

# CSE 160

## Lecture 5

Consistency in practice  
Synchronization

# Announcements

- We are back to Room 1202 (EBU3B)
  - But not on 3/1, 3/13, 3/15
  - Plan:
    - We add an extra class on a Tuesday or Thursday evening in 1202 to replace 3/1 lecture
    - Double lecture on Weds 3/13 1pm to 3pm (our usual start time). 2 hours allowing time to get to SDSC.
  - Lab hours in the basement of EBU3B
    - Adding TA-lab sessions 12N-2pm on Tuesday
    - Need 2 more hours: Monday morning or Tuesday 2pm to 4pm?
- In class quiz on Friday 1/18

## Today's lecture

- Cache Consistency : pragmatics
- Synchronization

# Memory consistency

- Cache coherence tells us that memory will *eventually* be consistent
- The memory consistency policy tells us *when* this will happen
- A memory system is consistent if the following 3 conditions hold
  - Program order (you read what you wrote)
  - Definition of a coherent view of memory (“eventually”)
  - Serialization of writes (a single frame of reference)

# Memory consistency in practice

- There are 3 issues
  - Atomicity
  - Visibility
  - Ordering
- Under a non-causal consistency model, it is possible for processor P2 to observe  $A=1 \wedge B==1$  while processor P3 sees  $B==1 \wedge A == 0$  !

| P1  | P2        | P3        |
|-----|-----------|-----------|
| A=1 | if (A==1) | if (B==1) |
|     | B=1       | C=A       |

# Undefined behavior in C++11

## Global

```
int x, y;
```

### Thread 1

```
x = 17
```

```
y = 37;
```

### Thread 2

```
cout << y << " ";
```

```
cout << x << endl;
```

- Compiler may rearrange statements to improve performance
- Processor may rearrange order of instructions
- Memory system may rearrange order that writes are committed
- Memory might not get updated; “eventually can be a long time” (though in practice it’s often not)

# Undefined behavior in earlier versions of C++

## Global

```
int x, y;
```

### Thread 1

```
char c;
```

```
c=1;
```

```
int x=c;
```

### Thread 2

```
char b;
```

```
b =1;
```

```
int y=b;
```

- In C++11,  $x=1$  and  $y=1$ ;  
they are “separate memory locations”
- But in earlier dialects you might get 1&0, 0&1, 1&1
- The linker could allocate b and c next to each other  
in the same word of memory
- Modern processors can’t write a single byte, so they  
have to do read-modify-write

## Another example

### Global

// initially  $x == 0$  and  $y == 0$

Thread 1

if (x)  $y = 1$ ;

Thread 2

if (y)  $x = 1$ ;

- There is no data race in C++11
- See [www.stroustrup.com/C++11FAQ.html#memory-model](http://www.stroustrup.com/C++11FAQ.html#memory-model)

# Today's lecture

- Cache Consistency : pragmatics
- Synchronization
  - Mutual Exclusion
  - Barriers

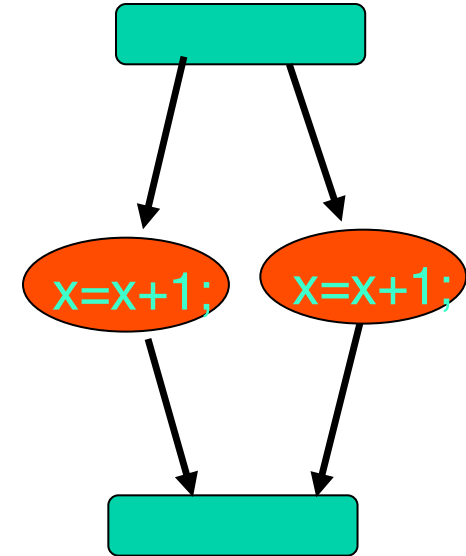
# Race conditions

- Consider the following thread function, where `x` is initially 0

```
void *threadFn(void *arg) {  
    x++;  
    pthread_exit(NULL); return 0;  
}
```
- What is the value of `x` after both threads have joined?
- A *race condition* arises because the timing of accesses to shared data can affect the outcome
- We say we have a *non-deterministic* computation

# Under the hood of a race condition

- Assume  $x$  is initially 0  
 $x = x + 1$ ;
- Generated assembly code
  - $r1 \leftarrow (x)$
  - $r1 \leftarrow r1 + \#1$
  - $r1 \rightarrow (x)$



- Possible interleaving with two threads

P1  
 $r1 \leftarrow x$

$r1 \leftarrow r1 + \#1$

$x \leftarrow r1$

P2  
 $r1 \leftarrow x$   
 $r1 \leftarrow r1 + \#1$   
 $x \leftarrow r1$

*$r1(P1)$  gets 0  
 $r1(P2)$  also gets 0  
 $r1(P1)$  set to 1  
 $r1(P2)$  set to 1  
P1 writes its R1  
P2 writes its R1*

# Avoiding race conditions

- Memory consistency and cache coherence are necessary but not sufficient conditions for ensuring program correctness
- We need to take steps to avoid race conditions through appropriate program synchronization
  - Critical sections
  - Barriers
  - Atomics

# Critical Sections

- Allow only 1 thread at a time to write to the shared memory location(s)
- The code performing the operation is called a *critical section*
- We use *mutual exclusion* to implement a critical section
- A critical section is non-parallelizing computation.. sensible guidelines?

