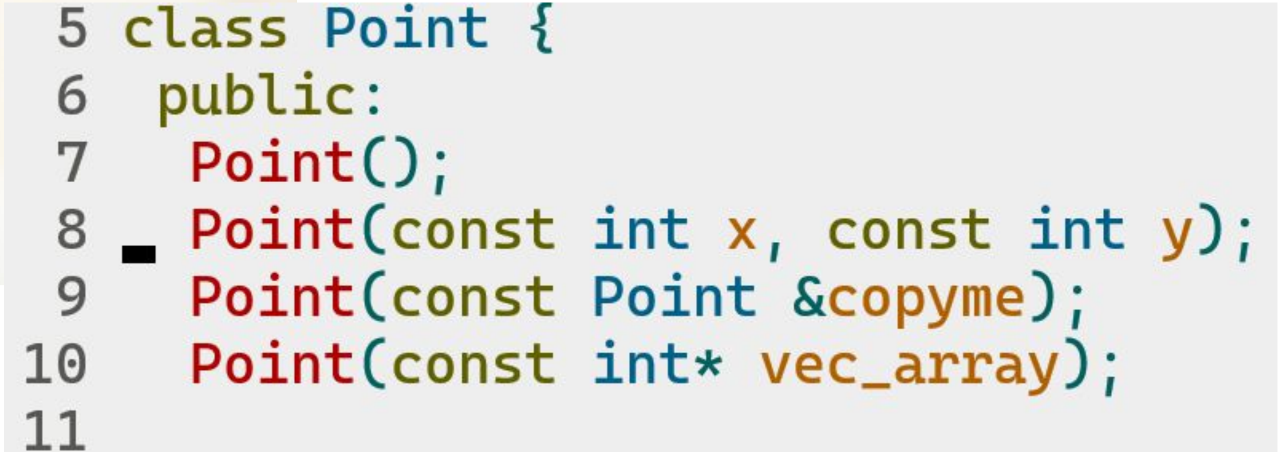
What do you think?

Name that Constructor!

What type of constructors do you see below? How do you know?



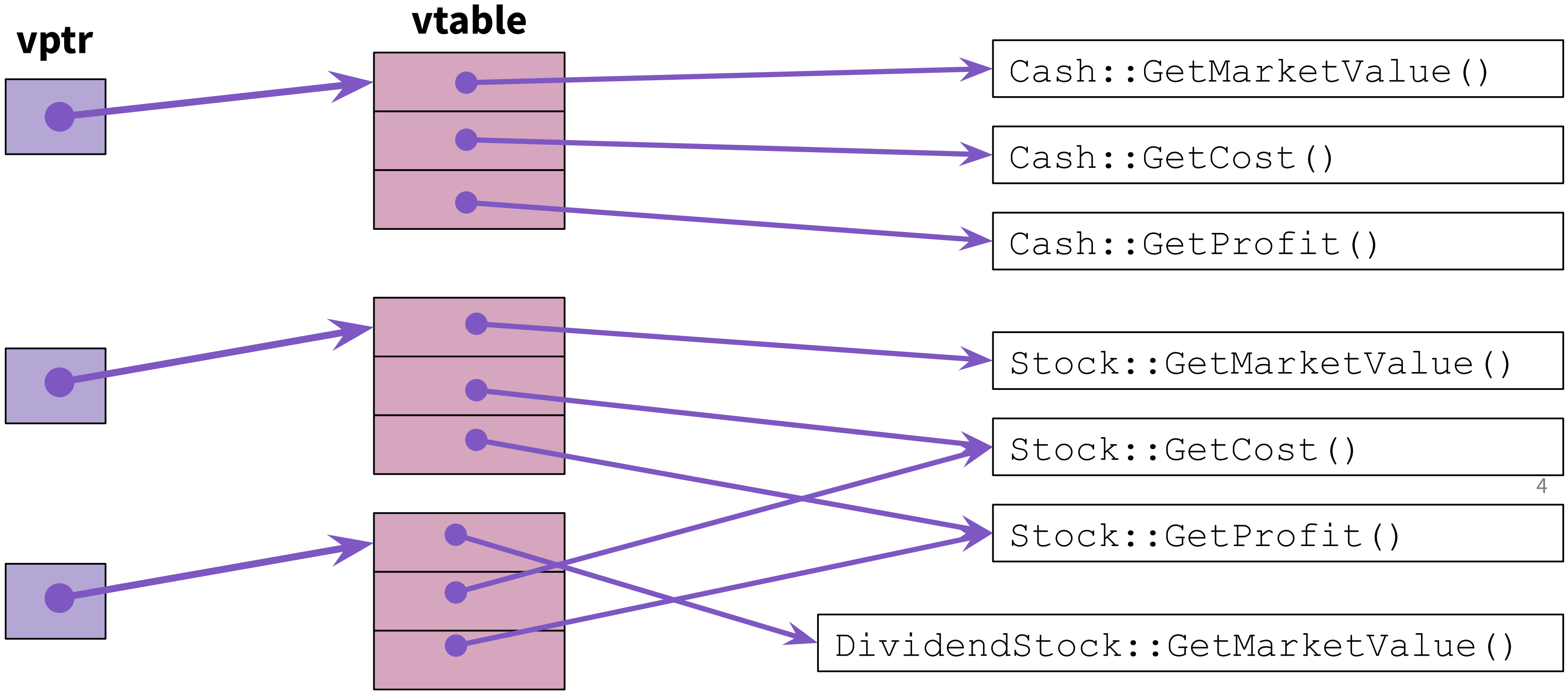
```
class Point {
 public:
  Point();
  Point(const int x, const int y);
  Point(const Point &copyme);
  Point(const int* vec_array);

```

CSE 374: Lecture 27

Concurrency

Function Pointers

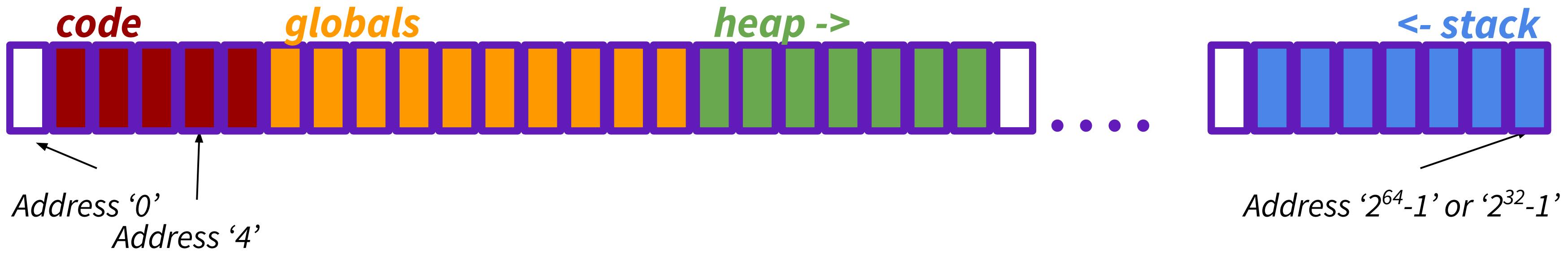
vptr and vtable Visualization



Function Pointers

Can point to code the way we point to data. In C, the syntax is:

```
<return_type> (*<pointer_name>) (function_arguments);
Set equal to 'address of function' (&f)
```



Function Pointers

Can point to code the way we point to data.

In C, the syntax is:

`<return_type> (*<pointer_name>) (function_arguments);`

`Set equal to 'address of function' (&f)`

```
double two(double x) {
  return 2.0;
}

printf("int two(x) = %e\n",
     integrate(&two, 0.0,
              2.0, 1.0));
```

```
double integrate(
    double (*f)(double),
    double lo, double hi,
    double delta) {
  ...
   ans += (*f)(x) *
          ((hi-lo) / ((n+1));
 ...
```

Function Pointers: nicer syntax

Typedef can be used to shorten datatype:

```
typedef double (*fdd)(double);  //fdd is function double-double
```

The C compiler is smart enough to know what is a function and what is a variable:

```
ans += (*f)(x) * ((hi-lo) / (n+1)); // eval. @location of f
ans += f(x) * ((hi-lo) / (n+1));
```

Also interprets function name as a pointer to the code:

```
integrate(&sin, 0.0, PI/2.0, 0.01));
integrate(sin, 0.0, PI/2.0, 0.000001));
```

Code

```
float* optimize (float(*obj)(float*), float* mins, float* maxs);


int main() {

  float* opt;
  …
  printf("Start
  opt = optimiz
  …
  return 0;
}

// spherefunc m
float spherefun
   return pos[0]*pos[0] + pos[1]*pos[1];
}
```

Review Opportunity:

`float(*obj)(float*)`

-- function pointer from float* to float

Concurrency

Sequential Programming

Sequential programming demands finishing sequence before starting the next one

Previously, performance improvements could be made by improving hardware
- no longer (Goodbye Moore’s Law)

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 already do this!

