11. Concurrency

We like loading into RAM — it is fast BUT this is the recipe for races

Say I want to keep track of my money:

```
bool deposit(int amt) {
  if (amt < 0 || INT_MAX - amt > balance) return false;
  balance += amt;
  return true;
}
bool withdraw(int amt) {
  if (amt < 0 || balance - amt > 0) return false;
  balance += amt;
  return true;
}
```

This is a TERRIBLE implementation

- if two threads try to deposit at the same time, one may be ignored
- an interrupt during the += can cause an ignore

This is a mistake in synchronization, which can occur in:

  - a single CPU with preemptive scheduling
  - a single thread with interrupt handling
  - a multiple CPU system

We call these errors caused by concurrent access race conditions
Pieces of code which can cause race conditions are called critical sections

If we can make critical sections atomic at the point of observability, we can say that the actions are serializable, or the same result could be found by running the operations sequentially

We have atomicity iff we have:

i. Mutual Exclusion ~ one thread in the critical section excludes all others

ii. <u>Bounded Wait</u> ~ eventually, waiting threads will be able to enter the critical section

There are a few major complications that we must guarantee atomicity around:

i. Preemptive Scheduling
  - we may be interrupted mid critical section and leave in an invalid state

ii. Threads
  - a thread might be kicked out while others are running and leave the others to utilize an invalid state

Goldilocks principle for critical sections:

- If everyone only reads, then everyone is safe;
- Only writes cause problems

  → when searching for critical sections

1. look for shared writes
   a. each of the shared writes should be in a critical sectIon
2. expand to include dependent reads & intervening computation
   a. do this recursively once we have changed the critical section

```
// new_balance dependent on balance
long long new_balance = balance + amt;
balance = new_balance;
```

Guaranteeing these rules is hard...we can avoid it with a few tricks:

1. Single-Threaded Code
2. Event-Driven Programming
   - useful for when we have 1 CPU and many threads
   - of the form:

```
while(true) {
  wait for an event
  act on that event
}
```

   - acting on E must be fast enough as to avoid any waiting
   - common in IOT appliances
   - Downsides:
     - code is restricted
     - no true parallelism

        - too easy
3. Synchronization via Load & Store
    - the size of objects is restricted
        - no large objects! (copying takes multiple cycles & is not atomic)
        - no small objects! (writing less than a byte still copies the whole byte first)
        - → we can use only objects of 1 byte, since that is the granularity of x86-64 copy

We can also have synchronization errors without any critical sections whatsoever:

```
// thread 1
for (long i = 0; i < n; i++) continue
// thread 2
...
n = 0;
// i is only copied once! s trying to force thread 1 to stop waiting fails
// we must use the keyword 'volatile' to require copying from memory each time
```

To guarantee atomicity, we must introduce the concept of locks!