Race Conditions ++

CS 350

race conditions arise ...

- multiple threads accessing (read/write) shared memory simultaneously

This is the programmers responsibility to identify and solve!

but ....

- race conditions can arise from things programmers **DIDN’T DO**

... nor are you aware of them!

register allocation optimization

- It’s faster to store variable values in registers than memory
  - Compiler optimizer will attempt to store values in registers for as long as possible
  - Compiler not necessarily aware of multi-threaded code!

```
int x = 5;

int A()
{
    int y = 0;
    for ( int i = 0; i < x; i ++ )
    {
        y ++;
    }
    x += y;
}
```

```
int B()
{
    x = 3;
    for ( int i = 0; i < x; i ++ )
    {
        x = i;
    }
    x += x;
}
```

int x = 5;

Thread 0: x is stored in $8

```
int A()
{
    int y = 0;
    for ( int i = 0; i < x; i ++ )
    {
        y ++;
    }
    x += y;
}
```

```
int B()
{
    x = 3;
    for ( int i = 0; i < x; i ++ )
    {
        x = i;
    }
    x += x;
}
```

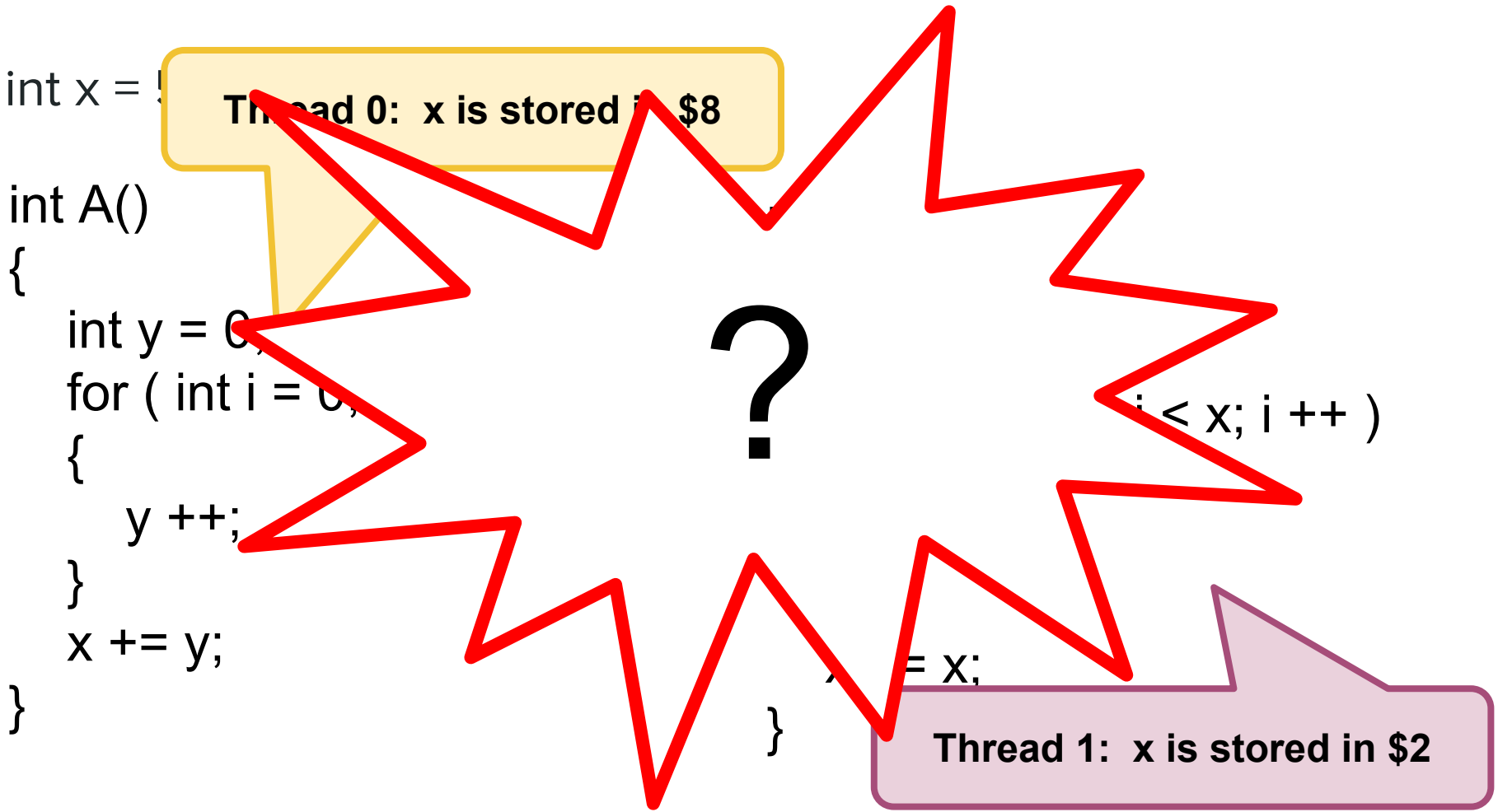
int x = 5;