‘Nice’ linux parallel processes

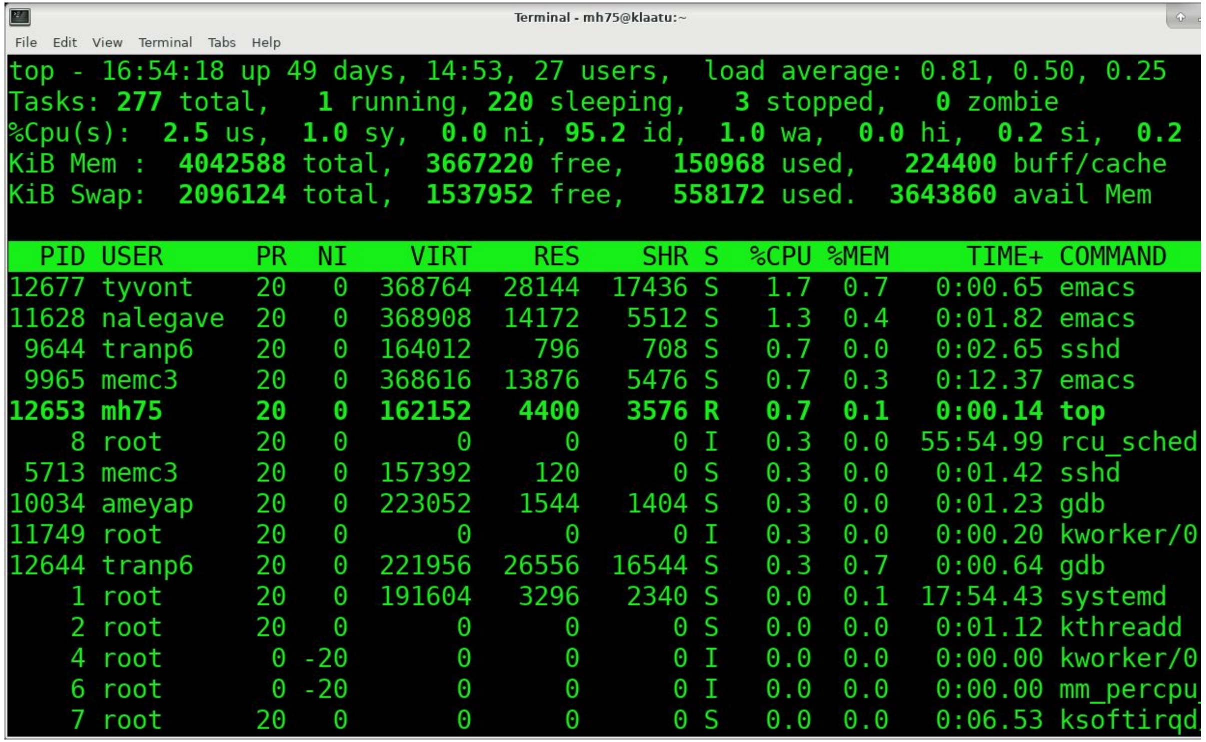
```
NAME
    nice - run a program with modified scheduling priority

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

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

Other Linux tools

Top - shows all processes with ‘niceness’ (NI)

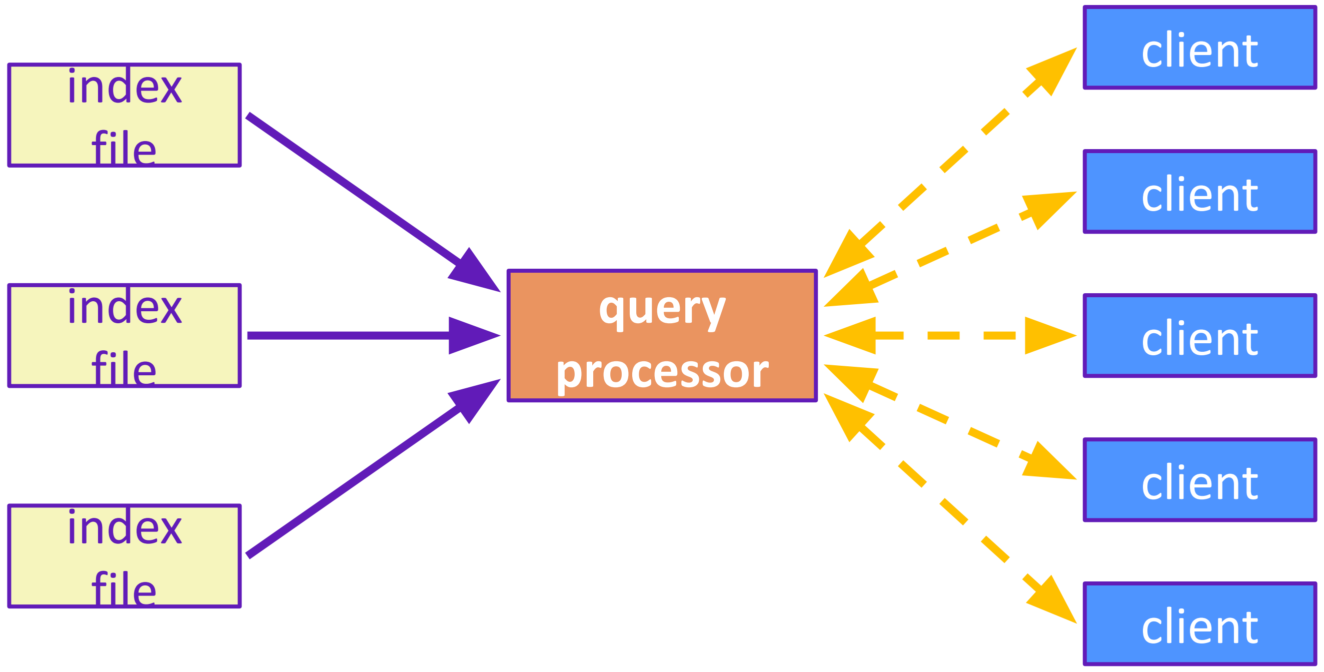
[mh75@klaatu ~]$ ps -o pid,comm,nice

PID COMMAND NI
11483 bash 0
13034 ps 0

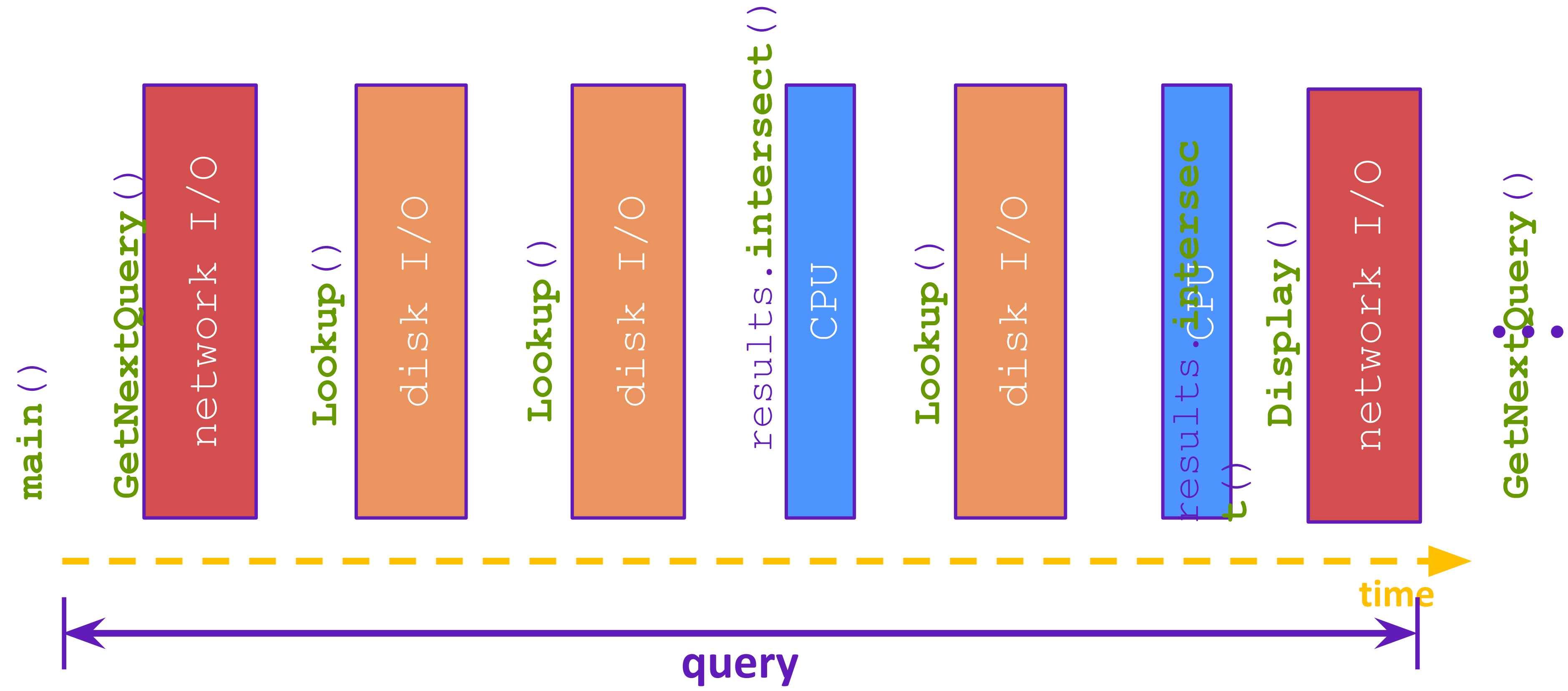


