CSE 374: Programming Concepts and Tools

Spring 2026
Instructor: Megan Hazen

Today's Goals

Subject Matter

- Function pointers
- Overview of concurrency

Your Goals

- Finish HW 7
- Work on reference sheet for assessment 3
  - (double sided 8.5x11 paper)

Function Pointers

Code Globals Heap-> <-Stack

- Code also lives in memory
  - That means we can hold an address where a function's code resides
- **<return_type> (*<pointer_name>) (function_arguments);**
- **Set equal to 'address of function' (&f)**

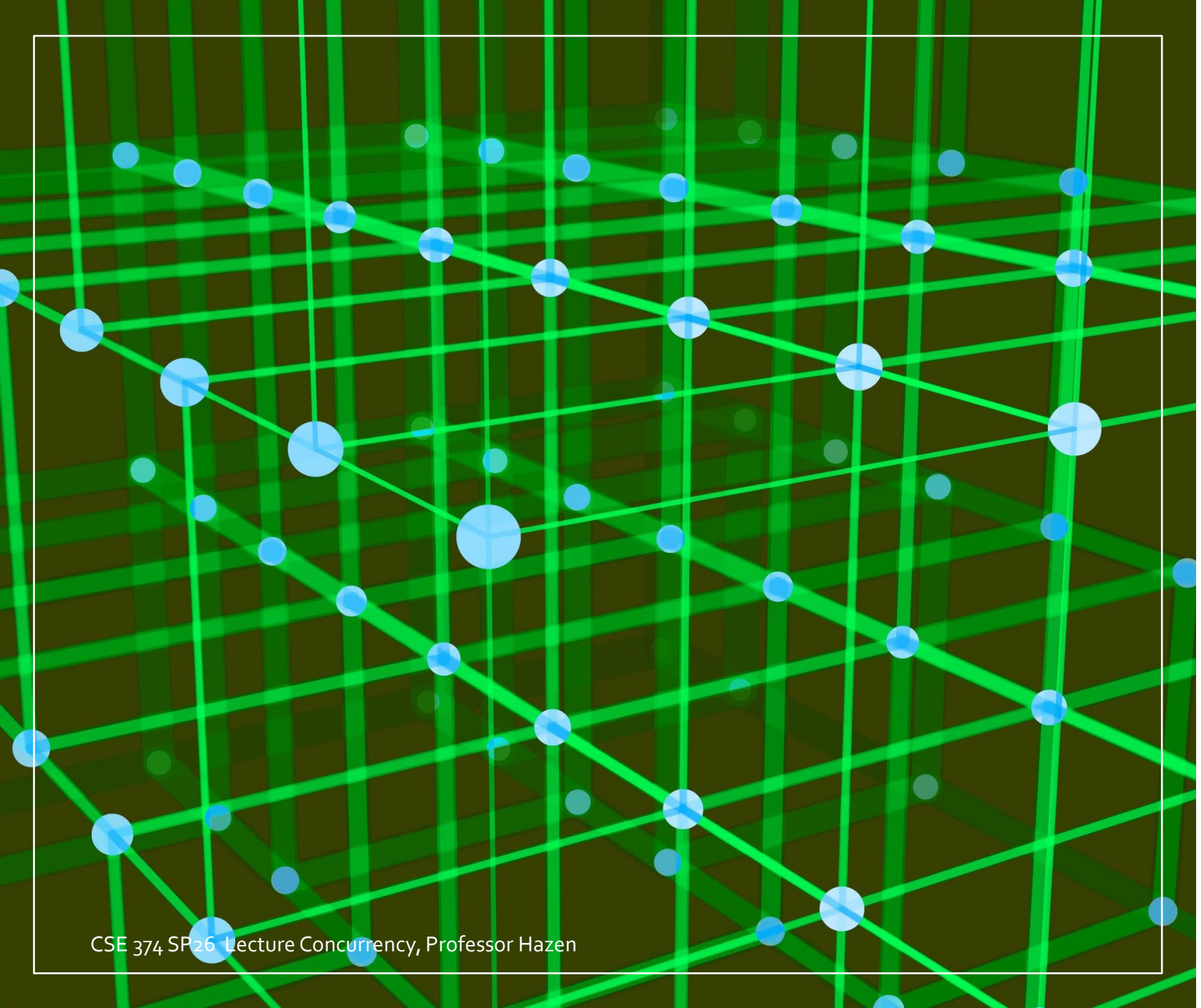
```
double two(double x) {
  return 2.0;
}
double integrate(double (*f)(double), double lo,
  double hi, double delta) {
    //    ans += (*f)(x) * ((hi-lo) / (n+1));
    ans += f(x) * ((hi-lo) / (n+1));
    ….
int main() {
  integrate(&two, 0.0, 2.0, 1.0));
```

