Administrivia

- Short week this week
  - Wed lecture cancelled (but OH available in AND 223 at 11:30)
  - 🎤 Fri holiday 🎤

- HW4 due in 1 ½ weeks (12/05)

- Ex 17 (час last час exercise!!) out, due Wednesday
Some Common hw4 Bugs 😞

- Your server works, but is really, really slow
  - Check the 2nd argument to the `QueryProcessor` constructor

- Funny things happen after the first request
  - Make sure you’re not destroying the `HTTPConnection` object too early (e.g. falling out of scope in a while loop)

- Bikeapalooza not loading properly
  - Check that you are handling all necessary file types. (can use the developer console in a web browser to check this)

- Server crashes on a blank request
  - Make sure that you handle the case that `read()` (or `WrappedRead()` ) returns 0
Lecture Outline

❖ Threads: Cleanup and Data Races
❖ `pthreads` and Locks
❖ Other Concurrency Techniques
**pthread API Review**

- **`int pthread_create()`**
  - Creates a new thread, stores a thread id in `*thread`
  - Returns 0 on success and an error number on error (can check against error constants)
  - The new thread runs `start_routine(arg)`

- **`void pthread_exit(void* retval);`**
  - Equivalent of `exit(retval);` for a thread instead of a process
  - The thread will automatically exit once it returns from `start_routine()`
pthread API review

- **pthread_join**
  - Waits for the thread specified by `thread` to terminate
  - The thread equivalent of `waitpid()`
  - The exit status of the terminated thread is placed in `**retval`.

  ```c
  int pthread_join(pthread_t thread, void **retval);
  ```

- **pthread_detach**
  - Mark thread specified by `thread` as detached – it will clean up its resources as soon as it terminates

  ```c
  int pthread_detach(pthread_t thread);
  ```
pthread Demos

- See `pthread.c`
  - Notice how we manage memory
    - When do we allocate deallocate memory?
    - How do we pass possession of memory to threads?

- See `exit_thread.c`
  - Do we need to join every thread we create?
Data Races

- A **data race** occurs when two or more different threads access the same location, at least one thread changes that memory, and they occur one after another.
  - Means that the result of a program can vary depending on chance (which thread ran first?)
Data Race Example

❖ If your fridge has no milk, then go out and buy some more
  ▪ What could go wrong?

❖ If you live alone:

❖ If you live with a roommate:

```c
if (!milk) {
    buy milk
}
```
Data Race Example

❖ Idea: leave a note!
  ▪ Does this fix the problem?

A. Yes, problem fixed
B. No, could end up with no milk
C. No, could still buy multiple milks
D. We’re lost…

```cpp
if (!note) {
    if (!milk) {
        leave note
        buy milk
        remove note
    }
}
```
Threads and Data Races

- Data races might interfere in painful, non-obvious ways, depending on the specifics of the data structure

- **Example**: two threads try to read from and write to the same shared memory location
  - Could get “correct” answer
  - Could accidentally read old value
  - One thread’s work could get “lost”

- **Example**: two threads try to push an item onto the head of the linked list at the same time
  - Could get “correct” answer
  - Could get different ordering of items
  - Could break the data structure!
Lecture Outline

- Difficulties with Threads: Cleanup and Data Races
- `pthreads` and Locks
- Other Concurrency Techniques
Synchronization

- **Synchronization** is the act of preventing two (or more) concurrently running threads from interfering with each other when operating on shared data
  - Need some mechanism to coordinate the threads
    - “Let me go first, then you can go”
  - Many different coordination mechanisms have been invented (see CSE 451)

- Goals of synchronization:
  - **Safety** – avoid unintended interactions with shared data structures (informally: “nothing bad happens”)
  - **Liveness** – ability to execute in a timely manner (informally: “something good happens”)
Lock Synchronization

- Use a “Lock” to grant access to a *critical section* so that only one thread can operate there at a time
  - Executed in an uninterruptible (*i.e.* atomic) manner

- **Lock Acquire**
  - Wait until the lock is free, then take it

- **Lock Release**
  - Release the lock
  - If other threads are waiting, wake exactly one up to pass lock to

**Pseudocode:**

```cpp
// non-critical code
lock.acquire();
// critical section
lock.release();
// non-critical code
```
Milk Example – What is the Critical Section?

