitor

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

void producer(void* buf)
{
    MyBuffer* sharedbuf = reinterpret_cast<MyBuffer*>(buf);
    for (;;) {
        Item nextProduced = means_of_production();
        sharedbuf->Enqueue(nextProduced);
    }
}

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

Producer/Consumer w/ Mutex & CV

```
Mutex mutex;

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

     consume_item(nextConsumed);
    }
}
```

* These are pseudocode. Class is a special data type used for OOP.

Semaphores: Mutex + Conditional Variables (but more general)

```
#include <semaphore.h>
sem_t s;
sem_init(&s, 0, 1);

int sem_wait(sem_t *s) {
  decrement the value of semaphore s by one
  wait if value of semaphore s is negative
}

int sem_post(sem_t *s) {
  increment the value of semaphore s by one
  if there are one or more threads waiting, wake one
}

sem_wait(&m);
// critical section here
sem_post(&m);
```

Semaphores: Mutex + Conditional Variables (but more general)

Semaphores manage a count, mutex+CV do not inherently do this

Semaphores can allow multiple threads access, unlike a basic mutex

Semaphores can be used for locking, but can also be used for other purpose

DO NOT USE SEMAPHORE IN THIS CLASS!