Concurrency: Threads

CSE 333 Summer 2020

Instructor: Travis McGaha

Teaching Assistants:

Jeter Arellano	Ramya Challa	Kyrie Dowling
Ian Hsiao	Allen Jung	Sylvia Wang

Poll Everywhere

pollev.com/cse33320su

About how long did Exercise 16 take?

A. 0-1 Hours
B. 1-2 Hours
C. 2-3 Hours
D. 3-4 Hours
E. 4+ Hours
F. I didn't submit / I prefer not to say

Side question:
Favourite breakfast food?

Administrivia

- Exercise 17 released today, due Monday (8/17)
  - Concurrency via pthreads
- hw4 due Thursday (8/20)
  - Submissions accepted until Sunday (8/23) @ 11:59 pm
  - If you want to use late day(s), you **MUST** let staff know. Make a private post on ed or send an email to staff letting us know you want to use late day(s).
- Remaining late days posted on canvas. Grades for various assignments will also be posted soon.
  - Please Contact staff if something seems incorrect!!!

Creating and Terminating Threads

Output parameter.
Gives us a "thread_descriptor"

- ```
  int pthread_create(
          pthread_t* thread,
          const pthread_attr_t* attr,
          void* (*start_routine)(void*),
          void* arg);
  ```
  Argument for the thread function

  Function pointer!
  Takes & returns void*
  to allow "generics" in C

  - Creates a new thread into `*thread`, with attributes `*attr` (`NULL` means default attributes)
  - Returns **0** on success and an error number on error (can check against error constants)
  - The new thread runs **`start_routine`**`(arg)`

