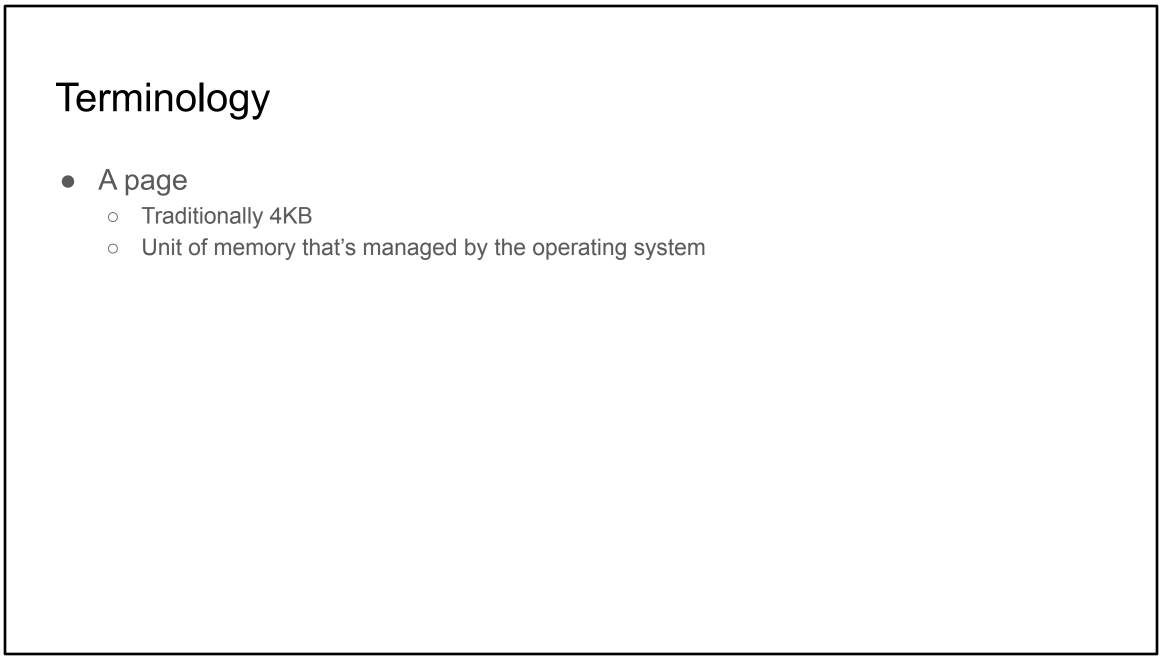
COS 316 Precept: Lecture Review + Socket Programming

Outline

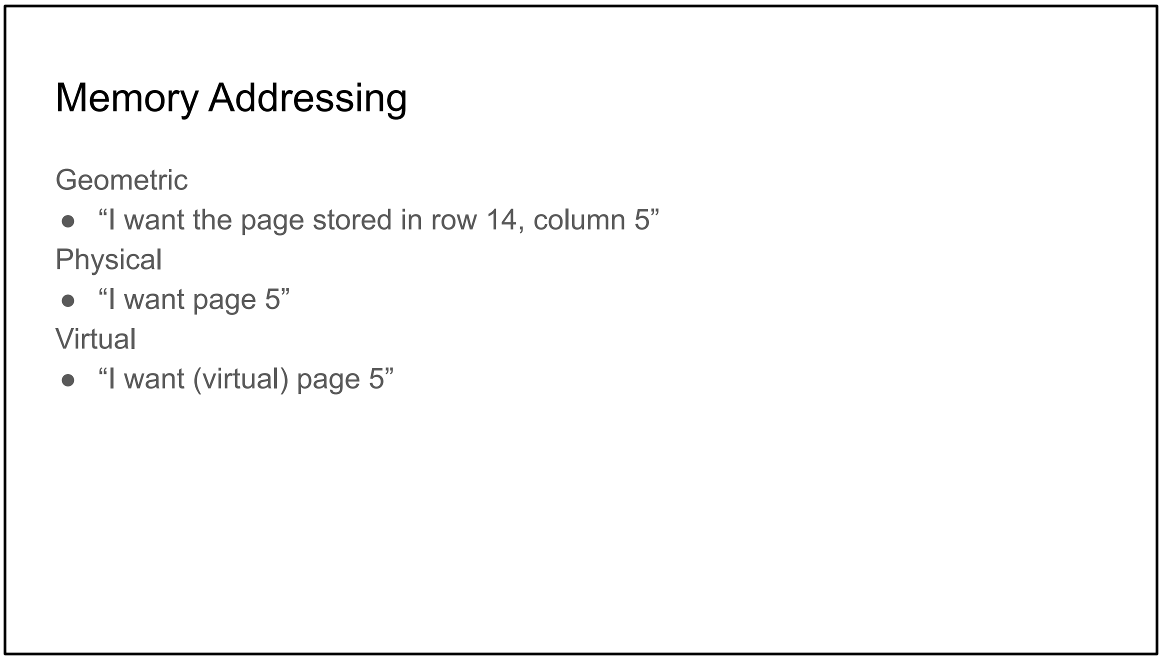
- Memory Addressing
- Socket programming