Thread 0: x is stored in $8

```
int A()
{
    int y = 0;
    for ( int i = 0; i < x; i ++ )
    {
        y ++;
    }
    x += y;
}
```

```
int B()
{
    x = 3;
    for ( int i = 0; i < x; i ++ )
    {
        x = i;
    }
    x += x;
}
```

Thread 1: x is stored in $2


int x =
Thread 0: x is stored in $8
int A()
{
int y = 0;
for (int i = 0;
{
y ++;
}
x += y;
}
?
i < x; i ++)
= x;
}
Thread 1: x is stored in $2

volatile

- value in memory is not accurate (up-to-date)
  - Which thread has the accurate value?
  - Do any threads have the accurate value?
- Optimization introduces race condition; turn **OFF** optimization
- **volatile** keyword **[e.g. bool volatile x = 0;]**
  - Compiler does **NOT** optimize value into register
  - Value must always be loaded and stored with EVERY use

- **Does NOT provide synchronization**; ONLY turns off that optimization

load/store reordering

- Compiler AND hardware re-order loads/stores for performance
  - **How would that affect multi-threaded programs?**

load/store reordering

- Compiler AND hardware re-order loads/stores for performance
  - **How would that affect multi-threaded programs?**

Memory models

- Specify thread-memory interactions
  - load/store behaviours
  - Store visibility to threads
  - Memory assumptions
  - Optimization assumptions

Memory models

- Multi-threaded code requires ***sequential consistency*** for shared variables
  - I.e., load/store order is same for optimized and non-optimized code
- **Language-level memory models**
  - Describe thread-memory interaction at language/compiler level
  - Provide sequential consistency
    - Compiler must be able to identify multi-threaded code; typically through language-provided synchronization primitives
- **Modern languages provide a memory model**
  - C (11+), C++ (11+), Java, etc.

Memory models

- **Hardware-level memory model**
  - Architecture-specific; some provide multiple options!
  - specify how/when loads and stores can/are re-ordered
  - **Example:** SPARC Total-Store-Order
    - Stores reordered after loads
      - Read from Y can start after but finish before write to X
      - Read from Y cannot return value from write to Y until all processors have seen it
- Requires barrier/fence instructions to “disable” re-orders where needed
- **SYS/161 MIPS R3000 (OS/161) does not have barriers**

Conclusion

- Memory models specify load/store behaviours and optimizations of language and hardware
- volatile/barriers **do not stop race conditions in your code**
  - They enable/disable compiler/hardware optimizations (instruction reordering, register allocation) to prevent these optimizations from causing race conditions

What stops race conditions in your code?

Fun aside ...

Who used the MIPS R3000?

- Sony PS1 and PS2 (I/O CPU)
- Sieko-Epson
- SGI
- DEC
- NEC
- ... and many more