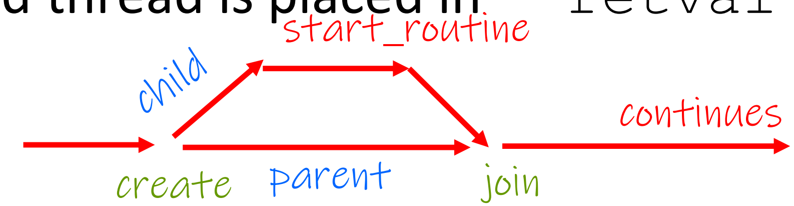
    child start_routine
    continue
    pthread_create parent

- ```
  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`**`()`

What To Do After Forking Threads?

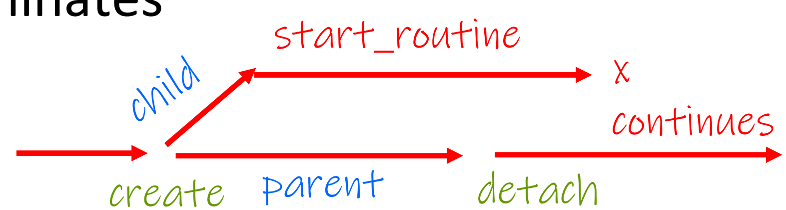
- `int pthread_join(pthread_t thread, void** retval);`
  - Waits for the thread specified by `thread` to terminate
  - The thread equivalent of **`waitpid`**`()`
  - The exit status of the terminated thread is placed in `**retval`

Parent thread waits for child thread to exit, gets the child's return value, and child thread is cleaned up