```
top - 16:54:18 up 49 days, 14:53, 27 users,  load average: 0.81, 0.50, 0.25
Tasks: 277 total,   1 running, 220 sleeping,   3 stopped,   0 zombie
%Cpu(s):  2.5 us,  1.0 sy,  0.0 ni, 95.2 id,  1.0 wa,  0.0 hi,  0.2 si,  0.2
KiB Mem :  4042588 total,  3667220 free,   150968 used,   224400 buff/cache
KiB Swap:  2096124 total,  1537952 free,   558172 used.  3643860 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
12677 tyvont    20   0  368764  28144  17436 S   1.7  0.7   0:00.65 emacs
11628 nalegave  20   0  368908  14172   5512 S   1.3  0.4   0:01.82 emacs
 9644 tranp6    20   0  164012    796    708 S   0.7  0.0   0:02.65 sshd
 9965 memc3     20   0  368616  13876   5476 S   0.7  0.3   0:12.37 emacs
12653 mh75      20   0  162152   4400   3576 R   0.7  0.1   0:00.14 top
    8 root      20   0       0      0      0 I   0.3  0.0  55:54.99 rcu_sched
 5713 memc3     20   0  157392    120      0 S   0.3  0.0   0:01.42 sshd
10034 ameyap    20   0  223052   1544   1404 S   0.3  0.0   0:01.23 gdb
11749 root      20   0       0      0      0 I   0.3  0.0   0:00.20 kworker/0
12644 tranp6    20   0  221956  26556  16544 S   0.3  0.7   0:00.64 gdb
    1 root      20   0  191604   3296   2340 S   0.0  0.1  17:54.43 systemd
    2 root      20   0       0      0      0 S   0.0  0.0   0:01.12 kthreadd
    4 root       0 -20       0      0      0 I   0.0  0.0   0:00.00 kworker/0
    6 root       0 -20       0      0      0 I   0.0  0.0   0:00.00 mm_percpu
    7 root      20   0       0      0      0 S   0.0  0.0   0:06.53 ksoftirqd
```

Web Search Architecture

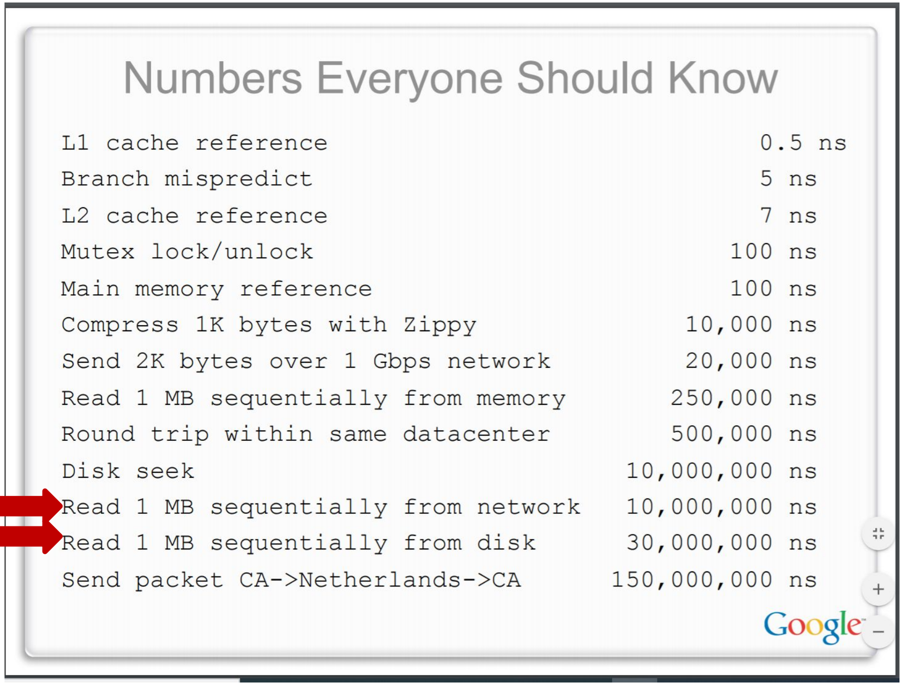


Execution Timeline: a Multi-Word Query



What About I/O-caused Latency?

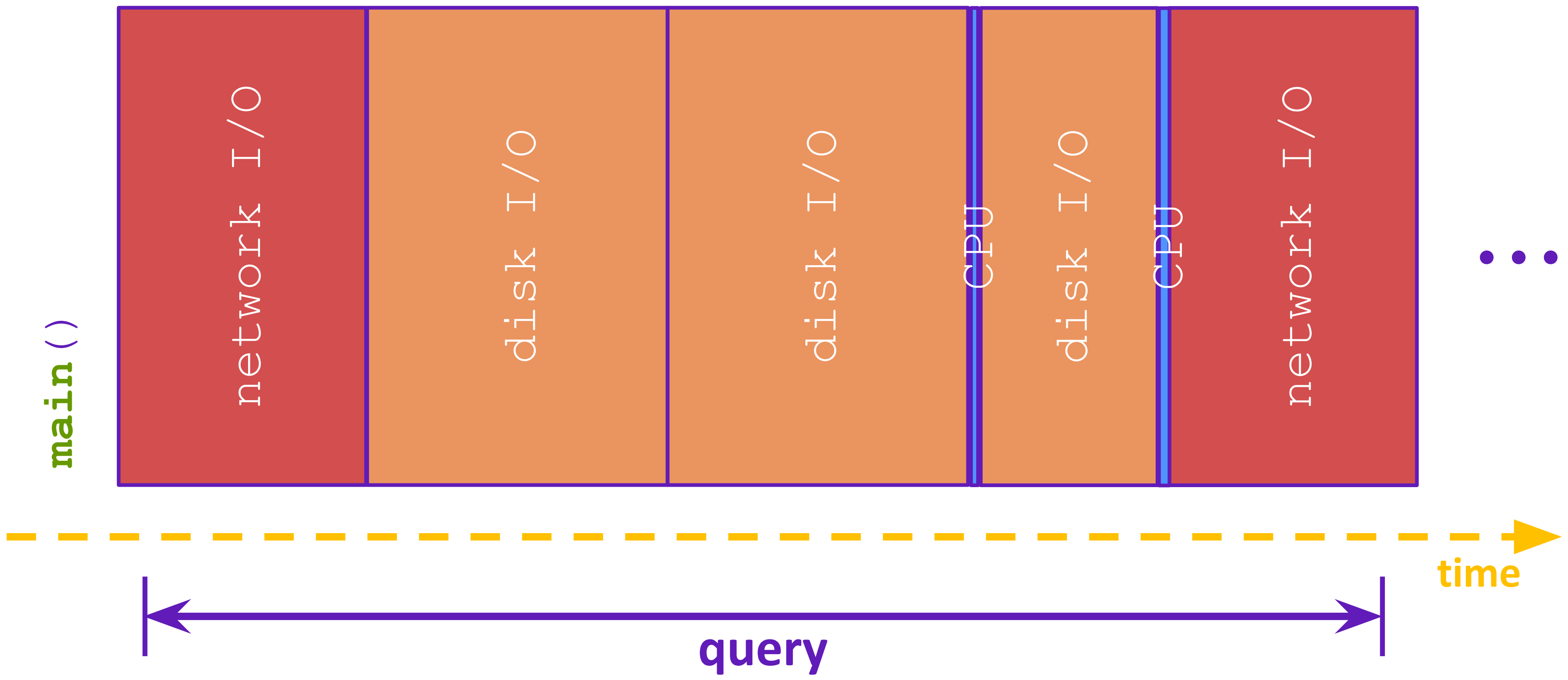
- Jeff Dean's "Numbers Everyone Should Know" (LADIS '09)



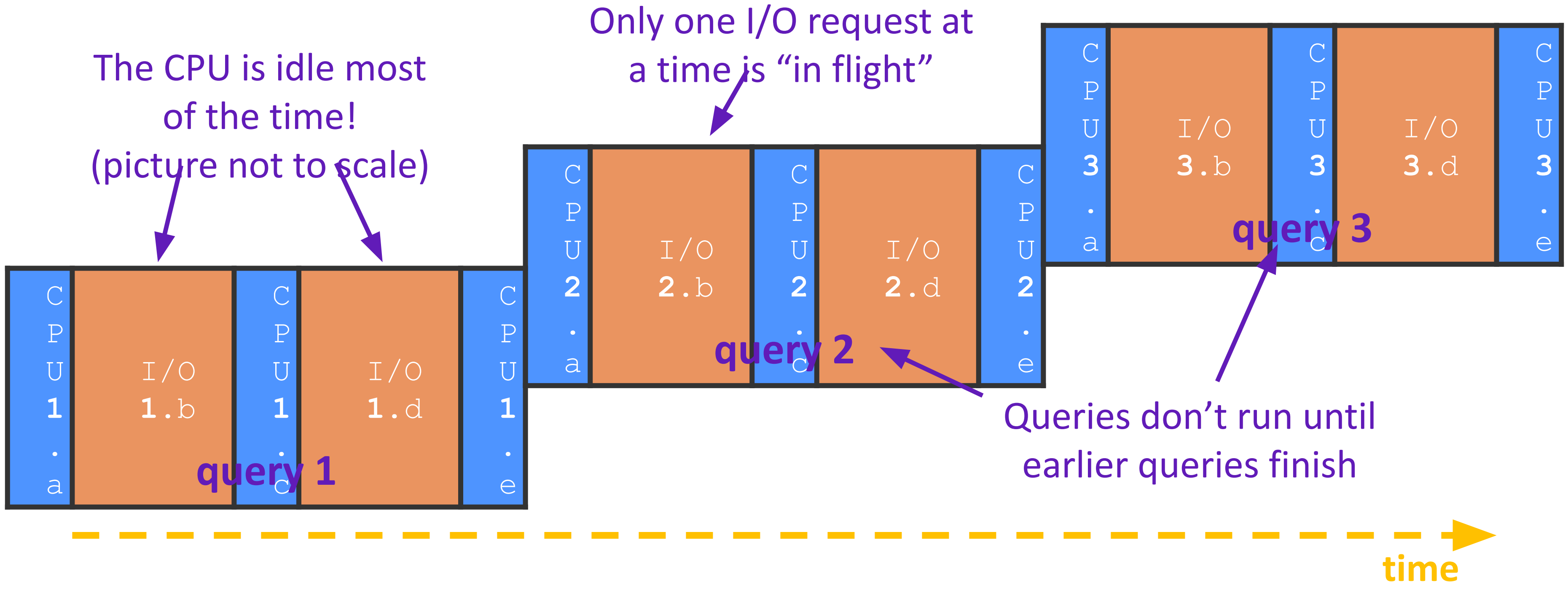
Numbers Everyone Should Know

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	100 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	10,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from network	10,000,000 ns
Read 1 MB sequentially from disk	30,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

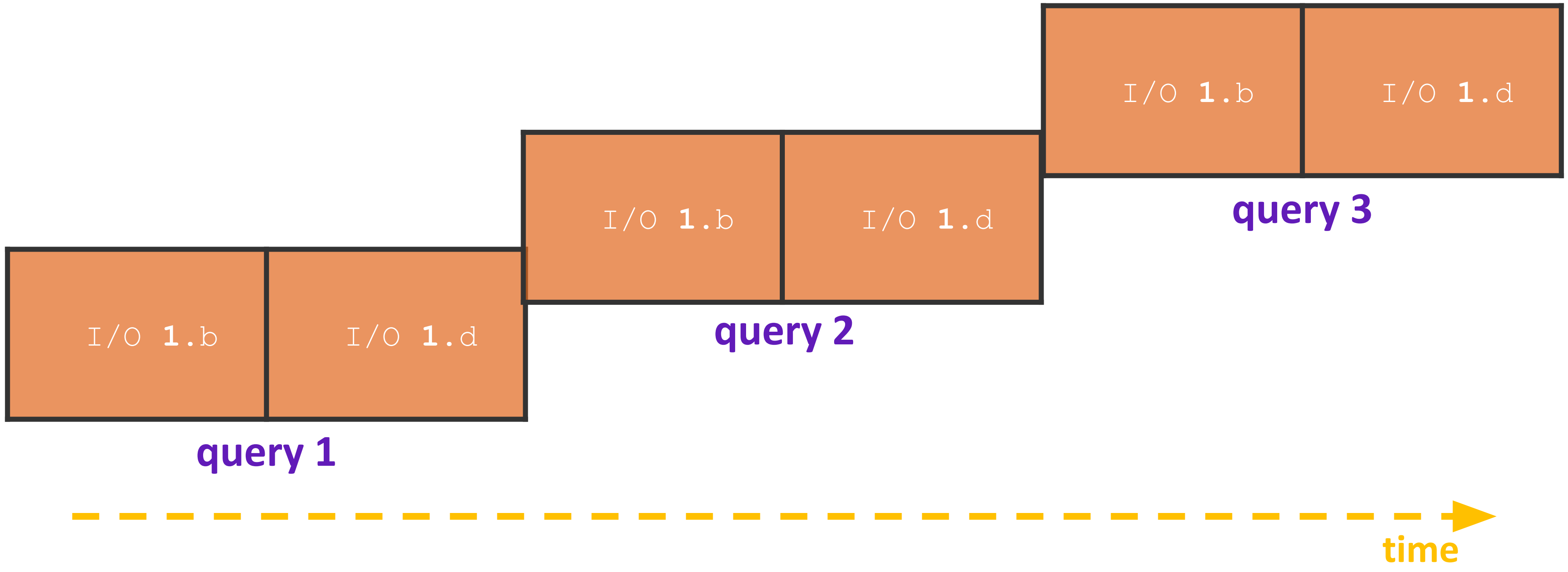
Execution Timeline: To Scale



Sequential Queries – Simplified


The CPU is idle most of the time! (picture not to scale)
Only one I/O request at a time is “in flight”
CPU 1.a
I/O 1.b
CPU 1.c
I/O 1.d
CPU 1.e
query 1
CPU 2.a
I/O 2.b
CPU 2.c
I/O 2.d
CPU 2.e
query 2
CPU 3.a
I/O 3.b
CPU 3.c
I/O 3.d
CPU 3.e
query 3
Queries don’t run until earlier queries finish
time

Sequential Queries: To Scale



Sequential Can Be Inefficient

- Only one query is being processed at a time
  - All other queries queue up behind the first one
- The CPU is idle most of the time
  - It is *blocked* waiting for I/O to complete
    - Disk I/O can be very, very slow
- At most one I/O operation is in flight at a time
  - Missed opportunities to speed I/O up
    - Separate devices in parallel, better scheduling of a single device, etc.

Concurrency

- A version of the program that executes multiple tasks simultaneously
    - Example: Our web server could execute multiple *queries* at the same time
        - While one is waiting for I/O, another can be executing on the CPU
    - Example: Execute queries one at a time, but issue *I/O requests* against different files/disks simultaneously
        - Could read from several index files at once, processing the I/O results as they arrive

- Concurrency != parallelism
    - Parallelism is when multiple CPUs work simultaneously on 1 job

A Concurrent Implementation

- Use multiple threads or processes
    - As a query arrives, fork a new thread (or process) to handle it
        - The thread reads the query from the console, issues read requests against files, assembles results and writes to the console
        - The thread uses blocking I/O; the thread alternates between consuming CPU cycles and blocking on I/O
- The OS context switches between threads/processes
    - While one is blocked on I/O, another can use the CPU
    - Multiple threads' I/O requests can be issued at once

Review: Processes

- To implement a “process”, the operating system gives us:
  - Resources such as file handles and sockets
  - Call stack + registers to support (eg, PC, SP)
  - Virtual memory (page tables, TLBs, etc …)
- If we want concurrency, what is the “minimal set” we need to execute a single line of code?

“Worker” 1

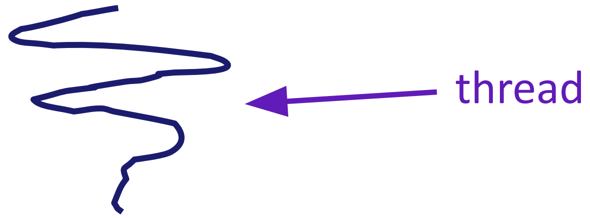
```
bucket = hash(word);
hitlist = file.read(bucket);
```

“Worker” 2