Function Pointers 2

```
double integrate(double (*f)(double), double lo,
  double hi, double delta) {
  int I, x;
  double ans = 0.0;
  int n = (int)(0.5+(hi-lo)/delta);
  for (i = 0; i ≤ n; i++) {
    x = lo + i*delta;
    ans += (*f)(x) * ((hi-lo) / (n+1));
  }
  return ans;
}
int main() {
  printf("sin(x) from 0 to pi/2 at 0.01 = %e\n",
         integrate(&sin, 0.0, PI/2.0, 0.01));
  return 0; }
```

```
/* Define fdd as a double→double function */
typedef double (*fdd)(double);

double integrate(fdd f, double lo, double hi,
  double delta) {
  int I, x;
  double ans = 0.0;
  int n = (int)(0.5+(hi-lo)/delta);
  for (i = 0; i ≤ n; i++) {
    x = lo + i*delta;
    ans += f(x) * ((hi-lo) / (n+1));
  }
  return ans;
}
int main() {
  printf("sin(x) from 0 to pi/2 at 0.01 = %e\n",
         integrate(sin, 0.0, PI/2.0, 0.01));
  return 0; }
```



CONCURRENCY

What is concurrency?

- Running multiple processes simultaneously
  - Running separate programs simultaneously
  - Running two different 'threads' in one program
- Each 'process' is one 'thread'
- Parallelism refers to running things simultaneously on separate resources (ex. Separate CPUS)
- Concurrency refers to running multiple processes on SHARED resources

Allows processes to run 'in the background'

- Responsiveness - allow GUI to respond while computation happens
- CPU utilization - allow CPU to compute while waiting (for data, input, etc)
- Isolation - keep threads separate so errors in one don't affect the others

We do this all the time!

'Nice' linux parallel processes

NAME: nice - run a program with modified scheduling priority

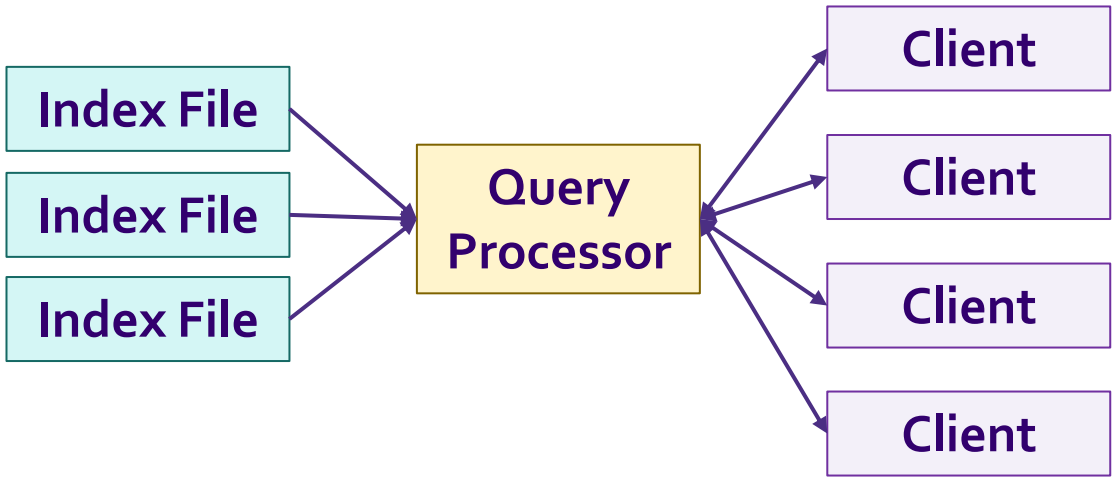
SYNOPSIS: nice [OPTION] [COMMAND [ARG]...]

