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

Revisiting the Handout

```
struct List_elem {
    int data;
    struct List_elem* next;
};

List_elem* head = 0;

insert(int data) {
    List_elem* l = new List_elem;
    l->data = data;
    l->next = head;
    head = l;
}
```

* this is pseudocode

What happens if two threads execute insert() at once and we get the following interleaving?

```
thread 1: l->next = head
thread 2: l->next = head
thread 2: head = l;
thread 1: head = l;
```

The list is broken!

Revisiting the Handout

```
void producer (void *ignored) {
    for (;;) {
        /* next line produces an item and puts it in nextProduced */
        nextProduced = means_of_production();
        while (count == BUFFER_SIZE)
            ; // do nothing
        buffer [in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
    }}

void consumer (void *ignored) {
    for (;;) {
        while (count == 0)
            ; // do nothing
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;
        /* next line abstractly consumes the item */
        consume_item(nextConsumed);
}}
```

* this is pseudocode

```
assuming count++ compiles to:
reg1 <- count # load
reg1 <- reg1 + 1 # increment register
count <- reg1 # store

assuming count-- compiles to:
reg2 <- count # load
reg2 <- reg2 – 1 # decrement register
count <- reg2 # store
```

What happens if we get the following interleaving?

```
reg1 <- count
reg1 <- reg1 + 1
reg2 <- count
reg2 <- reg2 - 1
count <- reg1
count <- reg2
```

The count is incorrect!

We call these situation a *race condition*

Or more specifically, a data race

It arises if multiple threads of execution enter the critical section at roughly the same time; both attempt to update the shared data structure, leading to a surprising (and perhaps undesirable) outcome.

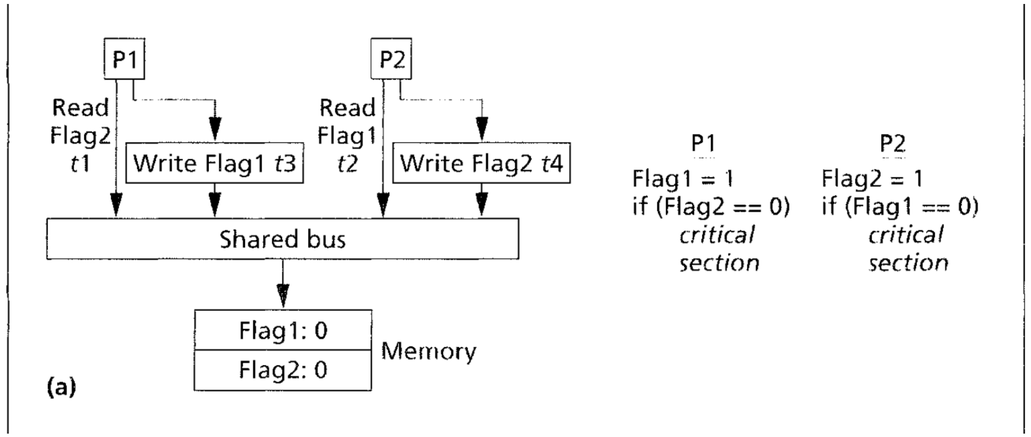
Revisiting the Handout

```
int flag1 = 0, flag2 = 0;

int main () {
    tid id = thread_create (p1, NULL);
    p2 (); thread_join (id);
}

void p1 (void *ignored) {
    flag1 = 1;
    if (!flag2) {
        critical_section_1 ();
    }
}

void p2 (void *ignored) {
    flag2 = 1;
    if (!flag1) {
        critical_section_2 ();
    }
}
```

* this is pseudocode

Can both "critical sections" run?

Maybe, if the hardware works like the following:



https://sadve.cs.illinois.edu/Publications/computer96.pdf

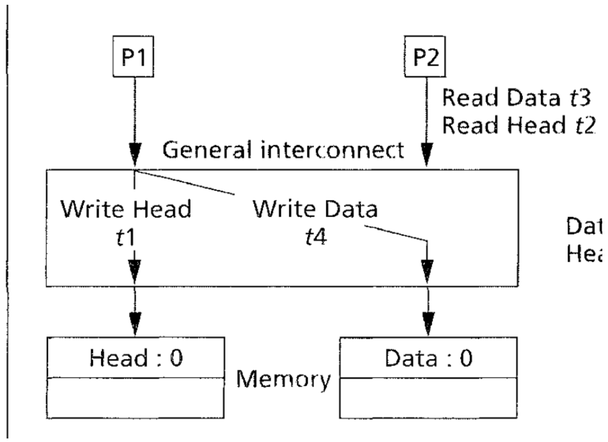
Revisiting the Handout

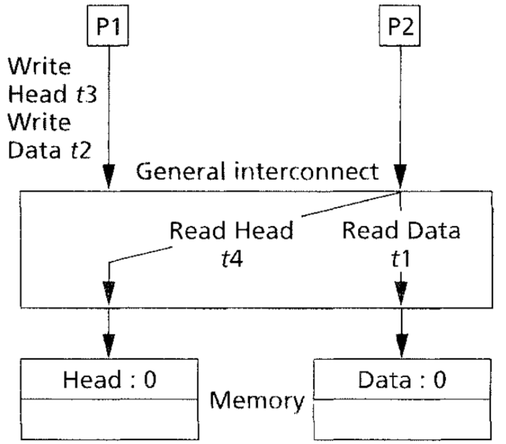
```
int data = 0, head = 0;

void p1 () {
    data = 2000;
    head = 1;
}

int p2 () {
    while (!head) {}
    use(data);
}
```

* this is pseudocode

Can use() be called with value 0, if p2 and p1 run concurrently?

Maybe, if the hardware works like the following:





https://sadve.cs.illinois.edu/Publications/computer96.pdf

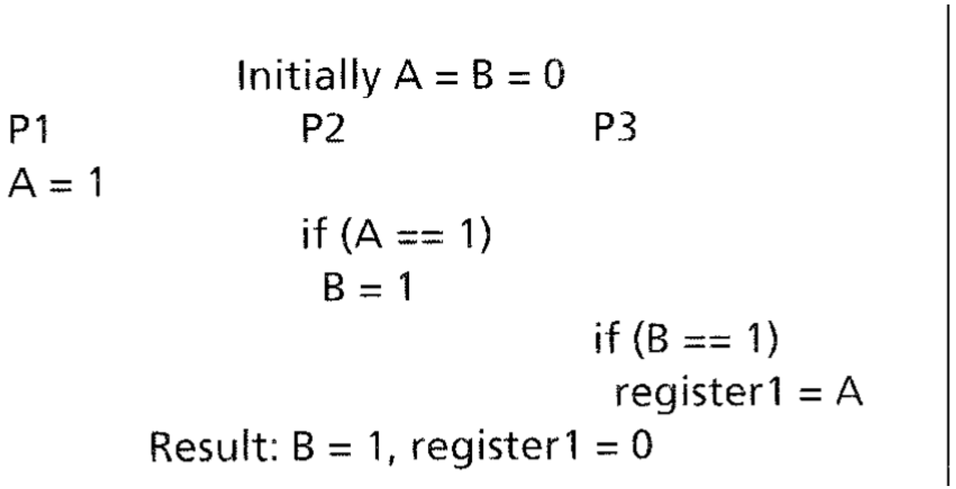
Revisiting the Handout

```
int a = 0, b = 0;

void p1 (void *ignored) { a = 1; }

void p2 (void *ignored) {
    if (a == 1) b = 1;
}

void p3 (void *ignored) {
    if (b == 1) use (a);
}
```

* this is pseudocode

Can use() be called with value 0?

Maybe, if the hardware works like the following:



Certain hardware allows P2 to return the value of P1's write before the write is visible to P3

Reasoning about Concurrency is Hard!

https://sadve.cs.illinois.edu/Publications/computer96.pdf

Don’t worry about hardware-related issues, for now

(Unless explicitly relax it) We assume **sequential consistency** in this class

(On each individual processors)
Writes to each memory location happen in the order that they are issued