```
foreach hit in hitlist {
  doclist.append(file.read(hit));
}
```

Introducing Threads

- Separate the concept of a process from an individual “*thread of control*”
  - Usually called a thread (or a *lightweight process*), this is a sequential execution stream within a process



- In most modern OS’s:
  - Process: address space, OS resources/process attributes
  - Thread: stack, stack pointer, program counter, registers
  - Threads are the *unit of scheduling* and processes are their *containers*; every process has at least one thread running in it

Threads

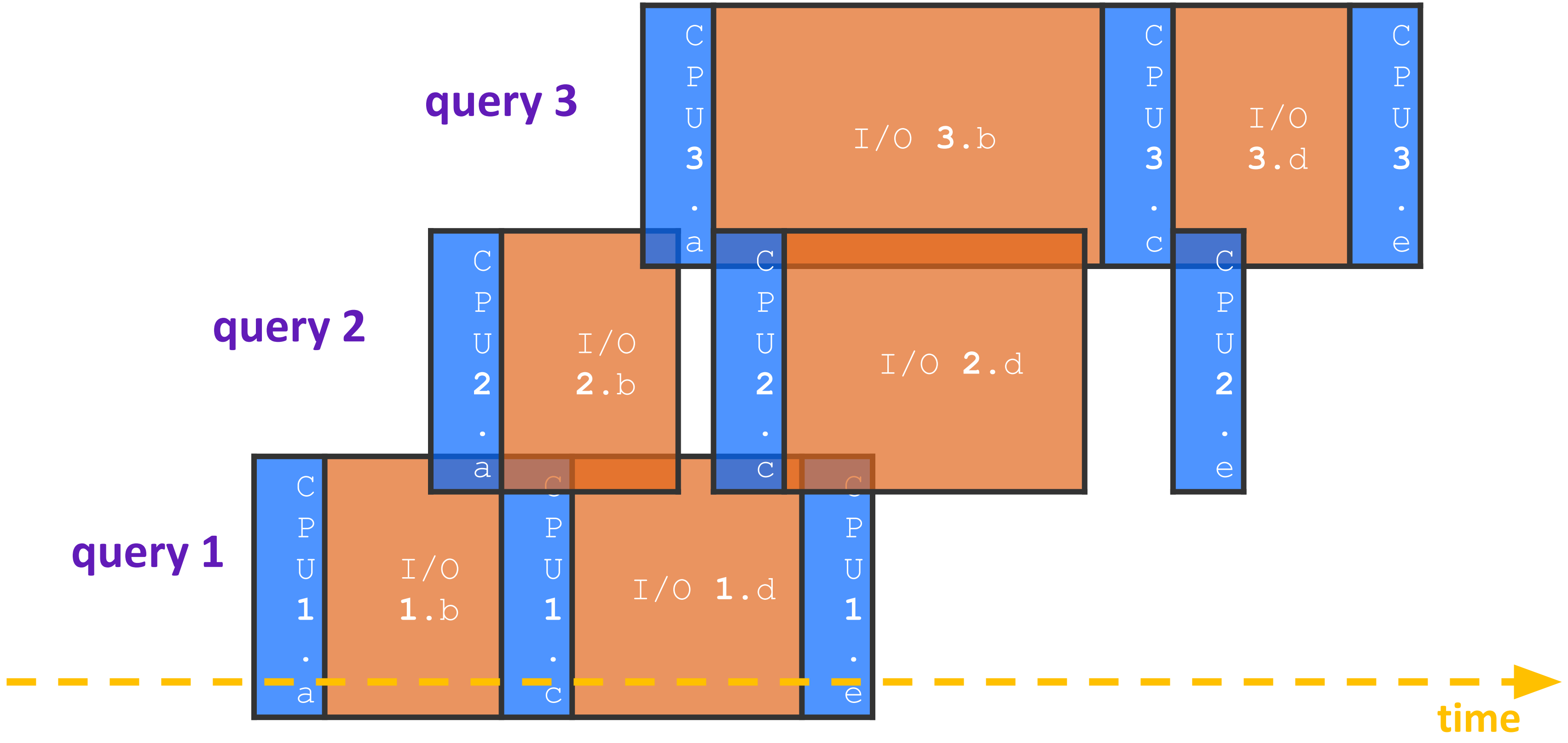
- Threads were formerly called “lightweight processes”
    - They execute concurrently like processes
        - OS’s often treat them, not processes, as the unit of scheduling
        - Parallelism for free! If you have multiple CPUs/cores, can run them simultaneously
    - Unlike processes, threads 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?

Multithreaded Pseudocode

```
main() {
  while (1) {
    string query_words[] = GetNextQuery();
    ForkThread(ProcessQuery());
  }
}
```

```
doclist Lookup(string word) {
  bucket = hash(word);
  hitlist = file.read(bucket);
  foreach hit in hitlist
    doclist.append(file.read(hit));
  return doclist;
}