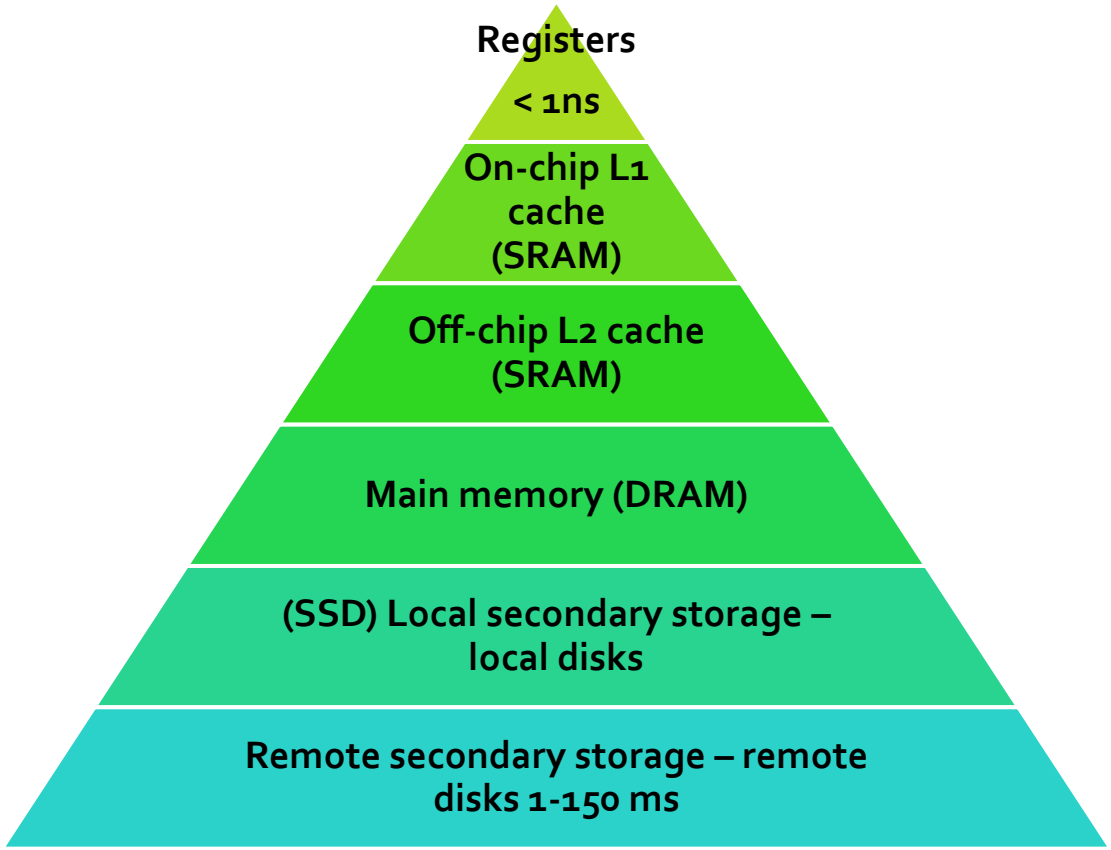
DESCRIPTION

Run COMMAND with an adjusted niceness, which affects process scheduling. With no COMMAND, print the current niceness. Niceness values range from -20 (most favorable to the process) to 19 (least favorable to the process).

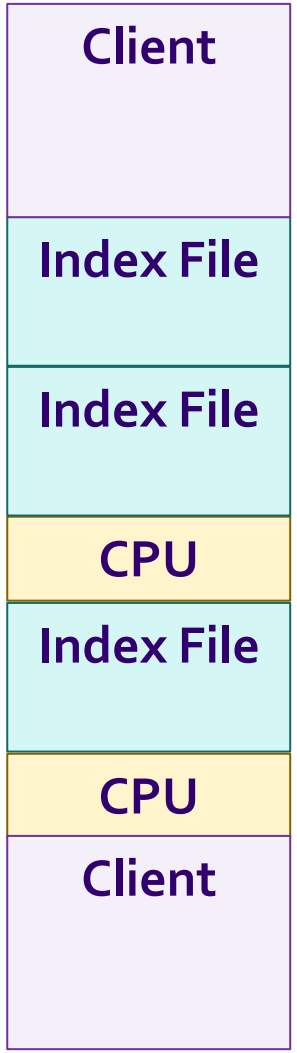
Where else?

What about web queries?