❖ What if we use a lock on the refrigerator?
  ▪ Probably overkill – what if roommate wanted to get eggs?

❖ For performance reasons, only put what is necessary in the critical section
  ▪ Only lock the milk
  ▪ But lock all steps that must run uninterrupted (i.e. must run as an atomic unit)

```c
fridge.lock();
if (!milk) {
    buy milk
}
fridge.unlock();
```

```c
milk_lock.lock();
if (!milk) {
    buy milk
}
milk_lock.unlock();
```
pthreads and Locks

❖ Another term for a lock is a **mutex** (“mutual exclusion”)
   ▪ **pthread.h** defines datatype **pthread_mutex_t**

❖ **pthread_mutex_init**
   ▪ Initializes a mutex with specified attributes

   ```c
   int pthread_mutex_init(pthread_mutex_t* mutex, const pthread_mutexattr_t* attr);
   ```

❖ **pthread_mutex_lock**
   ▪ Acquire the lock – blocks if already locked

   ```c
   int pthread_mutex_lock(pthread_mutex_t* mutex);
   ```

❖ **pthread_mutex_unlock**
   ▪ Releases the lock

   ```c
   int pthread_mutex_unlock(pthread_mutex_t* mutex);
   ```

❖ **pthread_mutex_destroy**
   ▪ “Uninitializes” a mutex – clean up when done

   ```c
   int pthread_mutex_destroy(pthread_mutex_t* mutex);
   ```
pthread Mutex Examples

- See `total.cc`
  - Data race between threads

- See `total_locking.cc`
  - Adding a mutex fixes our data race

- How does this compare to sequential code?
  - Likely *slower* – only 1 thread can increment at a time, but have to deal with checking the lock and switching between threads
  - One possible fix: each thread increments a local variable and then adds its value (once!) to the shared variable at the end
  - See `total_locking_better.cc`
C++11 Threads

- C++11 added threads and concurrency to its libraries
  - `<thread>` – thread objects
  - `<mutex>` – locks to handle critical sections
  - `<condition_variable>` – used to block objects until notified to resume
  - `<atomic>` – indivisible, atomic operations
  - `<future>` – asynchronous access to data
  - These might be built on top of `<pthread.h>`, but also might not be
Lecture Outline

❖ Difficulties with Threads: Cleanup and Data Races
❖ pthreads and Locks
❖ Other Concurrency Techniques
Review: Why Sequential?

❖ Advantages:
  ▪ Simple to write, maintain, debug
  ▪ The default, supported everywhere

❖ Disadvantages:
  ▪ Depending on application, poor performance
    • One slow client will cause *all* others to block
    • Poor utilization of resources (CPU, network, disk)
Review: Why Concurrent Threads?

❖ Advantages:
  - Almost as simple to code as sequential
  - Concurrent execution with good CPU and network utilization
  - Threads can run in parallel if you have multiple CPUs/cores
  - Shared-memory communication is possible

❖ Disadvantages:
  - Need language and OS support for threads
  - If threads share data, you need locks or other synchronization
  - Threads can introduce overhead (technical + cognitive)
  - Threads have a “shared fate” (eg, “rogue” thread, shared limits)
Alternative: Different I/O Handling (1 of 2)

- Use asynchronous or non-blocking I/O

- Your program begins processing a task
  - When your program needs to read data to make further progress, it registers interest in the data with the OS and then switches to a different task
  - The OS handles the details of issuing the read on the disk/console/network
  - When data becomes available, the OS lets your program know

- Your program (almost never) blocks on I/O
Alternative: Different I/O Handling (2 of 2)

- But some devices can truly block your program
  - Remote computer may wait arbitrarily long before sending data
  - User may walk away from console

- How to use non-blocking I/O:
  - Enable non-blocking I/O on its file descriptors
  - Issue `read()` and `write()` system calls
    - If the read/write would block, the system call returns immediately
  - Ask the OS which file descriptors are readable/writeable
    - Can choose to block while no file descriptors are ready
Alternative: Processes

- What if we forked processes instead of threads?

- Advantages:
  - No shared memory between processes
  - No need for language support; OS provides `fork()`

- Disadvantages:
  - More overhead than threads during creation and context switching
  - Cannot easily share memory between processes – typically communicate through the file system