Terminology

- A page
  - Traditionally 4KB
  - Unit of memory that’s managed by the operating system

- A page is the normal unit of memory that’s managed by an operating system
- They’re traditionally 4 kilobytes in size
- While DRAM is able to have much smaller units read from and written to it, for practical reasons, the operating system will manage things in units of 4KB
- For the sake of simplicity, let’s think of a page as a single unit of memory



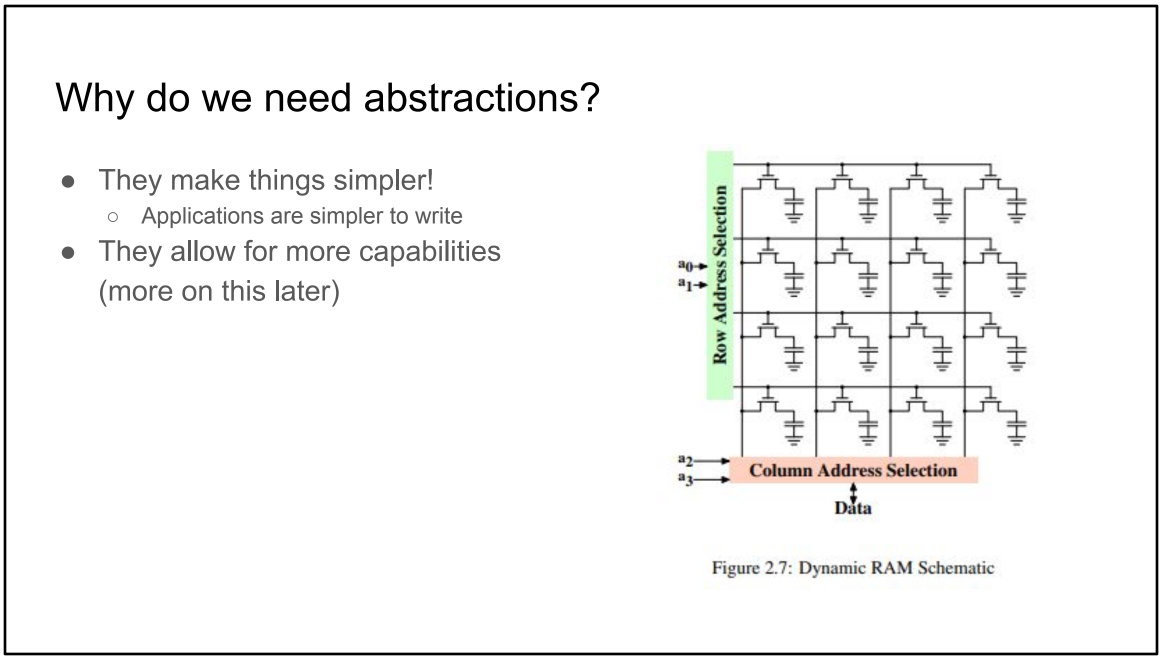
Memory Addressing

Geometric
- “I want the page stored in row 14, column 5”

Physical
- “I want page 5”

Virtual
- “I want (virtual) page 5”

- So, there are a few ways to address memory
- In Tuesday’s lecture, Wyatt covered geometric, physical, and virtual addressing
- For geometric addressing, reading from a page may look like the following: “I want the page stored in row 14, column 5”
- For physical addressing, reading from a page is simpler: “I want page 5”
- And finally, virtual addressing *looks* like the following to applications: “I want page 5”
- Importantly, each of the addressing methods builds an abstraction on top of the previous one
- That is, physical builds an abstraction on top of geometric, and virtual builds an abstraction on top of physical
- Note that while physical and virtual addressing LOOK identical, there’s a large and important difference that will be explained later



Why do we need abstractions?

- They make things simpler!
  - Applications are simpler to write
- They allow for more capabilities (more on this later)