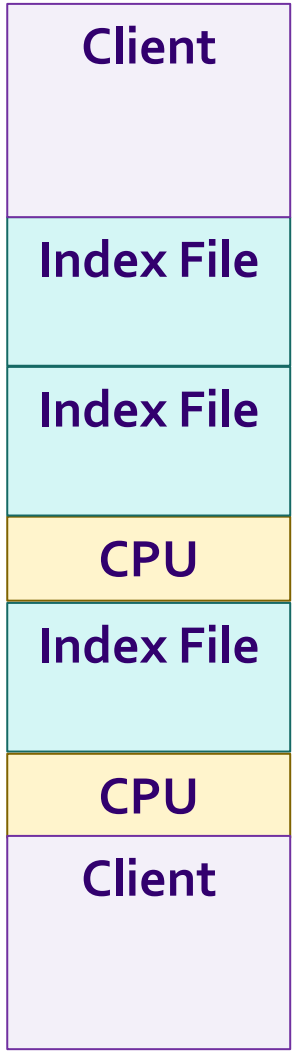


What about web queries? – one query



- In a single query, with I/O interleaved, the CPU is working almost none of the time. (This is not to scale)

What about web queries? – sequential queries



- Only one query is processed at a time
  - All others must wait
- The CPU is idle most of the time
  - It must wait for I/O which is very slow
- At most one I/O operation happens at once
  - Can't parallelize separate devices

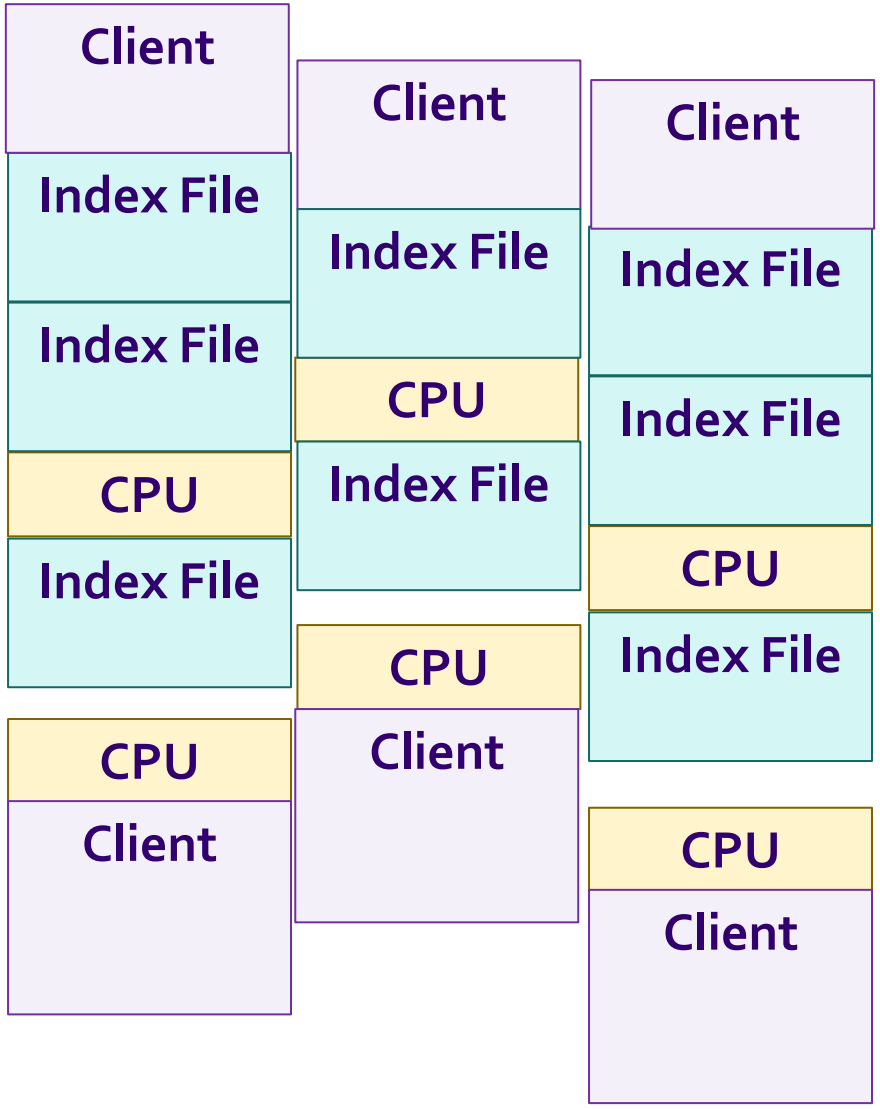
So, concurrency!

