Parallelism via Threads

Basics of

- C with Posix Threads
- Java Threads
Counting 3s

```c
int *array;
int length = 100000;
int count;
int iters = 10000; /* artificially multiply work */

int count3s()
{
    int i, j;
    int count = 0;

    for (j = 0; j < iters; j++) {
        for (i = 0; i < length; i++) {
            if (array[i] == 3) {
                count++;
            }
        }
    }

    return count;
}
```
Starting Threads

Using simplified Posix interface:

```c
int t = 2;

int count3s()
{
    int i;

    count = 0;

    for (i = 0; i < t; i++)
        thread_create(count3s_thread, i);

    join_threads();

    return count;
}
```
Each Thread

void count3s_thread(int id)
{
    int length_per_thread = length / t;
    int start = id * length_per_thread;
    int i, j;

    for (j = 0; j < iters; j++) {
        for (i = start; i < start + length_per_thread; i++) {
            if (array[i] == 3) {
                count++;
            }
        }
    }
}

Returns the wrong answer!
Data Race

count++; means v = count + 1;
count = v;

Possible interleaving:

thread 1

v = count + 1;

count = v;

thread 2

v = count + 1;
count = v;

Need a lock...
Use a lock to allow only one thread at a time:

```
lock(id);
count++;
unlock(id);
```

The code between `lock` and `unlock` is called a **critical section**
Peterson’s Algorithm (Slight Detour)

```c
int flag[2];
int turn;

static void lock(int id)
{
    flag[id] = 1;
    turn = !id;
    while (flag[!id] && turn == !id) { }
}

static void unlock(int id)
{
    flag[id] = 0;
}
```

Doesn't work... try adding `volatile`...

Still doesn't work... need `asm("mfence")`
Sharing Protected by Mutex

Obviously, it’s better to use locks supplied by the thread system:

```
mutex m = INIT_MUTEX;

...  

mutex_lock(m);
count++;
mutex_unlock(m);
```

Works, but very slowly
Reduce Lock Contention

```c
int private_count[MaxThreads];
...

void count3s_thread(int id)
{
    ...
    if (array[i] == 3) {
        private_count[id]++;
    }
    ...

    mutex_lock(m);
    count += private_count[id];
    mutex_unlock(m);
}
```

Still much slower!? This is a cache effect...
Reduce Cache Contention

```c
struct padded_int
{
    int value;
    char padding[60];
} private_count[MaxThreads];

....
private_count[id].value++;  
```

**Finally**, about twice as fast as the original!
int sub_counts[MaxThreads];

void count3s_thread(int id)
{
    ... 
    int private_count = 0;
    ... 
    sub_counts[id] = private_count;
}

int count3s()
{
    ... 

    join_threads();

    count = 0;

    for (i = 0; i < t; i++)
    {
        count += sub_counts[i];
    }

    return count;
}
Java Threading

[See provided Java variant]
Conclusion

Lessons for today:

- Threads, races, locks, contention
- With concurrency, consider carefully shared state
- `volatile` doesn’t fix concurrency bugs
- Avoid modifying shared variables