ProcessQuery() {
  results = Lookup(query_words[0]);
  foreach word in query[1..n]
    results = results.intersect(Lookup(word));
  Display(results);
}
```

Multithreaded Queries – Simplified



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

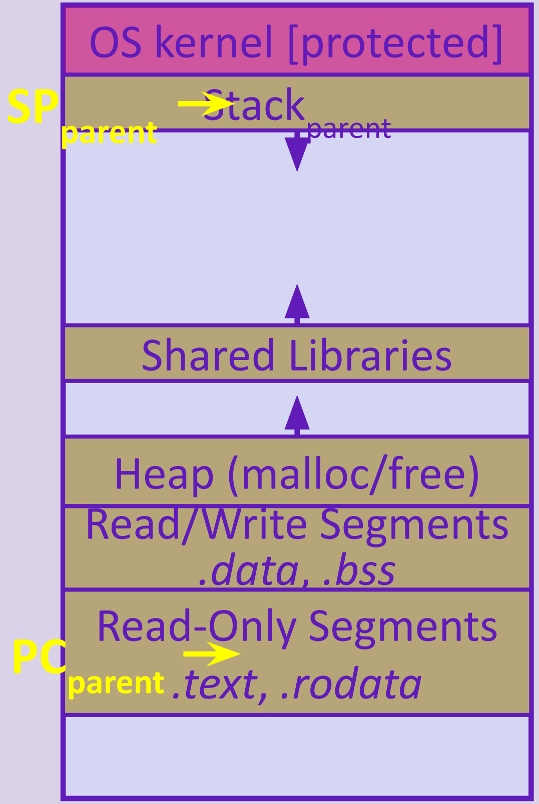
Alternative: Processes

- What if we forked processes instead of threads?

- Advantages:
  - No shared memory between processes
  - No need for language support; OS provides “fork”

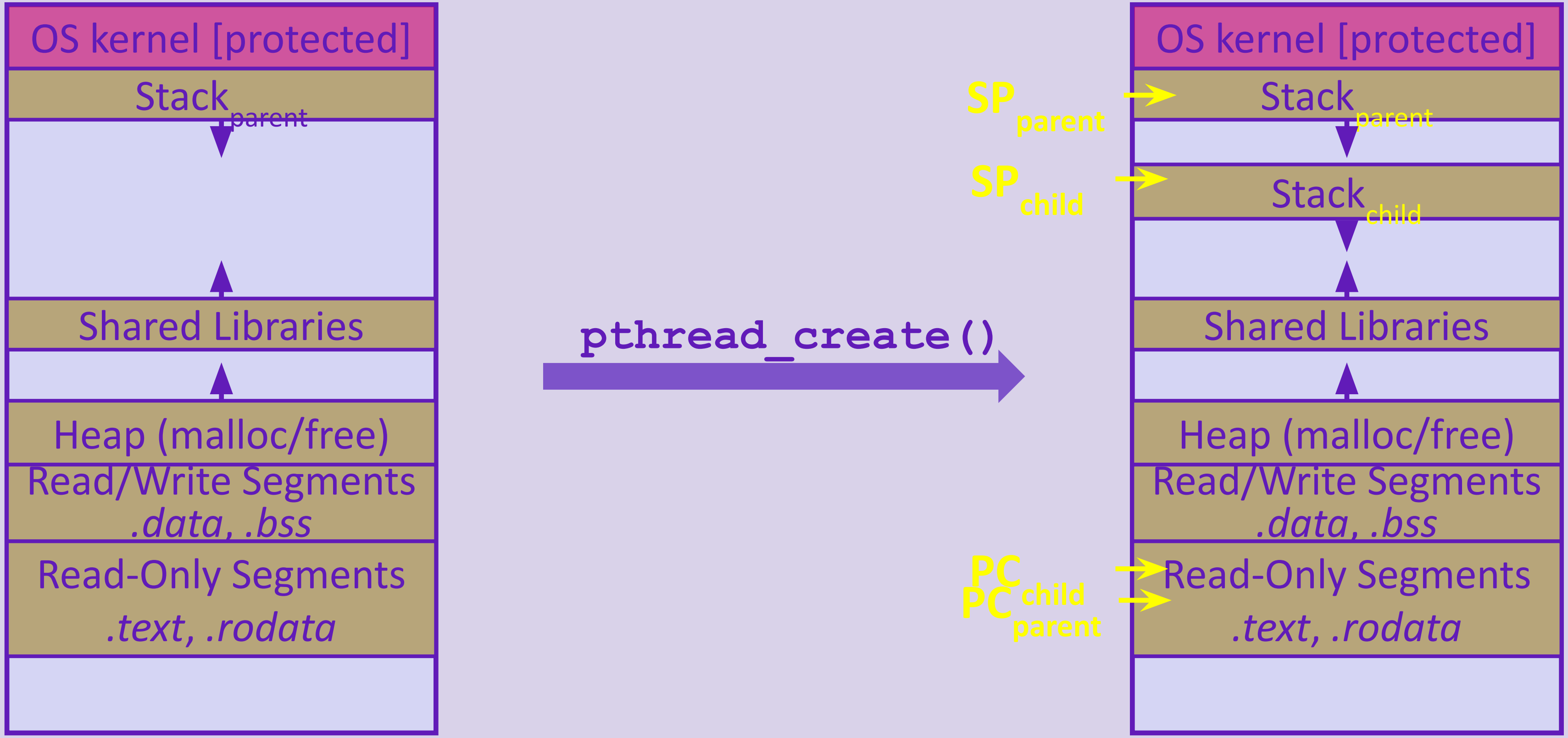
- Disadvantages:
  - More overhead than threads during creation and context switching
  - Cannot easily share memory between processes – typically communicate through the file system

Threads vs. Processes

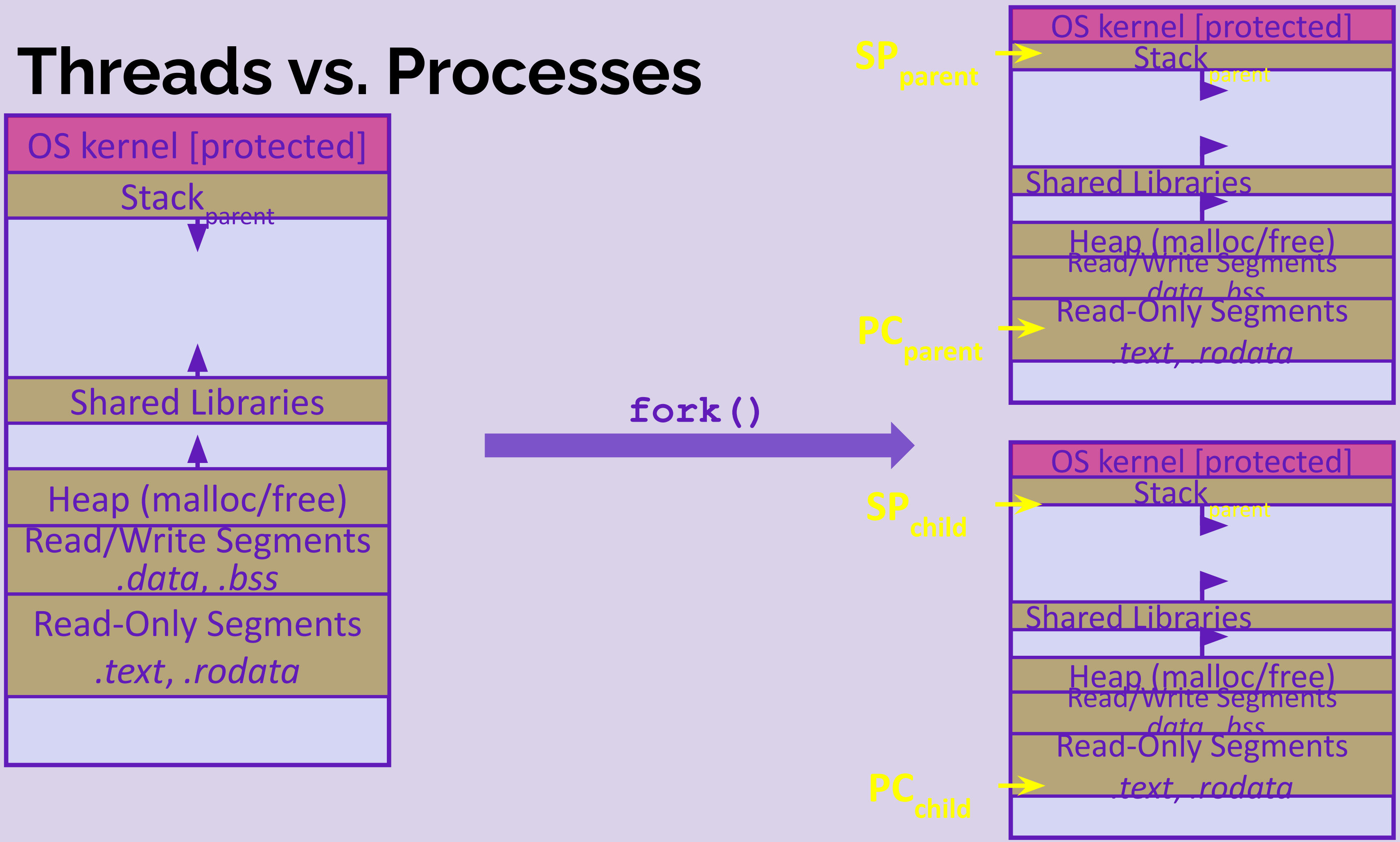


- Before creating a thread
  - One thread of execution running in the address space
    - One PC, stack, SP
  - That main thread invokes a function to create a new thread
    - Typically `pthread_create()`

Threads vs. Processes



Threads vs. Processes



Alternate: 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

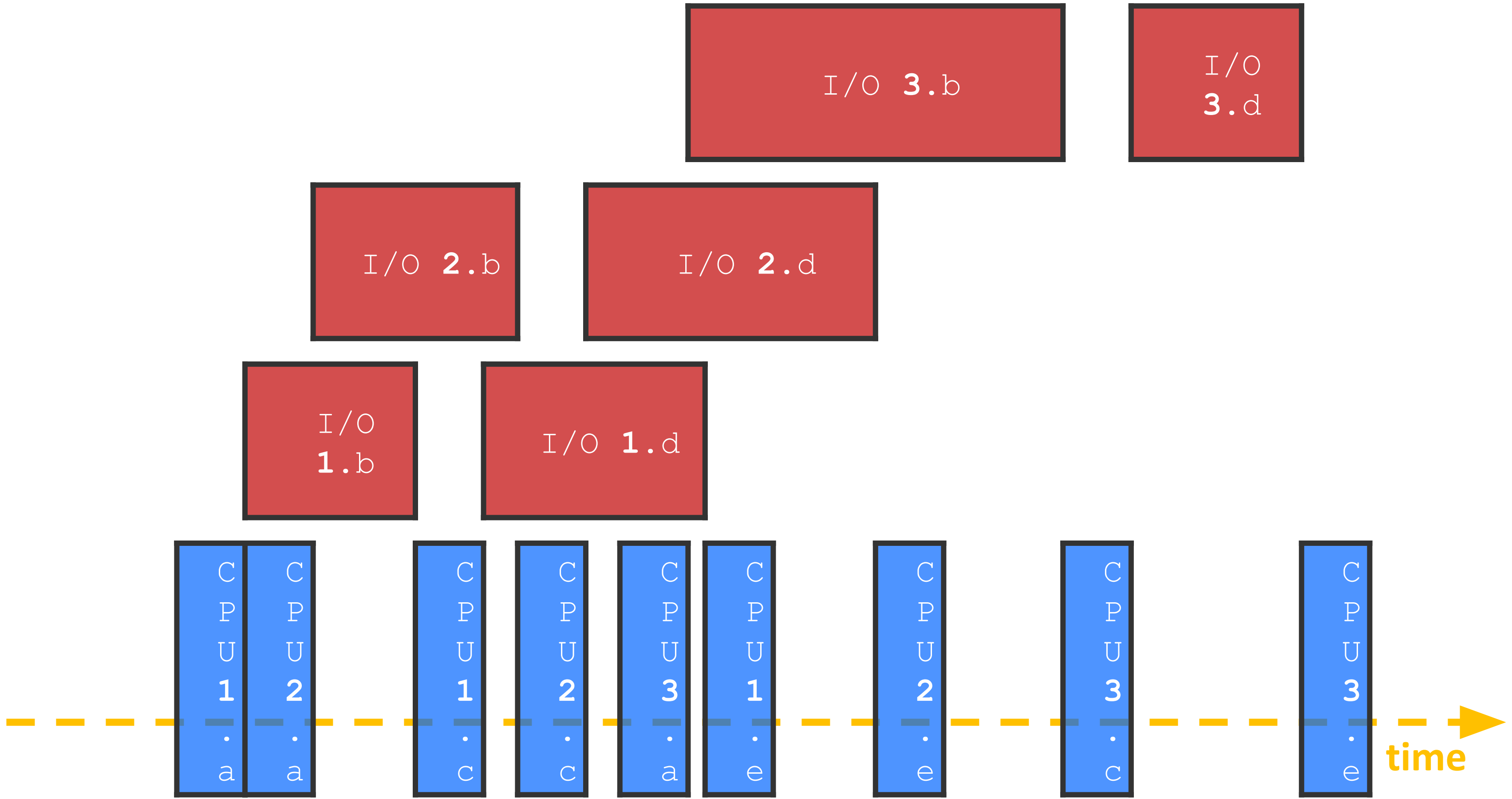
- Your program (almost never) blocks on I/O

Event-Driven Programming

- Your program is structured as an *event-loop*