- Running multiple processes simultaneously
  - Running separate programs simultaneously
  - Running two different 'threads' in one program
- Each 'process' is one 'thread'
- Parallelism refers to running things simultaneously on separate resources (ex. Separate CPUS)
- Concurrency refers to running multiple processes on SHARED resources

Web server could do many things at once.

- Process one query while another does I/O
- Execute many queries one at a time, but issue I/O request simultaneously

What about web queries? – threaded queries



- A new query can start as soon as a cpu is available
- Queries continue unless they both need to see the same memory
- Each query is a thread

Threads

- Threads were formerly called "lightweight processes"
  - They execute concurrently like processes
    - OS's often treat them, not processes, as the unit of scheduling
    - Parallelism for free! If you have multiple CPUs/cores, can run them simultaneously
  - Unlike processes, threads cohabitate the same address space
    - Threads within a process see the same heap and globals and can communicate with each other through variables and memory
      - But can interfere with each other – need synchronization for shared resources
    - Each thread has its own stack
- What does the OS do when you switch processes?
  - How does that differ from switching threads?

Why Threads

- Advantages:
  - You (mostly) write sequential-looking code
  - Threads can run in parallel if you have multiple CPUs/cores

- Disadvantages:
  - If threads share data, you need **locks** or other **synchronization**
    - Very bug-prone and difficult to debug
  - Threads can introduce overhead
    - Lock contention, context switch overhead, and other issues
  - Need language support for threads

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

Alternative: Asynchronous I/O

- Use asynchronous or non-blocking I/O
- Your program begins processing a query
  - 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 query
  - The OS handles the details of issuing the read on the disk, or waiting for data from the console (or other devices, like the network)
  - When data becomes available, the OS lets your program know
- Advantage: Your program (almost never) blocks on I/O
- Disadvantage: You need to write more complex code to handle events

How do we actually make threads work?

- C, Java support concurrency similarly (other languages can be different)
  - one pile of code, globals, heap
  - multiple "stack + program counter"s — called threads
  - threads are run or pre-empted by a scheduler
  - threads all share the same memory
- Various synchronization mechanisms control when threads run
  - "don't run until I'm done with this"

Multi-threaded options

C: POSIX thread (pthreads)

- `#include <pthread.h>`
- Pass `-lpthread` to gcc (for linking)
- `pthread_create` takes a function pointer and arguments
  - Runs as a separate thread

Java: built into language

- Subclass `java.lang.Thread`
  - Override `run` method
- Create a thread object and call its `start` method
- Any object can be 'synchronized on'

Aside: POSIX

" The Portable Operating System Interface (POSIX)[1] is a family of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems. POSIX defines the application programming interface (API), along with command line shells and utility interfaces, for software compatibility with variants of Unix and other operating systems.[2][3]" - Wikipedia

The C 'pthread' conforms to the POSIX standard for threading.

pthread functions

```
Pthread_t threadID;
```

The threadID keeps track of which thread we are referring.

```
int pthread_create(pthread_t *thread, const pthread_attr_t
 *attr, void *(*start_routine)(void*), void *arg);
```

https://man7.org/linux/man-pages/man3/pthread_create.3.html

Note - pthread_create takes two generic (untyped) pointers interprets the first as a function pointer and the second as an argument pointer. This kicks off a new thread.

```
int pthread_join(pthread_t thread, void **value_ptr);
```

Puts calling thread 'on hold' until *'thread'* completes - useful for waiting to thread to exit

`pthread` actions

- Create: make a new thread with given attributes
  - Returns status code (0 or error)
  - Runs `start_routine (arg)`
- Exit: equivalent of an exit statement
  - Automatic when it returns from `start_routine`
- Join: Waits for thread to terminate and puts thread exit code in `retval`
- Detach: Let's a thread go, and cleans up resources afterwards.

```
int pthread_create(
      pthread_t* thread,
      const pthread_attr_t* attr,
      void* (*start_routine)(void*),
      void* arg);


void pthread_exit(void* retval);


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


int pthread_detach(
                   pthread_t thread);
```

Threaded memory

Threads are like lightweight processes

- They execute concurrently like processes
- Multiple threads can run simultaneously on multiple CPUs/cores
- Unlike processes, threads cohabitate the same address space
  - Threads within a process see the same heap and globals and can communicate with each other through variables and memory
    - But they can interfere with each other – need synchronization for shared resources
  - Each thread has its own stack

This sounds complicated….