- `int pthread_detach(pthread_t thread);`
  - Mark thread specified by `thread` as detached – it will clean up its resources as soon as it terminates

Detach a thread.
Thread is cleaned up when it is finished

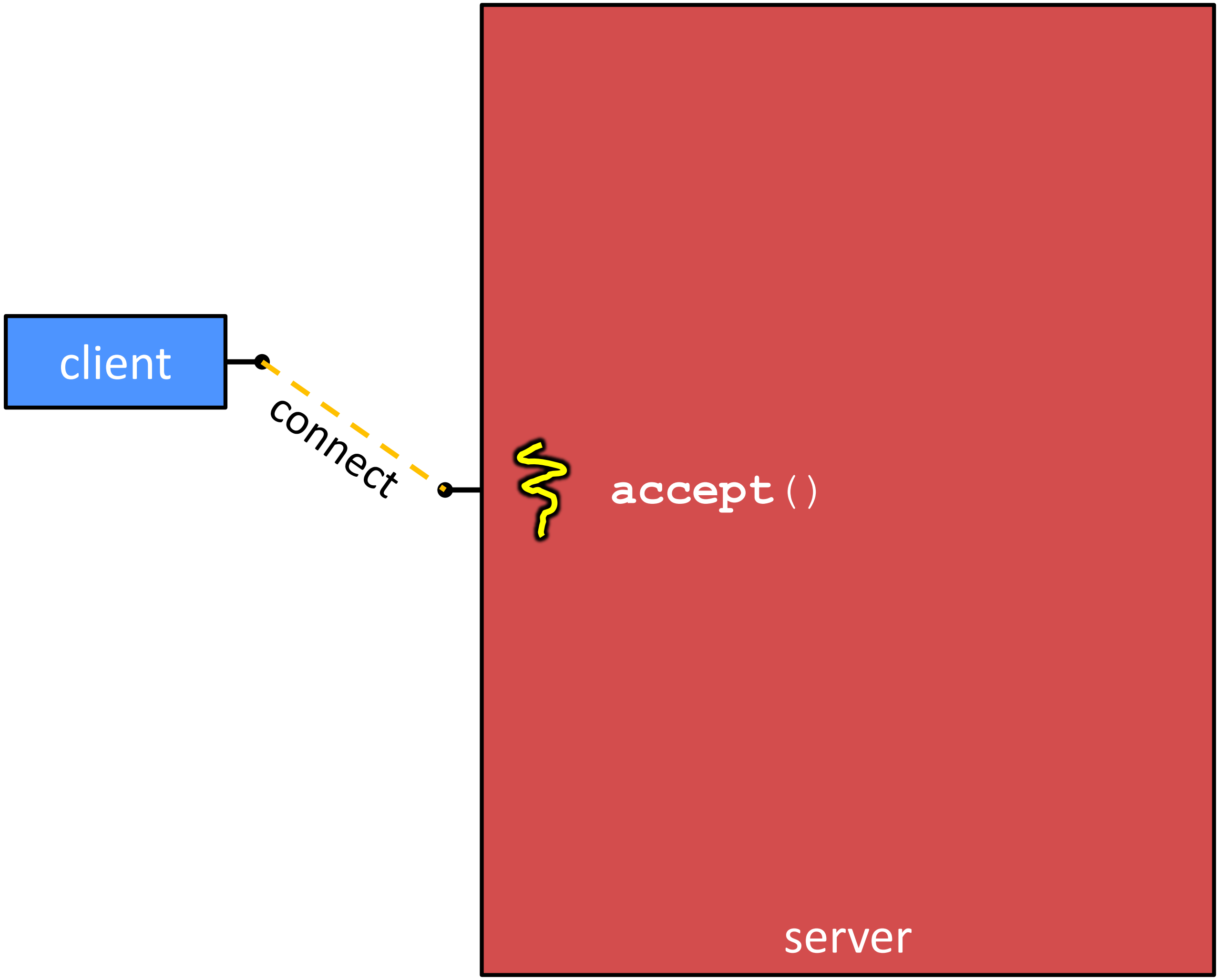


Concurrent Server with Threads

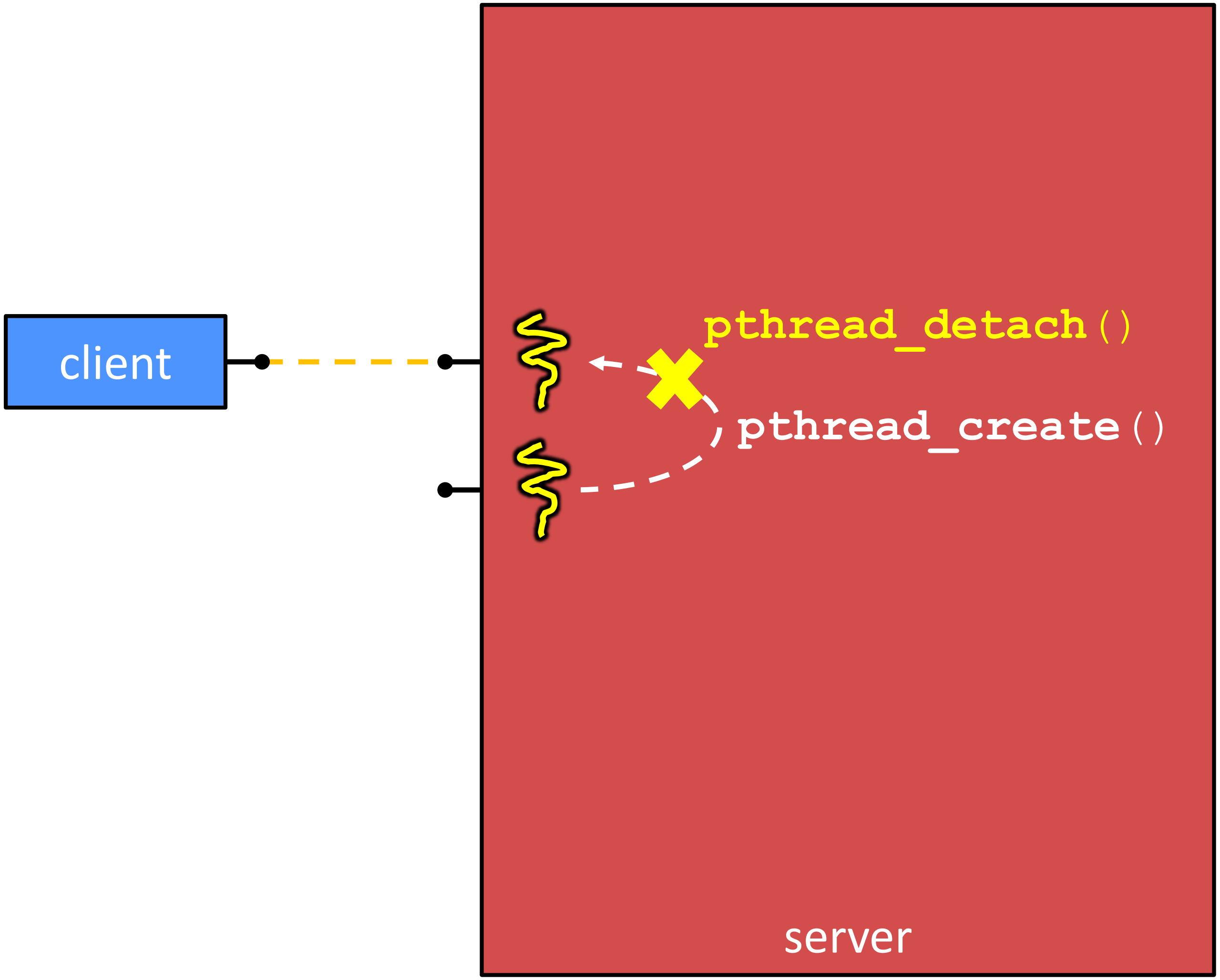
- A single *process* handles all of the connections, but a parent *thread* dispatches (creates) a new thread to handle each connection
  - The child thread handles the new connection and then exits when the connection terminates

- See `searchserver_threads/` for code if curious