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

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

Conditional Variables

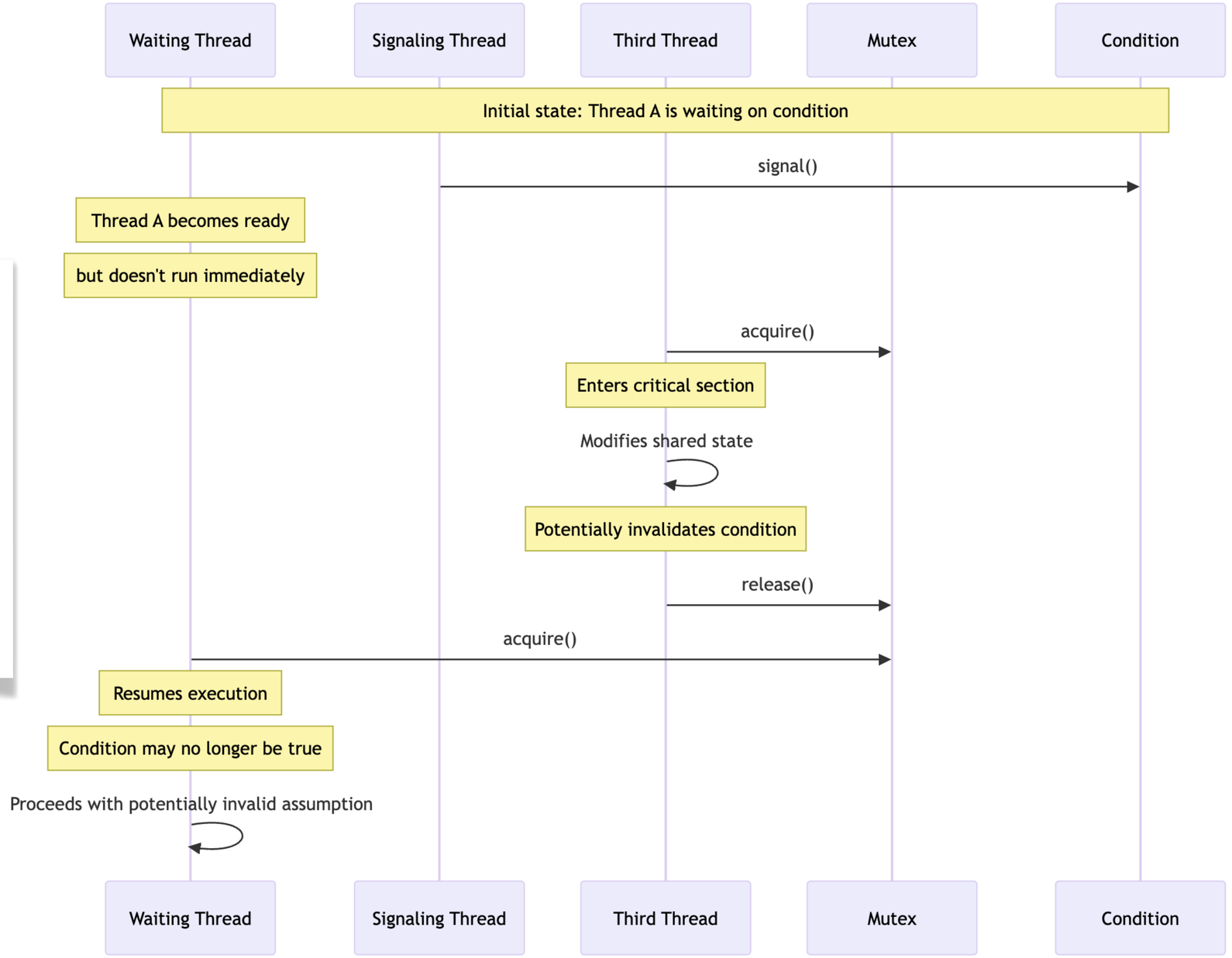
Warning: Conditional Variable is not really a Variable!

```
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?

```
mutex_lock(&mutex);

if (!condition_is_met) {
    cond_wait(&cond, &mutex);
}

// Proceed with the assumption
// that the condition is met
// Perform operations based on this
assumption
// ...

mutex_unlock(&mutex);
```



Conditional Variables

Warning: Conditional 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

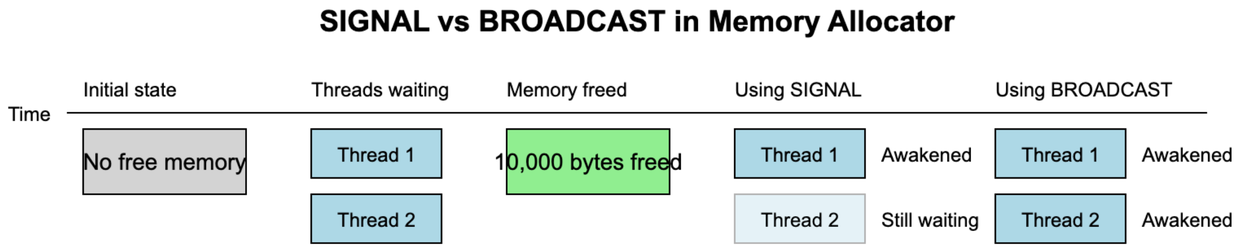
```
Since while() checks the invariant,
Only thread satisfying the invariant will make progress

=> 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



Explanation:

1. Initially, there's no free memory and two threads are waiting to allocate memory.
2. A third thread frees 10,000 bytes of memory.
3. With SIGNAL, only one thread is awakened, potentially leaving memory unused.
4. With BROADCAST, both threads are awakened, allowing both to attempt allocation.
5. BROADCAST ensures better resource utilization and prevents potential thread starvation.

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