(ex. pthreadex.c)

- If one thread did nothing of interest to any other thread, why bother running?
- Threads must communicate and coordinate
  - Use results from other threads, and coordinate access to shared resources
- Simplest ways to not mess each other up:
  - Don't access same memory (complete isolation)
  - Don't write to shared memory (write isolation)
- Next simplest:
  - One thread doesn't run until/unless another is done

Taking care of memory (races!)

- If your fridge has no milk, go buy some more?
- What could go wrong?
- If you live alone? Maybe nothing

- If you have a roommate?

```
if (!milk) {
  buy milk
}
```

Taking care of memory – what to do?

- If your fridge has no milk, go buy some more?
  But leave a note first?

- Is the problem fixed?

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

Memory – race walk through

Ada	Bob
!note !milk	
	!note !milk
	Leave note Buy milk Remove note
Leave note Buy milk Remove note	

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

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!

Synchronicity

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

Goals of synchronization:

- **Liveness** – ability to execute in a timely manner (informally, "something good 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

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

```
// non-critical code

lock.acquire();
// critical section
lock.release();

// non-critical code
```

Milk – what is a '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()
```

`pthread` & Locks

Another term for a lock is a mutex ("mutual exclusion")

`pthread.h` defines datatype **`pthread_mutex_t`**

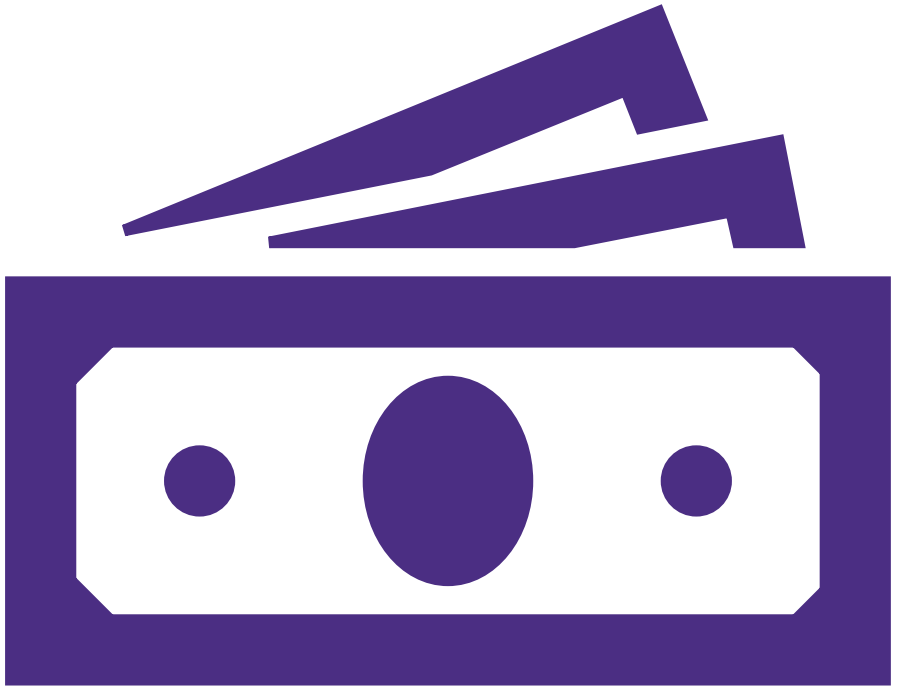
- Initializes a mutex with specified attributes
- Acquire the lock – blocks if already locked
- Releases the lock
- "Uninitializes" a mutex – clean up when done

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

int pthread_mutex_lock(
                    pthread_mutex_t* mutex);

int pthread_mutex_unlock(
                    pthread_mutex_t* mutex);

int pthread_mutex_destroy(
                    pthread_mutex_t* mutex);
```



DEMO

Bank Accounts

(BankAccountThread.[h,c])

BankAccount

Bank Accounts may have many users interacting with them (or other accounts in the same system) simultaneously.

