CS202 (003): Operating Systems
Concurrency III

Instructor: Jocelyn Chen

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

Last Time

Mutex (mutual exclusion objects)

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

Conditional Variables

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

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

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.

Monitor: Mutex + Conditional Variables

All method calls 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**

Please follow these conventions on Lab 3!

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

Standards for Programming w/ Threads

Rule I: acquire/release at beginning/end of methods

Rule II: hold lock when doing condition variable operations

Rule III: a thread that is in wait() must be prepared to be restarted at any time, not just when another thread calls "signal()"

Rule IV: don't call sleep()

Advice for concurrent programming

Top-level piece of advice: SAFETY FIRST

Locking at coarse grain is easiest to get right, so do that

Don’t worry about performance at first

Don’t view deadlock as a disaster

MAKE SURE YOU PROGRAM NEVER DOES THE WRONG THING

Advice for concurrent programming

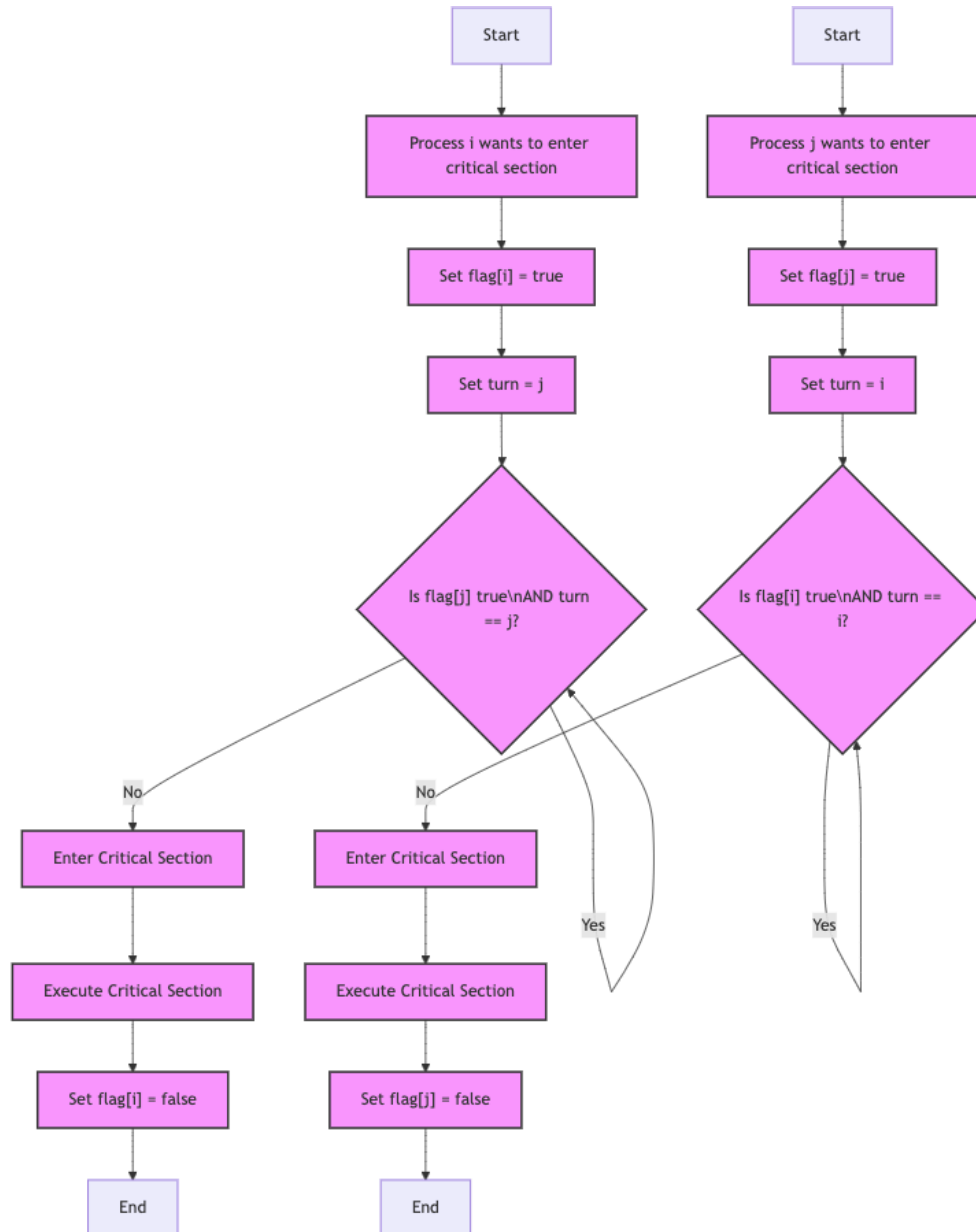
Getting started

1. Identify unit of concurrency
2. Identify chunks of state
3. write down high-level main loop of each thread

Write down the synchronization constraints, and the type

Create a lock or CV for each constraint

Implement the methods, using the locks and CVs



Implementation of mutex

Peterson's algorithm

expensive (busy waiting)
requires number of threads to be fixed statically
assumes sequential consistency

Implementation of mutex

Disable Interrupts

Works only on a single CPU
Cannot expose to user processes

Implementation of mutex

Spinlock