Begin Critical Section

**X++;**

End Critical Section

# Mutual exclusion

- Pthreads provides mutex variables (locks)
- May be **CLEAR** or **SET**
- **Lock()** waits if the lock is set, else sets the lock
- **Unlock()** clears the lock if set

```
pthread_mutex_t mutex_sum;  
pthread_mutex_init(&mutex_sum, NULL);  
pthread_mutex_lock (&mutex_sum);  
        x++ ;                // Critical Section  
pthread_mutex_unlock (&mutex_sum);
```

# Computing a sum in parallel

## Globals

```
pthread_mutex_t    mutex_sum;  
int64_t            global_sum, *x[ ] = ...;
```

## Main

[\\$PUB/Examples/Threads/sumMtxG.cpp](#)

```
global_sum = 0;  
assert(!pthread_mutex_init(&mutex_sum, NULL));  
pthread_t* threads = new pthread_t[nThreads];  
ThreadArgs* args = new ThreadArgs[nThreads];  
for(int t=0; t<nThreads; t++){  
    args[t].tid = t;    args[t].n = N;    args[t].NT = nThreads;  
    assert(!pthread_create(&threads[t], NULL, sum, &args[t] ));  
}  
//Join threads...  
cout << "The sum of 1 to " << N << " is: " << global_sum << endl;
```

# The computation

```
for (int r =0; r<REPS; r++){  
    global_sum = 0;  
    for (i=i0; i<i1; i++){  
        pthread_mutex_lock(&mutex_sum);  
        global_sum += x[i];  
        pthread_mutex_unlock(&mutex_sum);  
    }  
}
```

```
% ./sumMtxG // 1M on 8 threads  
Result verified to be CORRECT  
Running time on 8 threads: 1.29 sec.
```

## Fixing the performance bug

- Locks are expensive, should be used sparingly

```
local_sum = 0;
for (int64_t i=i0; i<i1; i++)
    local_sum += x[i];
pthread_mutex_lock (&mutex_sum);
global_sum += local_sum;
pthread_mutex_unlock (&mutex_sum);
```

```
% sumMtx      // 1M on 8 threads
Running time on 8 threads: 0.00101018
```

```
% ./sumMtxG   // 1M on 8 threads
Running time on 8 threads: 1.29 sec.
```

## Implementation issues

- If we take out the critical section, we don't observe program failure!

```
% sum 2 16384 // 16K on 2 threads  
The sum of 1 to 16384 is: 134225920  
Run took 0.194073 milliseconds  
Result verified to be CORRECT.
```

- “Eventually” doesn't have to be a long time!
- Scheduler issues affect the outcome
  - Busy waiting or spinning
  - Pre-emption by scheduler forces thread to *yield*
- Hardware support
  - Test and set
  - Cache coherence protocol provides syncing

## More on Correctness

```
int64 sum = 0;    // Global
void *sumIt(void *arg){
    ThreadArgs* args = (ThreadArgs*) arg;
    int TID = args->tid;
    pthread_mutex_lock (&mutex_sum);
    sum += (TID+1);
    pthread_mutex_unlock (&mutex_sum);
    if (TID == 0)
        cout << "Sum of 1 : " << NT << " = " << sum << endl;
    pthread_exit(NULL);
    return NULL;
}
```

```
% ./sumIt 5
# threads: 5
The sum of 1 to 5 is 1
After join returns, the sum of 1 to 5 is: 15
```

# Barrier synchronization

- Why was the sum reported incorrectly ?
- Don't read a location updated by other threads that had not had the chance to produce its contribution (true dependence)
- Don't overwrite the values used by other processes in the current iteration until they have been consumed (anti-dependence)
- No thread can move past a barrier until all have arrived

```
pthread_mutex_lock (&mutex_sum);  
sum += 2*(TID+1);  
pthread_mutex_unlock (&mutex_sum);  
Barrier();  
if (TID == 0)  
    cout << "Total sum is " << sum << endl;
```

# Building a linear time barrier with locks

```
Mutex arrival=UNLOCKED, departure=LOCKED;  
int count=0;
```

```
void Barrier( )  
    arrival.lock( );           // atomically count the  
    count++;                  // waiting threads  
    if (count < $NT) arrival.unlock( );  
    else departure.unlock( );  // last processor  
                                // enables all to go  
  
    departure.lock( );  
    count--;                  // atomically decrement  
    if (count > 0) departure.unlock( );  
    else arrival.unlock( );    // last processor resets state
```