It would be bad to interrupt a withdraw or transfer in process.

```
#include <atomic>
#include <mutex>
class BankAccount {
 public:
  BankAccount();
  void deposit(double amount);
  void withdraw(double amount);
  double getBalance();
  void setBalance(double amount);
  void transferTo(double amount, BankAccount& other);

 private:
  // This function is only called when mutex is already locked.
  void setBalanceWithLock(double amount);
  const int accountId_;

  // This mutex protects access to the balance_.
  std::mutex m_;
  double balance_;
  // Atomic so when we get a new id, we don't have any races
  static std::atomic<int> accountCount_;
};
```

Bank account data races

- This code is correct in a sequential program
- It may have a **race condition in a concurrent program**, allowing for a negative balance
- Discovering this bug with testing is very hard

```
void BankAccount::withdraw(double amount) {
  double b = getBalance();
  if (amount > b) {
    throw std::invalid_argument(
                              "can't withdraw");
  }
  setBalance (b - amount);
}
```

Withdraw race example

Thread 1

- Balance == $150
- T1 tries to withdraw $100
- Checks balance – OK!

- T1 withdraws $100

Thread 2

- Balance == $150

- T2 tries to withdraw $100
- Checks balance – OK!
- T2 withdraws $100

Bank account data races – fix with `mutex`

- An operation we want to be done all at once
  - No interruptions
- Note: Must be the right size
  - Too big - program runs sequentially
  - Too small - program has potential races

```
void BankAccount::withdraw(double amount) {
  std::lock_guard<std::mutex> lock(m_);
  double b = getBalance();
  if (amount > b) {
    throw std::invalid_argument(
                        "can't withdraw");
  }
  setBalanceWithLock(b - amount);
 // m_ unlocks when it goes out of scope
}
```

C `mutex` lock

1. Create a lock for specific data
2. Lock before atomic part of code
3. Unlock after atomic operation

- What happens if more than one piece of code affects the data?
- Idea: Use same mutex (‘`m`’) for each piece of code that modifies ‘`balance_`’

C++ lockguards

- A "lock guard" is an object that
  - Locks the mutex in the constructor
  - Unlocks in the destructor

```
std::lock_guard<std::mutex> lock(m_);
```

- If the lock guard is added to the stack it is locked upon creation
- Mutex is unlocked when object is removed from the stack; even correctly responding for an exception.

Bank account deadlocking

Every piece of code that refers to a datum calls the lock for that datum

If **`withdraw`** locks **`balance`**, and then calls **`setBalance`** which also must lock **`balance`**, we get a deadlock - we can't progress because **`setBalance`** can not complete

One solution is to write a helper function to replace **`setBalance`** – **`setBalanceWithLock`**

```
void BankAccount::withdraw(double amount) {
  std::lock_guard<std::mutex> lock(m_);
  double b = getBalance();
  if (amount > b) {
    throw std::invalid_argument(
                          "can't withdraw");
  }
  setBalanceWithLock(b - amount);
}
void BankAccount::setBalance(double balance) {
  std::lock_guard<std::mutex> lock(m_);
  setBalanceWithLock(balance);
}
void BankAccount::setBalanceWithLock(double amount) {
  balance_ = amount;
}
```

Bank account more deadlocking

Different kind of deadlock:

- Must lock the value on each account.
- But what happens if one transfer is started from account A to account B while a simultaneous transfer is started from account B to account A?

```
void BankAccount::transferTo(double amount,
                             BankAccount& other) {
  std::mutex* first = &m;
  std::mutex* second = &other.m;

  std::lock_guard<std::mutex> lock1(*first);
  std::lock_guard<std::mutex> lock2(*second);

  if (getBalance() < amount) {
    throw std::invalid_argument("can't transfer, not enough
  funds");
  }
  setBalanceWithLock(getBalance() - amount);
  other.setBalanceWithLock(other.getBalance() + amount);
}
```

Fixes? Bank account more deadlocking

- **Use smaller critical sections.** Lock A's mutex only around the modification of A's balance, and lock B's mutex when modifying B's balance.
  - But - we expose an intermediate state in which A's account has been debited but the funds haven't been put in B's account yet - we've temporarily lost money, which isn't great.
- **Use larger critical sections.** Add a single lock for all bank accounts that must be acquired before doing multi-account transactions.
  - But it means that we can only do one transaction at a time throughout the entire bank, even if the accounts aren't related to each other. This is a performance loss.
- **Always lock mutexes in a specific order.** We can choose to always lock the mutex of the account with the lower account id first, then lock the id of the higher account id. This works because account ids are unique and immutable, thus we can rely on them without synchronization.