```
void dispatch(task, event) {
  switch (task.state) {
    case READING_FROM_CONSOLE:
      query_words = event.data;
      async_read(index, query_words[0]);
      task.state = READING_FROM_INDEX;
      return;
    case READING_FROM_INDEX:
      ...
  }
}

while (1) {
  event = OS.GetNextEvent();
  task = lookup(event);
  dispatch(task, event);
}
```

Asynchronous, Event-Driven



Why Events?

- Advantages:
    - Don’t have to worry about locks and race conditions
    - For some kinds of programs, especially GUIs, leads to a very simple and intuitive program structure
        - One event handler for each UI event
- Disadvantages:
    - Can lead to very complex structure for programs that do lots of disk and network I/O
        - Sequential code gets broken up into a jumble of small event handlers
        - You have to package up all task state between handlers

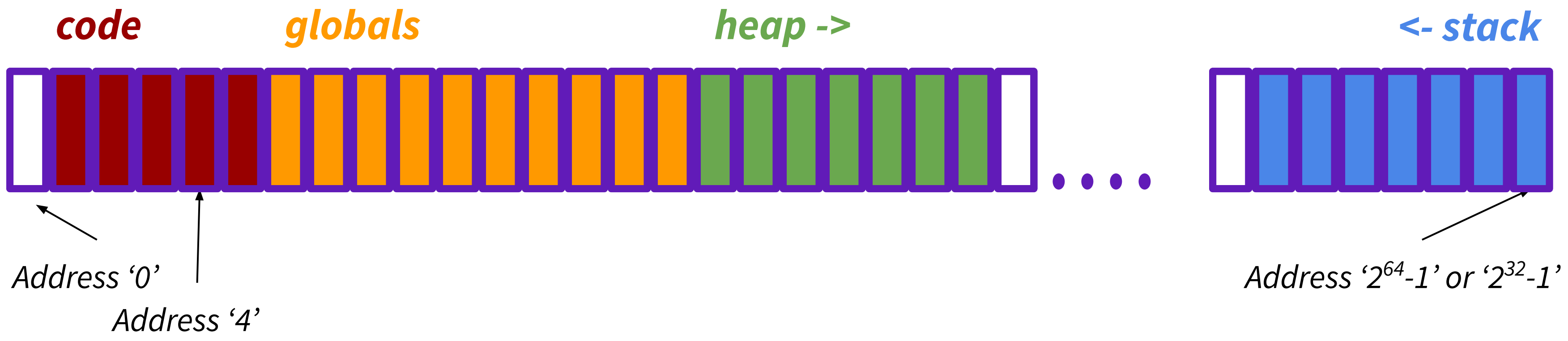
One Way to Think About It

- Threaded code:
  - Each thread executes its task sequentially, and per-task state is naturally stored in the thread's stack
  - OS and thread scheduler switch between threads for you

- Event-driven code:
  - *You* are the scheduler
  - You have to bundle up task state into continuations (data structures describing what-to-do-next); tasks do not have their own stacks

So, how do we do this?

- C, Java support parallelism 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”



Concurrency in C & Java

C: the POSIX Threads (pthreads) library

- #include <pthread.h>
- pass -lpthread to gcc (when linking)
- pthread_create takes a function pointer and arguments, runs as a separate thread

Java: built into the language

- Subclass java.lang.Thread, and override the run method
- Create a Thread object and call its start method
- Any object can “be synchronized on” (later today)

(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

https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html

Handling Memory

Threads

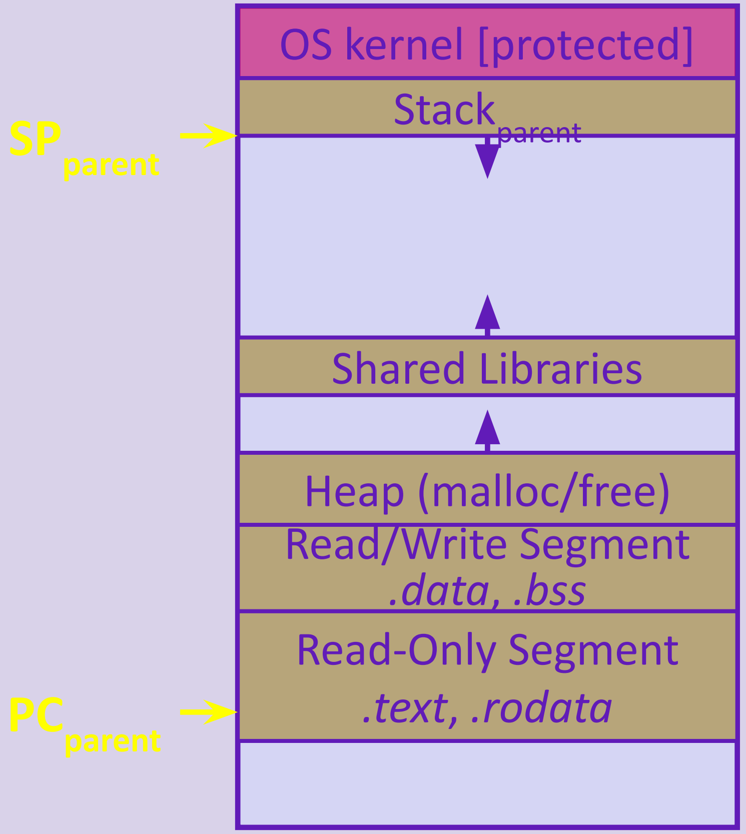
- Threads are like lightweight processes
  - They execute concurrently like processes
    - Multiple threads can run simultaneously on multiple CPUs/cores
