Announcements

- HW13 due tonight
- Scheme contest due Friday
- Special guest lecture by Brian Harvey on Friday at 2pm
  - Attendance is mandatory!!!

The Problem with Shared State

```python
def increment():
    count = counter[0]
    sleep(0)  # May cause the interpreter to switch threads
    counter[0] = count + 1
```

Given a switch at the `sleep` call, here is a possible sequence of operations on each thread:

- Thread 0: read `counter[0]`:
  - Thread 1: read `counter[0]`:
  - calculate `0 + 1: 1`
  - write `1 -> counter[0]`

The counter ends up with a value of 1, even though it was incremented twice!

Synchronized Data Structures

```
from queue import Queue
queue = Queue()
def increment():
    count = queue.get()
    sleep(0)
    queue.put(count + 1)
other = Thread(target=increment, args=())
other.start()
queue.put(0)
increment()
other.join()
print('count is now', queue.get())
```

Manual Synchronization with a Lock

```
from threading import Lock
counter = [0]
counter_lock = Lock()
def increment():
    counter_lock.acquire()
    count = counter[0]
    sleep(0)
    counter[0] = count + 1
    counter_lock.release()
other = Thread(target=increment, args=())
other.start()
increment()
other.join()
print('count is now', counter[0])
```

The With Statement

```
def increment():
    counter_lock.acquire()
    count = counter[0]
    sleep(0)
    counter[0] = count + 1
    counter_lock.release()
def increment():
    with counter_lock:
        count = counter[0]
        sleep(0)
        counter[0] = count + 1
```
Example: Web Crawler

A web crawler is a program that systematically browses the Internet.

For example, we might write a web crawler that validates links on a website, recursively checking all links hosted by the same site.

A parallel crawler may use the following data structures:
- A queue of URLs that need processing
- A set of URLs that have already been seen, to avoid repeating work and getting stuck in a circular sequence of links.

These data structures need to be accessed by all threads, so they must be properly synchronized.

They synchronized `Queue` class can be used for the URL queue.

There is no synchronized set in the Python library, so we must provide our own synchronization using a lock.

Example: Particle Simulation

A set of particles all interact with each other (e.g. short range repulsive force).

The set of particles is divided among all threads/processes.

Forces are computed from particles’ positions:
- Their positions constitute shared data.

The simulation is discretized into timesteps.

Synchronization in the Web Crawler

The following illustrates the main synchronization in the web crawler:

```python
def get_url():
    """Retrieve a URL."""
    return queue.get()

def put_url(url):
    """Queue the given URL."""
    queue.put(url)

def already_seen(url):
    """Check if a URL has already been seen."""
    with seen_lock:
        if url in seen:
            return True
        seen.add(url)
    return False
```

Solution #1: Barriers

In each timestep, each thread/process must:

1. Read the positions of every particle (read shared data).
2. Update acceleration of its own particles (access non-shared data).
3. Update velocities of its own particles (access non-shared data).
4. Update positions of its own particles (write shared data).

Steps 1 and 4 conflict with each other.

We can solve this conflict by dividing the program into phases, ensuring that all threads change phases at the same time.

A barrier is a synchronization mechanism that accomplishes this.

```python
from threading import Barrier

barrier = Barrier(num_threads)
barrier.wait()  # Waits until num_threads threads reach it
```

Solution #2: Message Passing

Alternatively, we can explicitly pass state from the thread/process that owns it to those that need to use it.

In each timestep, every process makes a copy of its own particles.

Then, they do the following `num_processes - 1` times:

1. Interact with the copy that is present.
2. Send the copy to the left, receive from the right.

Thus, reads are on copies, so they don’t conflict with writes.
**Summary**

Parallelism is necessary for performance, due to hardware trends

But parallelism is hard in the presence of mutable shared state

- Access to shared data must be synchronized in the presence of mutation

Making parallel programming easier is one of the central challenges that Computer Science faces today

---

**Abstraction, Abstraction, Abstraction**

The central idea of 61A is abstraction

- Not only central in Computer Science, but in any discipline that deals with complex systems

Abstraction is our main tool for managing complexity

- Complex systems have multiple abstraction layers to divide the system as a whole into manageable pieces

Not only did we learn how to use abstractions, we learned how to build them

- Nothing is magical
- We saw lots of cool ideas (e.g. objects, rlists, interpreters, logic programming), but we also saw how they work
- Simple and compact implementations provide very powerful abstractions

---

**61A Topics in Future Courses**

You will see the topics you learned here many times over your academic career and beyond

Here is a (partial) mapping between CS classes and 61A topics:

- **61B**: Object-oriented programming, inheritance, multiple representations, recursive data (rlists and trees), orders of growth
- **61C**: MapReduce, Parallelism
- **70**: Recursion/induction, halting problem
- **162**: Parallelism
- **164**: Recursive data, interpretation, declarative programming
- **170**: Recursive data, orders of growth, logic
- **172**: Halting problem
- **186**: Declarative programming

Of course, you will see abstraction everywhere!

---

**Stay Involved!**

The community is what makes 61A great (TAs, readers, lab assistants)

The entire teaching staff consists of undergrads like you

- Most of them are sophomores!

If you can, please lab assist for future semesters

- You get units!
- Readers and TAs are often chosen based on their involvement with the course, in addition to grades and other factors

You can apply to be a reader or TA here:

https://willow.coe.berkeley.edu/PHP/gsiapp/menu.php

---

**The 61A Staff**

From all of us:
Thank you for a wonderful semester!

---

**61A Rocks!**

Thanks to Andy Qin!

Thanks to Adithya Murali!