Overview

- Concurrent programming
- Race conditions
- Mutual exclusion
Concurrent Programming

- Programming with two or more concurrent threads that cooperate to perform a common task
- Threads cooperate by accessing shared data

```c
int A[1000];

main() {
    thread_create(T1);
    thread_create(T2);
    main_loop();
}

T1() {
    int v1;
    A[0] = v1;
}

T2() {
    int v1, v2;
    v2 = A[0];
}
```
Problems with Concurrent Programming

- **Race Conditions**
  - Certain thread interleavings cause incorrect program behavior
  - E.g., Two threads T1 and T2 read and update the same variable

- **Synchronization**
  - Need to ensure operations are performed in certain order
  - E.g., T1 initializes a variable, T2 reads initialized variable

```
T1() {
    int v1;
    A[0] = v1;
}
```

```
T2() {
    int v1, v2;
    v2 = A[0];
}
```
Race Condition Example

worker() { ...; counter = counter + 1; ... }

Dump of assembler code for worker():
// 0x406018 is address of counter variable
mov 0x406018, %eax ; 1. read from mem into eax
add $0x1, %eax ; 2. increment eax reg
mov %eax, 0x406018 ; 3. write eax to mem

worker() { ...; counter = counter + 1; ... }

Dump of assembler code for worker():
// 0x406018 is address of counter variable
mov 0x406018, %eax ; 1. read from mem into eax
add $0x1, %eax ; 2. increment eax reg
mov %eax, 0x406018 ; 3. write eax to mem

Is there a thread interleaving that causes problems?
Problems With Races

- Race occurs when some execution sequences lead to unexpected results.
- May or may not occur depending on thread scheduler, execution speed, etc.
- Program may crash at times …
How can we avoid races?

Need to disallow concurrent access to avoid race

- T1 will read and update counter before T2 can access counter

A Critical Section is a region of code that needs to be protected from concurrent access

How can we enforce a critical section?
Mutual Exclusion

- Mutual exclusion ensures that only one thread executes in a critical section at a time

- E.g., shared counter updates occur one after the other
A mutex lock helps ensure mutual exclusion

- Threads use locks when accessing critical sections
- `lock(mutex):`
  - Acquire lock if mutex is free, mutex is now held
  - Otherwise wait until mutex is free
- `unlock(mutex):`
  - Release the lock, mutex is now free
  - If other threads are waiting, let them know mutex is free

Thread enters critical region

Thread leaves critical region
Mutex Locks

Thread 1

T1: Lock, mutex held

A enters critical region

T3: Unlock, mutex free

A leaves critical region

T3+: mutex held

Thread 2

T2: Lock

B attempts to enter critical region

B blocked

T4: Unlock, mutex free

B enters critical region

B leaves critical region

Time
Using a Mutex Lock

```c
// counter is located in shared address space
int counter;
// mutex is located in shared address space
struct lock lock mutex; // same mutex used by both threads

while() { 
    lock(&mutex);
    // critical section;
    counter++;
    unlock(&mutex);
    // remainder section;
}
```

Thread 1

```c
while() {
    lock(&mutex);
    // critical section;
    counter++;
    unlock(&mutex);
    // remainder section;
}
```

Thread 2
Mutual Exclusion Conditions

- A correct implementation of locking satisfies the following conditions:
  - Correctness
    - No two threads simultaneously in critical section
    - No assumption on the speed of thread execution
  - Performance
    - No thread running outside its critical section may block another thread in critical section
    - No thread must wait forever to enter its critical section
Summary

- Concurrent programming model
  - Threads enable concurrent execution
  - Threads cooperate by accessing shared variables

- Races
  - Concurrent accesses to shared variables can lead to races

- Critical sections and mutual exclusion
  - Avoiding races requires defining critical code sections, and
  - Running critical sections using mutual exclusion
  - Mutual exclusion is enforced using mutex lock abstraction

- Next lecture discusses how mutex locks are implemented
Think Time

- What is a race condition?
- How can we protect against race conditions?
- What is a critical section?
- What is mutual exclusion?
- Are there any drawbacks of long critical sections?
- Can you think of real world examples of critical sections? How are these critical sections managed?
- What is a data race?
What is a race condition?
- When certain thread interleavings are possible that lead to incorrect results.

How can we protect against race conditions?
- Code that accesses shared variables should be executed by one thread at a time.

What is a critical section?
- Code that accesses shared variables and needs to be protected from concurrent access.

What is mutual exclusion?
- The requirement that critical sections are executed by at most one thread at a time.
Think Time Answers

Are there any drawbacks of long critical sections?

- Only one thread operates in a critical section at a time, i.e., different threads execute serially in critical sections. Thus, long critical sections mean that there is less concurrency or parallelism available, which reduces performance. So, it is important in concurrent programs to lock just the critical sections (where shared variables are accessed) and not the non-critical sections of code.
Can you think of real world examples of critical sections? How are these critical sections managed?

- A toilet is a critical section! You don’t want any races there … You acquire a lock by closing the door, and release the lock by opening the door.

- A road intersection is a critical section. A race there leads to an accident. With stop signs, each car acquires a lock on their lane in the intersection and enters the intersection at a time, avoiding accidents.
What is a data race?

We have talked about race conditions in general in the lecture. A special kind of a race condition is a data race: two threads access a shared variable concurrently, at least one access is a write, and the threads do not use any synchronization (e.g., locks) to control access to the variable. The counter example in this lecture is a data race. Race conditions can be of other types as well.