  - Unlike processes, threads cohabitate the same address space
    - Threads within a process see the same heap and globals and can communicate with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

Memory Consideration

(ex. pthreadex.c)

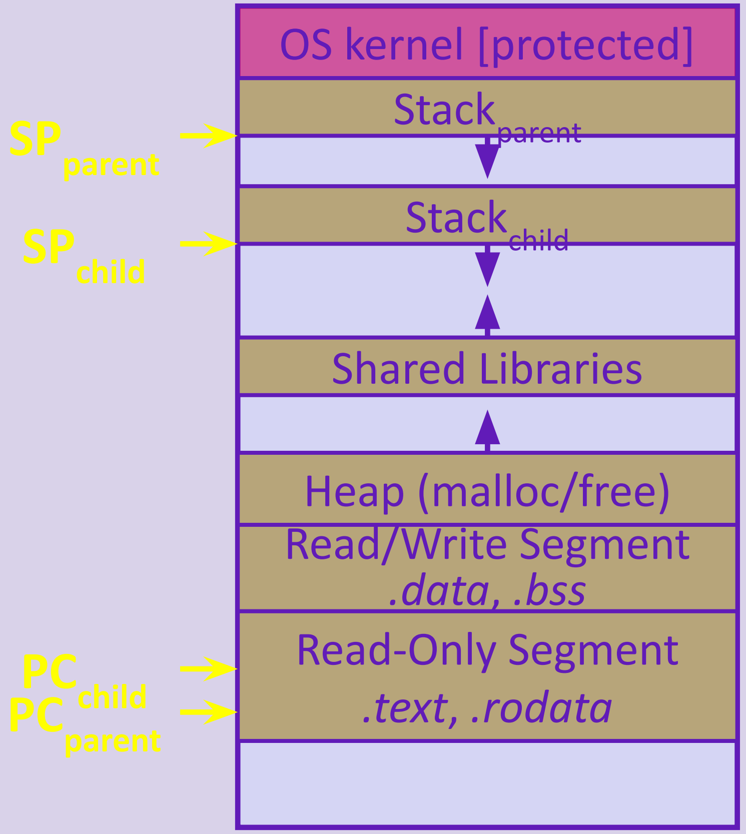
- If one thread did nothing of interest to any other thread, why bother running?
- Threads must communicate and coordinate
    - Use results from other threads, and coordinate access to shared resources
- Simplest ways to not mess each other up:
    - Don’t access same memory (complete isolation)
  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

Threads and Address Spaces



- Before creating a thread
  - One thread of execution running in the address space
    - One PC, stack, SP
  - That main thread invokes a function to create a new thread
    - Typically `pthread_create()`

Threads and Address Spaces



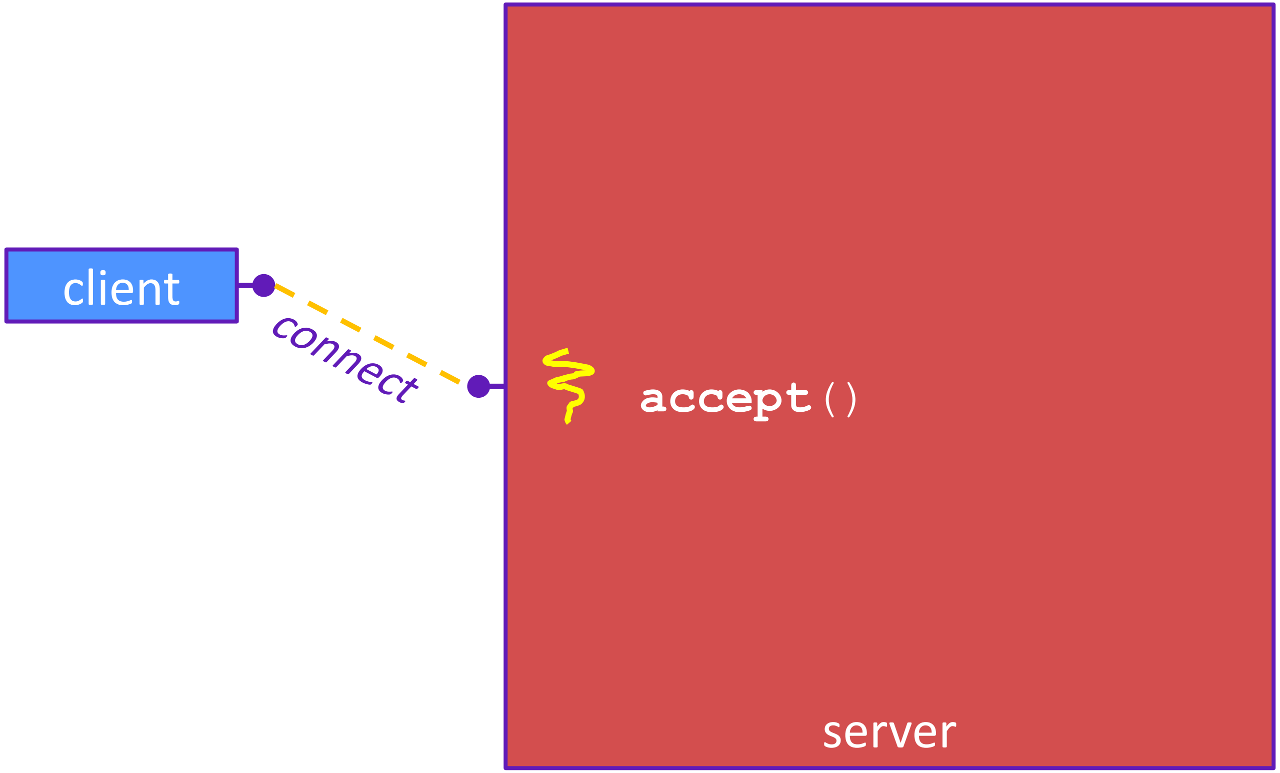
- After creating a thread
  - *Two* threads of execution running in the address space
    - Original thread (parent) and new thread (child)
    - New stack created for child thread
    - Child thread has its own PC, SP
  - Both threads share the other segments (code, heap, globals)
    - They can cooperatively modify shared data

Multithreaded Server: Architecture

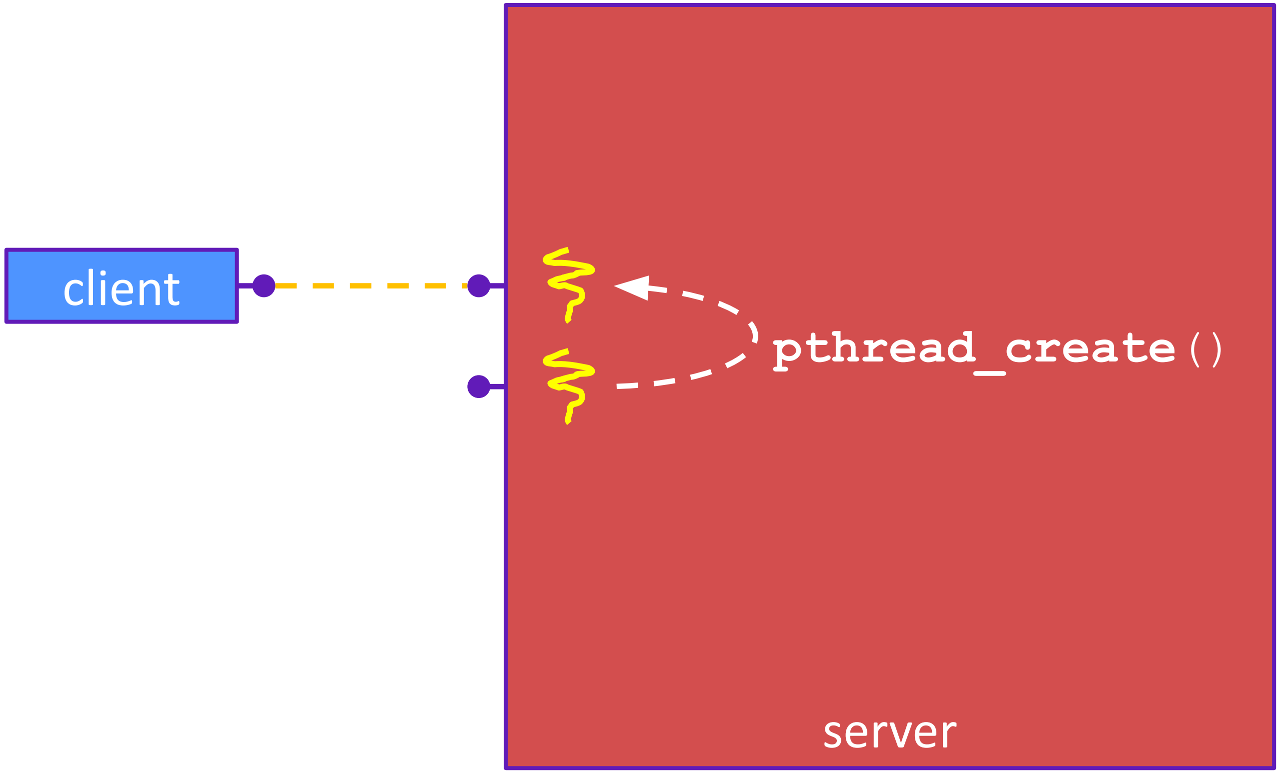
- A parent *thread* creates a new thread to handle each incoming connection
    - The child thread handles the new connection and subsequent I/O, then exits when the connection terminates

- See `searchserver_threads/` for code if curious

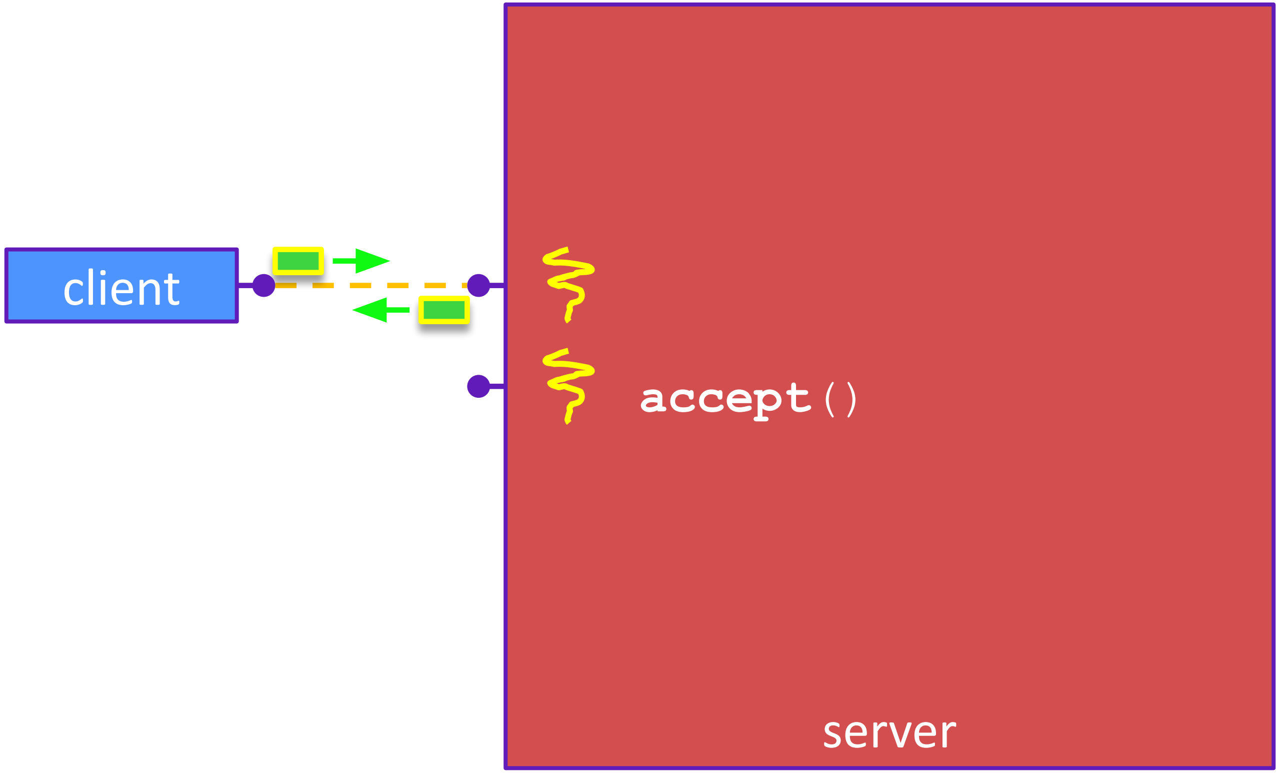
Multithreaded Server



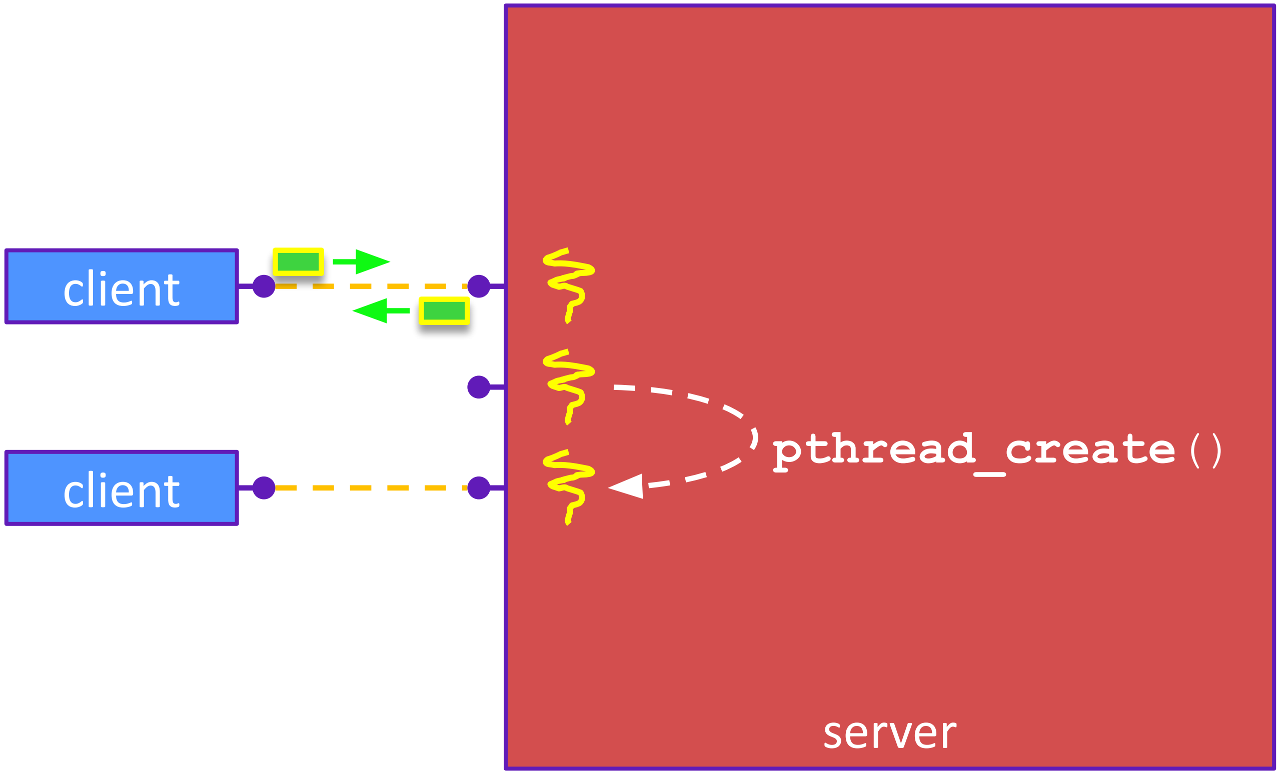
Multithreaded Server



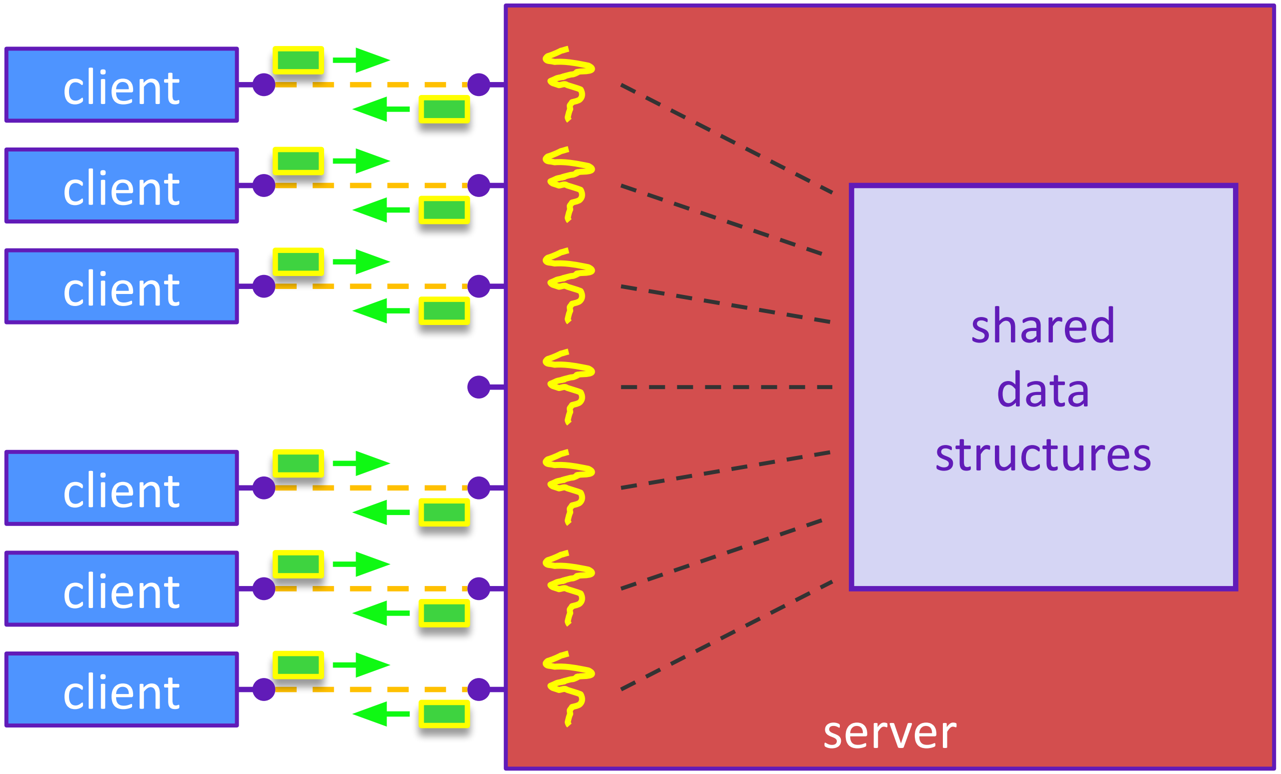
Multithreaded Server



Multithreaded Server



Multithreaded Server



POSIX Threads (pthreads)

- The POSIX APIs for dealing with threads
- Declared in `pthread.h`
  - Not part of the C/C++ language (cf. Java)
- To enable support for multithreading, must include `-pthread` flag when compiling and linking with `gcc` command

pthreads Threads: Creation