FIXED! Bank account more deadlocking

Mutex locks are set in a specific order – always the lower account id first.

```
void BankAccount::transferTo(double amount, BankAccount& other) {
  std::mutex* first;
  std::mutex* second;
  if (accountId_ < other.accountId_) {
    first = &m_;
    second = &other.m_;
  } else {
    first = &other.m_;
    second = &m_;
  }
  std::lock_guard<std::mutex> lock1(*first);
  std::lock_guard<std::mutex> lock2(*second);
  if (getBalance() < amount) {
    throw std::invalid_argument("can't transfer, not enough
 funds");
  }
  setBalanceWithLock(getBalance() - amount);
  other.setBalanceWithLock(other.getBalance() + amount);
}
```

C++ `atomic`

- Even single line integer operations (`++accountCount`) may be subject to race conditions.
- Instead of manually locking and unlocking every integer operation, can make the data declaration `std::atomic`
- Atomic renders that variable safe for read/write operations.
- Atomic uses a hardware primitive

```
Header:
#include <atomic>
static std::atomic<int> accCount_;

Module:
BankAccount::BankAccount()
  : accountId_(++accCount_),
    balance_(0) {}
```

Other types of locks

- There are other types of locks and primitives that are useful, besides the regular mutex, lock guard, and std::atomic:
- **Reentrant locks.** We had a problem earlier where one function that locked the mutex tried to call another function that would lock the same mutex, but this didn't work because the first function already had the lock! Use this behavior with a "reentrant lock": the same thread may re-lock the same lock any number of times. The lock will be released to a different thread once all of the lock() calls have been correspondingly unlock()'ed. Re-entrant locks can be difficult to trace.
- **Reader-writer locks.** All of the problems that we've seen so far have resulted by read/write or write/write combinations of calls. It is only the writing that causes problem. To improve efficiency, you might use "reader-writer locks": these allow multiple threads to read the same data at a time, but if any thread tries to write, it will make sure that no other thread is either reading or writing at the same time. This improves the performance of reads (allowing them to happen at once) while still maintaining correctness of the program.
- **Condition variables.** Let's say you are trying to dequeue from a queue, but there's no data in the queue at the moment. You want to wait until some other thread inserts into the queue, then you can wake up and dequeue that element! In this case you can use a "condition variable": a primitive that can be used to block a thread until another thread notifies the condition variable that the waiting condition has been satisfied.

C++ C11 standard 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

Memory Take-aways

For every memory location, you should obey at least one of the following:

- Make it **thread-local**. Whenever possible, avoid sharing resources between threads - make a copy for each thread. If threads do not need to communicate with each other through the shared resource (for example, a random-number generator), then make it thread-local. In typical concurrent programs, the vast majority of objects should be thread-local.
  - Shared-memory should be rare - minimize it.
- Make it **immutable**. Whenever possible, do not update objects; make new objects instead. If a location is only read (never written), then no synchronization is necessary. Simultaneous reads are not data races, and not a problem.
  - In practice, programmers over-use mutation - minimize it.
- Make access **synchronized**, ie use locks and other primitives to prevent race conditions.

Synchronization Take-aways

- **No data races.** Never allow two threads to read/write or write/write a location at the same time.
- **Think of what operations need to be atomic.** Consider atomicity first, then figure out how to implement it with locks).
- **Consistent locking.** For each location that should be synchronized, have a lock that is ALWAYS locked when reading or writing that location. The same lock may (and often should) be used to guard multiple locations/pieces of memory. Clearly document with comments the mutex that guards a particular piece of memory.
- **Start with coarse-grained locking; move to finer-grained locking only if blocking for locks becomes an issue.** Coarse-grained locking is the practice of having fewer locks: one for the whole data structure, or one for all bank accounts. It is simpler to implement, but performance can be bad (fewer operations can be done at the same time). But if there isn't a lot of concurrent access, then coarse locking is probably fine. Fine-grained locking is the practice of having more locks, each guarding less data: one lock per data element, or one lock per field in the bank account. Fine-grained locking is trickier to get correct, requires more programming, and has more overhead (more locks to lock), but it we can do more things at once.
- **Don't do expensive computations or I/O in critical sections, but also don't introduce race conditions.** This balances performance with correctness.
- **Use built-in libraries whenever possible.** Concurrency is extremely tricky and difficult to get right; experts have spent countless hours building tools for you to use to make your code safe.