# Concurrency Control

Azza Abouzied

# Synchronization

# Case Study: The Ad server

Multi-threaded Ad Server counts number of times an ad was served: 0.001 fils/ad!!!

```
int hc; //global variable holding count of total hits

void update_hit_counter{
    hc++;
}
```

# From C to Assembly

```
hc++;
```

becomes

```
movl   hc, %eax    ;get hc
addl   $1, %eax    ;increment hc
movl   %eax, hc    ;store hc
```

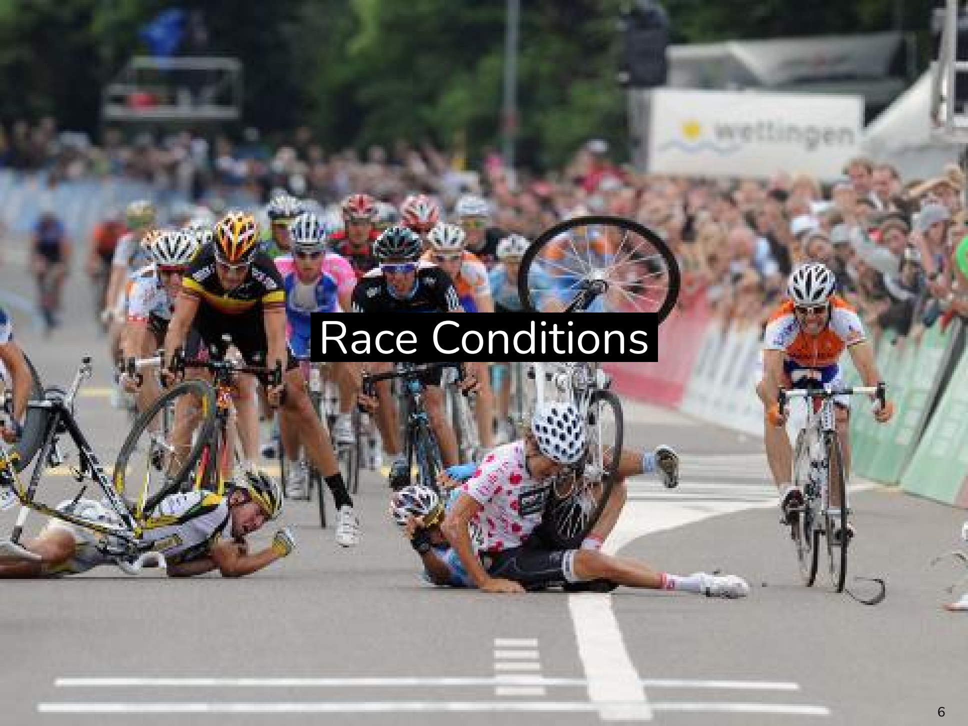
# The Life of Two Threads

| Thread 1 | Thread 2 |
| --- | --- |
| movl hc, %eax | |
| get hc → 5 | |
| addl $1, %eax | |
| increment hc → 6 | |
| movl %eax, hc | |
| store hc → 6 | |
| | movl hc, %eax |
| | get hc → 6 |
| | addl $1, %eax |
| | increment hc → 7 |
| | movl %eax, hc |
| | store hc → 7 |

# Damnatio Memoriae of a thread

| Thread 1 | Thread 2 |
|---|---|
| movl hc, %eax | |
| get hc $\rightarrow$ 5 | |
| | movl hc, %eax |
| | get hc $\rightarrow$ 5 |
| addl $1, %eax | |
| increment hc $\rightarrow$ 6 | |
| | addl $1, %eax |
| | increment hc $\rightarrow$ 6 |
| movl %eax, hc | |
| store hc $\rightarrow$ 6 | |
| | movl %eax, hc |
| | store hc $\rightarrow$ 6 |

# Race Conditions



# Race Conditions

Order of threads affects outcome of the computation then we have race conditions. These create non-determinism!

Code paths that access/manipulate shared data are critical sections.

If a critical section executes atomically then we prevent concurrent access to shared data at critical sections.

| Thread 1 | Thread 2 |
|---|---|
| get hc; increment hc; store hc; | |
| | get hc; increment hc; store hc; |

| Thread 1 | Thread 2 |
|---|---|
| | get hc; increment hc; store hc; |
| get hc; increment hc; store hc; | |

# What about mutual exclusion? Locking

| Thread 1 | Thread 2 |
|---|---|
| try to acquire lock | try to acquire lock |
| Success: lock acquired | Failed: wait ... |
| get hc | wait |
| increment hc | wait |
| store hc | wait |
| unlock lock | wait |
| ... | Success: lock acquired |
| | ... |

But you just pushed the problem to this lock thing ... how do you make a lock?

What if another process ignores the locks: locks are advisory and voluntary

# Sources of Concurrency

- **Interrupts**
- **User-space preemption**: The scheduler decides when to preempt you and when to execute you
- **Kernel preemption**: The kernel itself is a multi-threaded beast sharing address space and is preemptive
- **Sleep, Block**
- **SMP**: two processors can be executing the same code at exactly the same time (kernel or user)

# Protect your code

- **Interrupts** — *interrupt-safe*
- **User-space preemption**: The scheduler decides when to preempt you and when to execute you — *preempt-safe*
- **Kernel preemption**: The kernel itself is a multi-threaded beast sharing address space and is preemptive — *preempt-safe*
- **Sleep, Block** — *preempt-safe*
- **SMP**: two processors can be executing the same code at exactly the same time (kernel or user) — *SMP-safe*

# Synchronization Primitives

# Our first lock

```c
int hc_busy;
int hc;

void update_hit_counter(){
    while(hc_busy);
    hc_busy = 1;
    hc++; //Critical Section
    hc_busy = 0;
}
```

Why does this not work?

# Our first lock

```
int hc_busy;
int hc;

void update_hit_counter(){
    while(hc_busy);
    hc_busy = 1;
    hc++; //Critical Section
    hc_busy = 0;
}
```

Why does this not work?

The crux of any synchronization primitive is the ability to run atomic operations. All architectures provide a test_and_set() instruction. On x86 this is the

```
lock cmpxchg   %eax, lock_in_mem
```

# The Spin Lock

```
int hc_busy;
int hc;

void update_hit_counter(){
    while(1){      //Spin Lock
        if(test_and_set(hc_busy, 1)){
            hc++;   //Critical Section
            test_and_set(hc_busy, 0);
            return;
        }
    }
}
```

What if you have no hardware support for atomic operations?

If interrupts are sources of concurrency, can't we just disable interrupts?

cli disabled interrupts on a uniprocessor, sti enabled interrupts.

# Locking without hardware support

```
int hc;
int thread_waiting[2];

void update_hit_counter(int me){
    thread_waiting[me] = 1;
    while(1){
        if(thread_waiting[!me] == 0) break;
    }
    hc++;
    thread_waiting[me] = 0;
}
```

# Peterson's algorithm

```
int hc;
int turn;
int thread_waiting[2];

void update_hit_counter(int me){
  thread_waiting[me] = 1;
  turn = me; //my turn to wait
  while(1){
    if(thread_waiting[!me] == 0) break;
    if(turn != me) break;
  }
  hc++;
  thread_waiting[me] = 0;
}
```

Questions?