-
  ```
  int pthread_create(
          pthread_t* thread,
          const pthread_attr_t* attr,
          void* (*start_routine)(void*),
          void* arg);
  ```
  - Creates a new thread into `*thread`, with attributes `*attr`
  - Returns a status code (**0** or an error number)
  - The new thread runs **`start_routine`**`(arg)`

-
  ```
  void pthread_exit(void* retval);
  ```
  - Equivalent of **`exit`**`(retval)` for a thread instead of a process
  - thread automatically exits when it returns from **`start_routine`**`()`

pthreads Threads: Afterwards

-
  ```
  int pthread_join(pthread_t thread,
                   void** retval);
  ```
    - Waits for thread to terminate (equivalent to waitpid, but for threads)
    - Exit status of the terminated thread is placed in `**retval`

-
  ```
  int pthread_detach(pthread_t thread);
  ```
    - Mark thread as detached; will clean up its resources as soon as it terminates
- See `thread_example.cc`

Wherefore 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:
    - Shared fate within a process
        - One “rogue” thread can hurt you badly
    - Synchronization is complicated

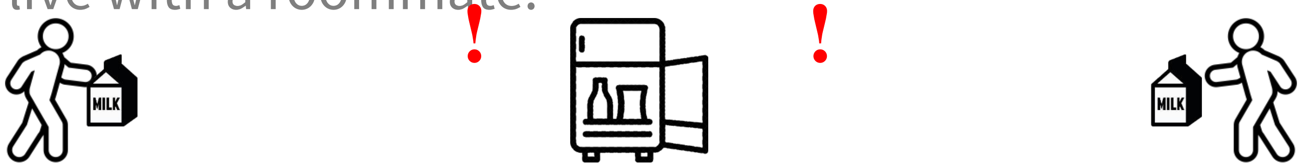
Data Race Example

- If your fridge has no milk,
  then go out and buy some more

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

- What could go wrong?
- If you live alone:

- If you live with a roommate:



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 milk

D. We’re lost…

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

Race Walkthrough

Alice	Bob
! note ! milk	
	! notes ! milk Leave note Buy milk Remove note
Leave note Buy milk Remove note	

```
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!

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

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

❖ Pseudocode:

```
// non-critical code

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

// non-critical code
```

loop/idle if locked

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?

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

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)

```
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

```
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

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

Example: Bank Accounts

Data races

```
struct Acct {int balance; /*etc…*/ };
int withdraw(struct Acct* a, int amt) {
    if (a->balance < amt) return FAIL;
    a->balance -= amt; return SUCCESS;
}
```

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

A Data Race - *two threads withdraw $100 simultaneously*

Thread 1

```
struct Acct {int balance; /*etc…*/ };
int withdraw(struct Acct* a, int amt) {
     if (a->balance < amt) {
          return FAIL; }







     a->balance -= amt; return SUCCESS;
}
```

Thread 2

```



struct Acct {int balance; /*etc…*/ };
int withdraw(struct Acct* a, int amt) {
     if (a->balance < amt) {
          return FAIL; }
     a->balance -= amt; return SUCCESS;
}
```

Atomic Operations

- 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
- 'Atomic' requires a hardware primitive

We can wrap the hardware primitive with a lock

In C : 'mutex'

```
std::mutex BankAccount::m_;
void BankAccount::withdraw(double amount) {
    m_.lock();
    if (getBalance() > b) {
        throw std::invalid_argument();
    }
    setBalance(getBalance() - amount);
    m_.unlock();
}
```

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_`’

```
std::mutex BankAccount::m_;
void BankAccount::withdraw(double amount)
{
     m_.lock();
     if (getBalance() > b) {
          throw std::invalid_argument();
     }
     setBalance(getBalance() - amount);
     m_.unlock();
}
```

Deadlocking

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

If `foo` locks `D`, and then calls `bar` which also must lock `D`, we get a deadlock - we can't progress because `bar` can not complete

One solution is to write a helper function to replace `bar` - `lockedBar`

```
void BankAccount::withdraw(double amount) {
  m_.lock();
  if (getBalance() < amount) {
    throw std::invalid_argument();
  }
  setBalanceUnderLock(getBalance() - amount);
  m_.unlock();
}

void setBalance(double amount) {
  m_.lock();
  setBalanceUnderLock(amount);
  m_.unlock();
}

void setBalanceUnderLock(double amount) {
  balance_ = amount;
}
```

Deadlock

Problem:

If every method that modifies balance_ is locked with mutex m, that balance can not be updated.

Solution:

Must create helper function that allows for modifying balance_ under the lock.

```
void BankAccount::withdraw(double amount) {
  m_.lock();
  if (getBalance() < amount) {
    throw std::invalid_argument();
  }
  setBalanceUnderLock(getBalance() - amount);
  m_.unlock();
}

void setBalance(double amount) {
  m_.lock();
  setBalanceUnderLock(amount);
  m_.unlock();
}

void setBalanceUnderLock(double amount) {
  balance_ = amount;
}
```

C++ Lock Guards

- A “lock guard” is an object that
  - Locks the mutex in the constructor
  - Unlocks in the destructor
- 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.

```
void deposit(double amount) {

   std::lock_guard<std::mutex> lock(m_);
  // locks mutex m_ in the lock_guard constructor
  // mutex is now locked

   setBalanceWithLock(getBalance() + amount);
    // When deposit() returns,
    // the stack-allocated lock_guard will be deleted,

   // calling the destructor and releasing the mutex.

 }
```

Another Deadlock

If we have an operation for two accounts

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 transferTo(double amount, BankAccount& other) {
    m_.lock();
    other.m_.lock();

    setBalanceInternal(getBalance() - amount);
    other.setBalanceInternal(other.getBalance() + amount);

    other.m_.unlock();
    m_.unlock();
  }
```

Thread T1: A.transferTo(50, B);	Thread T2: B.transferTo(20, A);
m_.lock(); // Locks A's mutex	
	m_.lock(); // Lock's B's mutex
other.m_.lock(); // Waits for B's mutex	
	other.m_.lock() // Waits for A's mutex

Another Solution

- **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.
    a. 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.
    a. 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.

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.

```
// In h:
static std::atomic<int> accountCount_;

// In cpp:
BankAccount::BankAccount() {
     accountId_ = ++accountCount_;
    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.

Memory Guidelines

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.

Synchronicity

1. **No data races.** Never allow two threads to read/write or write/write a location at the same time.
2. **Think of what operations need to be atomic.** Consider atomicity first, then figure out how to implement it with locks).
3. **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.
4. **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.
5. **Don't do expensive computations or I/O in critical sections, but also don't introduce race conditions.** This balances performance with correctness.
6. **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.