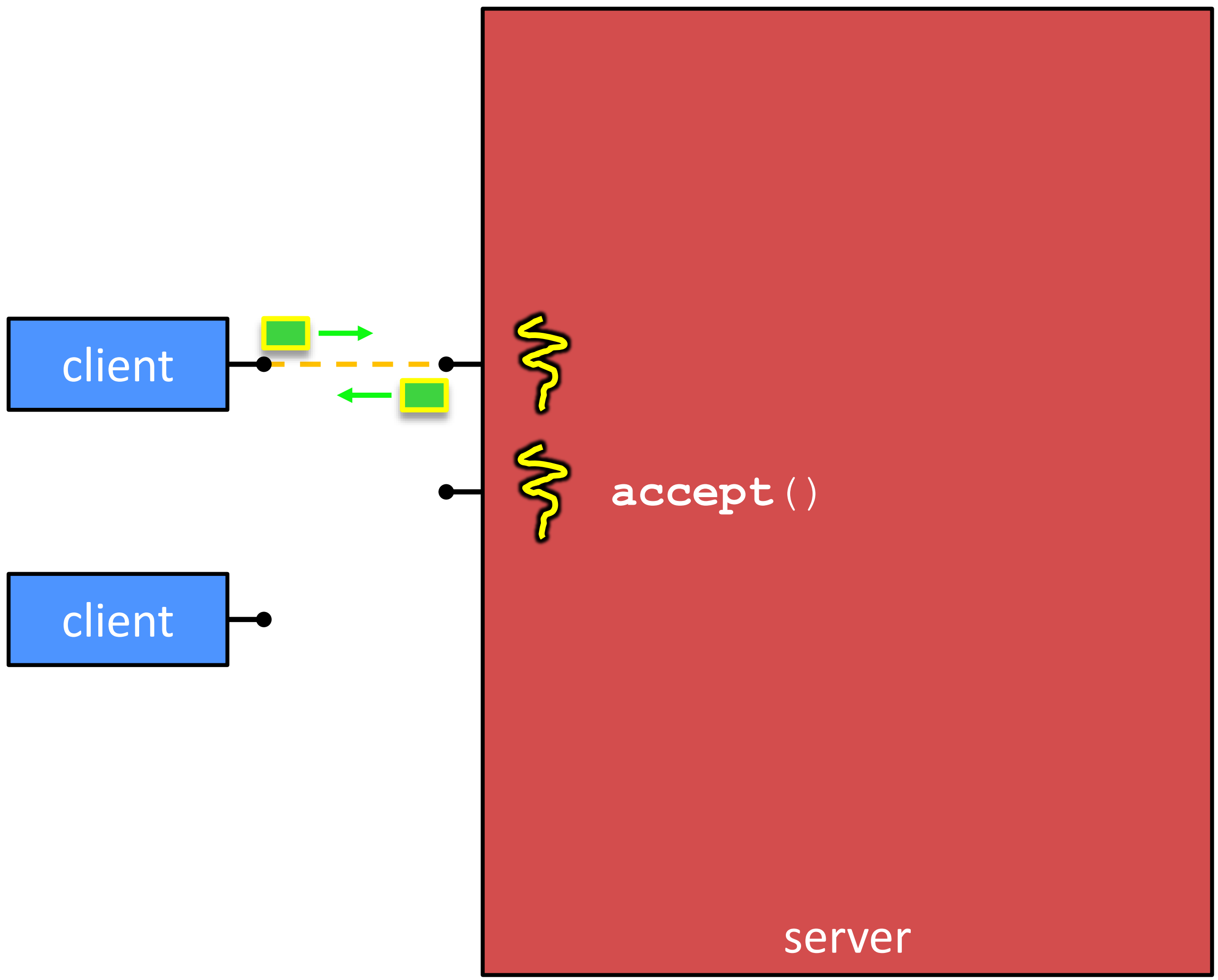
Multithreaded Server



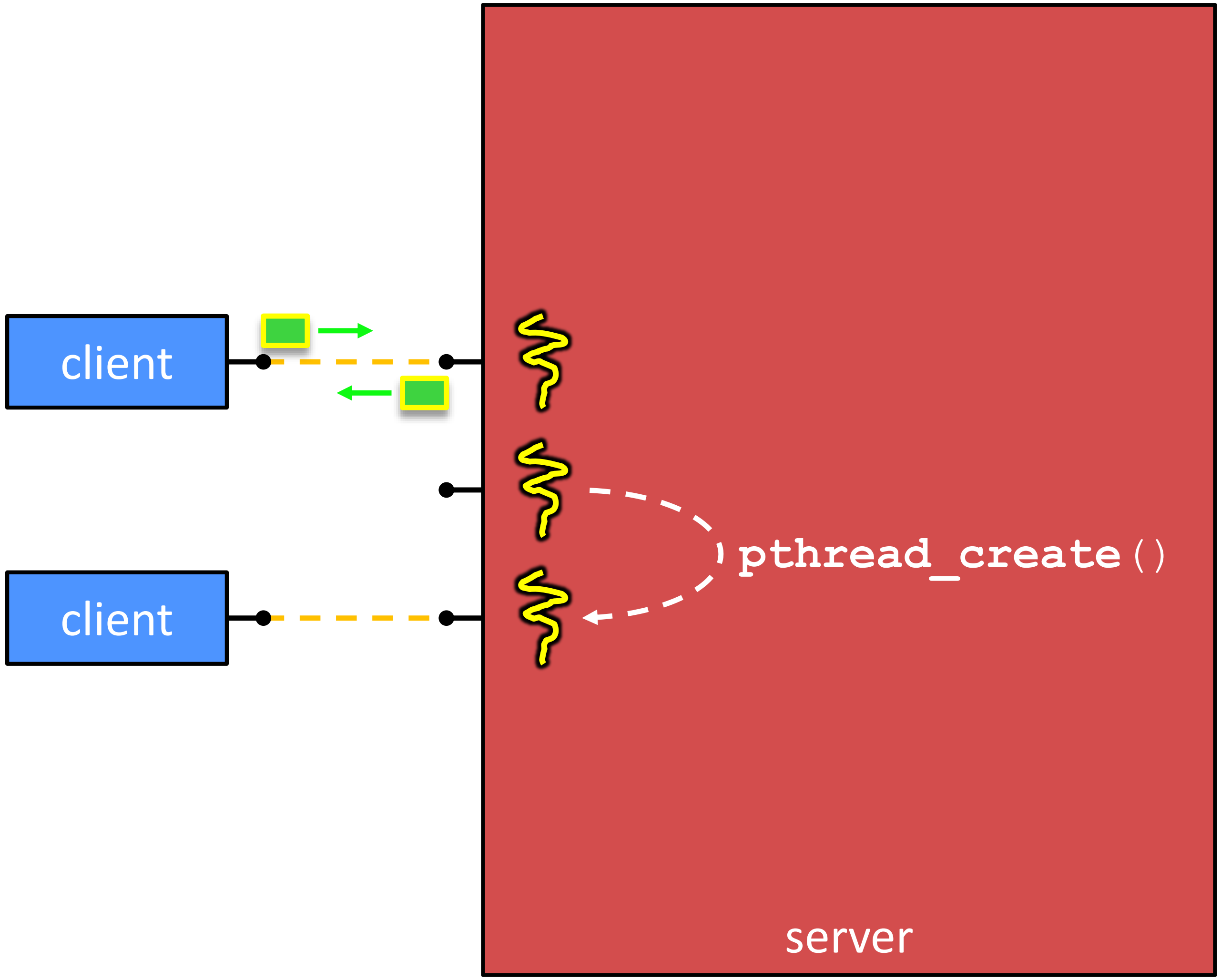
Multithreaded Server



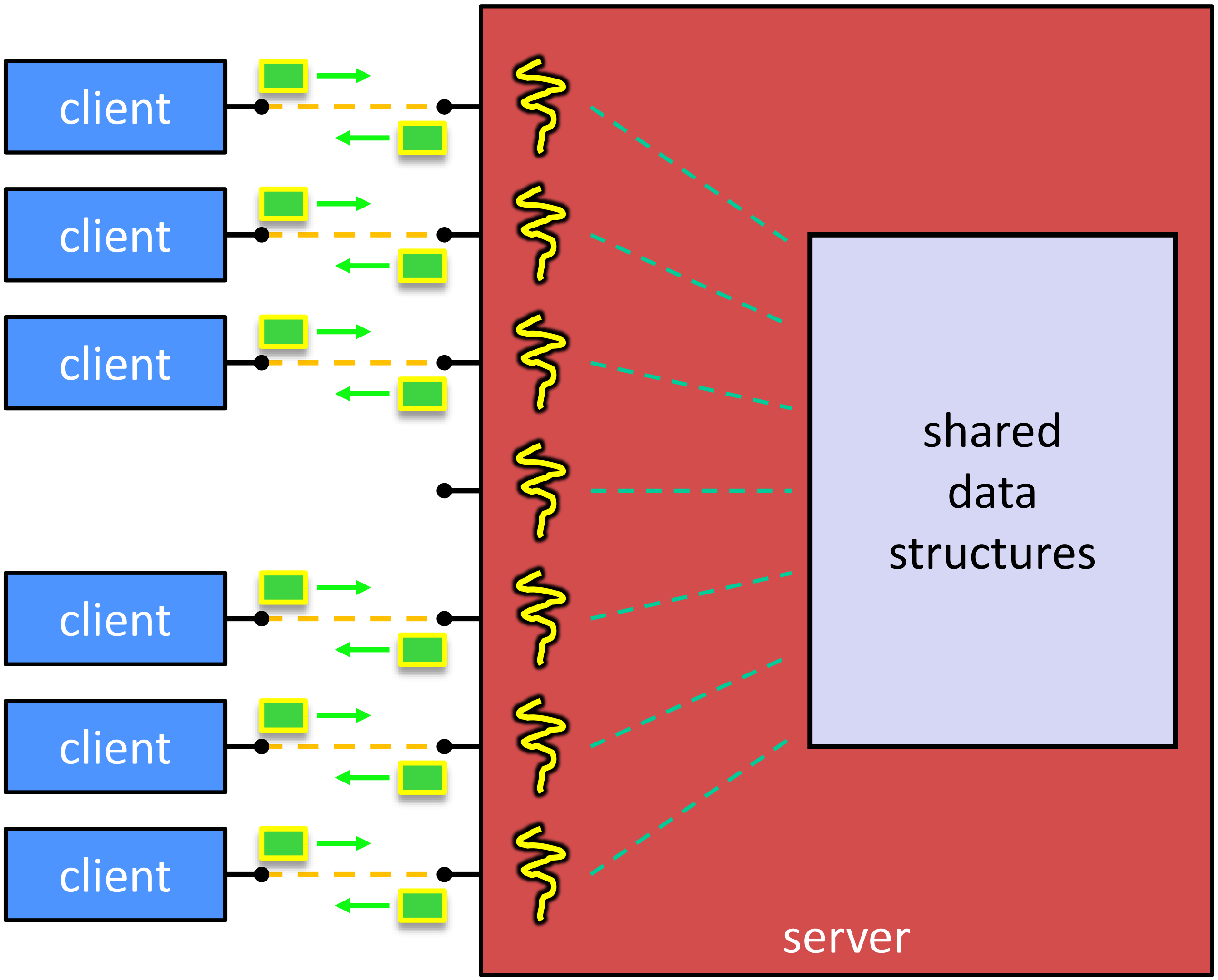
Multithreaded Server



Multithreaded Server



Multithreaded Server



Thread Examples

- See `cthread.c`
  - How do you properly handle memory management?
    - Who allocates and deallocates memory?
    - How long do you want memory to stick around?

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

- See `pthread.cc`
  - More instructions per thread = higher likelihood of interleaving

Why Concurrent Threads?

- Advantages:
  - Almost as simple to code as sequential
    - In fact, most of the code is identical! (but a bit more complicated to dispatch a thread)
  - Concurrent execution with good CPU and network utilization
    - Some overhead, but less than processes
  - Shared-memory communication is possible

- Disadvantages:
  - Synchronization is complicated
  - Shared fate within a process
    - One "rogue" thread can hurt you badly

Data Races

- Two memory accesses form a data race if different threads access the same location, and at least one is a write, 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 (!milk) {
  buy milk
}
```

- If you live alone:

- If you live with a roommate:

Poll Everywhere

pollev.com/cse33320su

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

```
if (!note) {
  if (!milk) {
    leave note
    buy milk
    remove note
  }
}
```

A. Yes, problem fixed

B. No, could end up with no milk

C. No, could still buy multiple milk

D. We're lost…

Poll Everywhere

pollev.com/cse33320su

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

We can be interrupted between checking note and leaving note ☹

```
if (!note) {
  if (!milk) {
    leave note
    buy milk
    remove note
  }
}
```

A. Yes, problem fixed

B. No, could end up with no milk

C. No, could still buy multiple milk

D. We're lost…

*There are other possible scenarios that result in multiple milks

you	roommate
Check note	
	Check note
Check milk	
Leave note	
	Check milk
	Leave note
	Buy milk
Buy milk	

time

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! ☠

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:
  - Liveness – ability to execute in a timely manner (informally, "something good eventually happens")
  - Safety – avoid unintended interactions with shared data structures (informally, "nothing bad 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

- Pseudocode:

```
// non-critical code

lock.acquire();    loop/idle if locked
// critical section
lock.release();

// non-critical code
```

- 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

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)

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

```
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`

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

  - Initializes a mutex with specified attributes

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

  - Acquire the lock – blocks if already locked Un-blocks when lock is acquired

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

  - Releases the lock

```
int pthread_mutex_destroy(pthread_mutex_t* mutex);
```

  - "Uninitializes" a mutex – clean up when done

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

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

- Definitely use in C++11 code if local conventions allow, but pthreads will be around for a long, long time
  - Use pthreads in current exercise