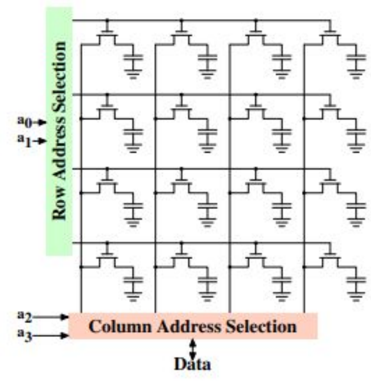
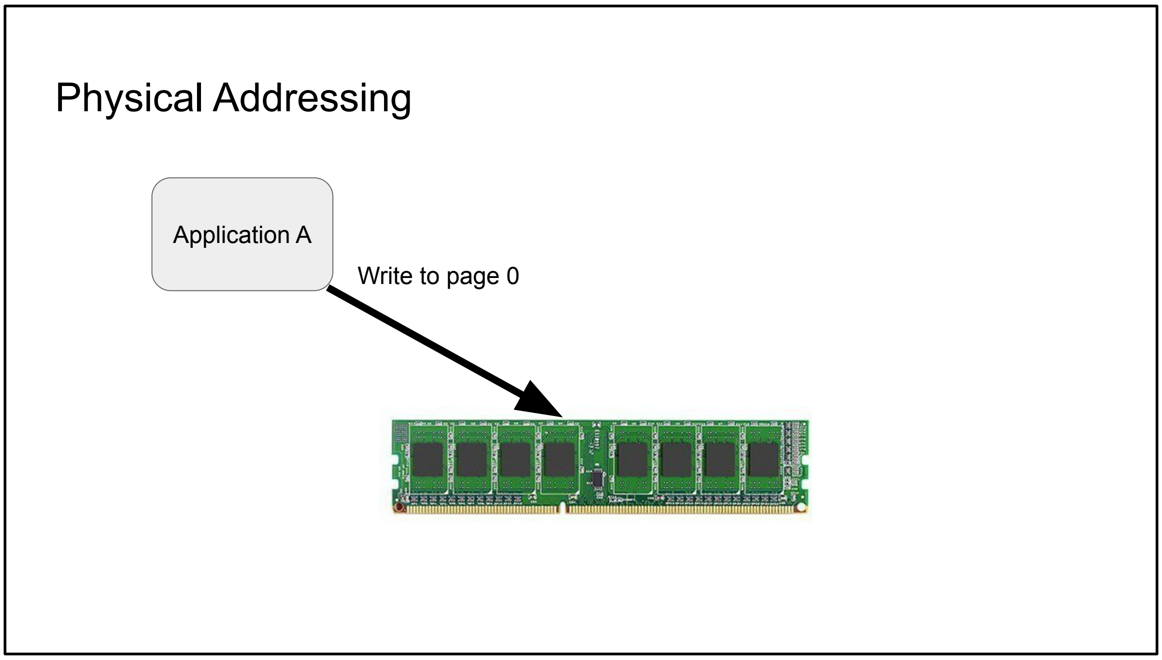


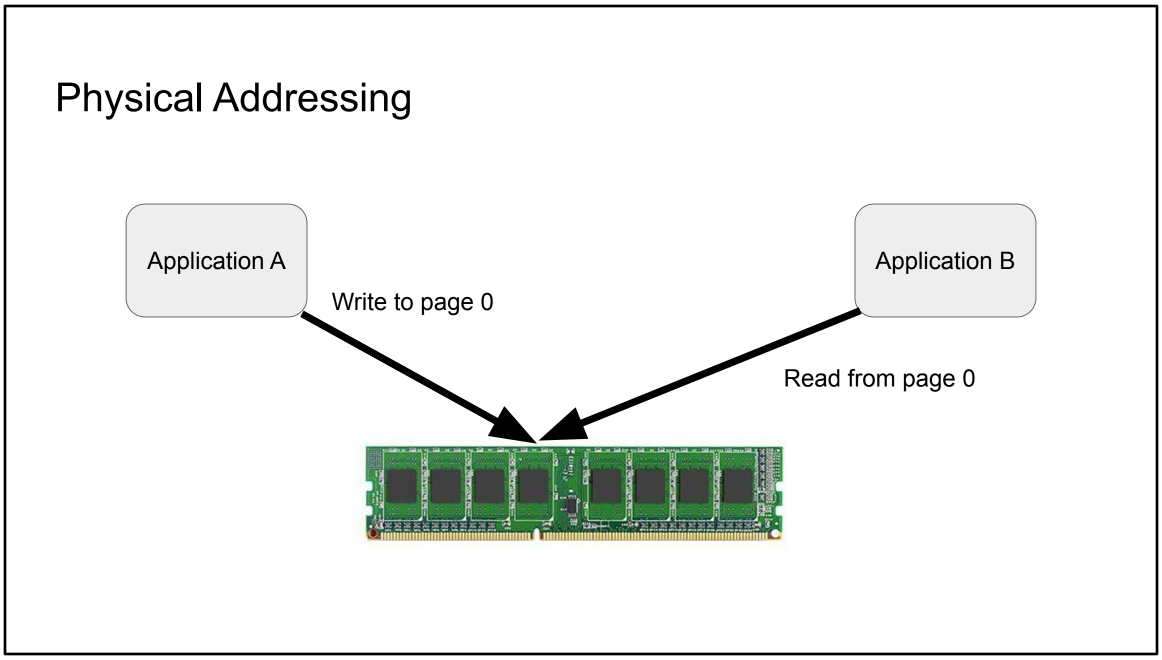
Figure 2.7: Dynamic RAM Schematic

- So why do we need these abstractions?
- The first reason is that they simplify things!
- The second reason is that they allow for more capabilities, which we’ll discuss later
- For now, let’s focus on the first reason
- Let’s look at geometric addressing. An application that wants to access a memory location would need to know the geometry of the memory itself.
- For instance, an access may look something like the following:
- “There are 128 rows and 256 columns on this DIMM, which means that the page I want to access is on row 5 and column 50”
- This is complex and also tied to the geometry of a particular DIMM
- If I switch out one DIMM for another, the number of rows and columns may change



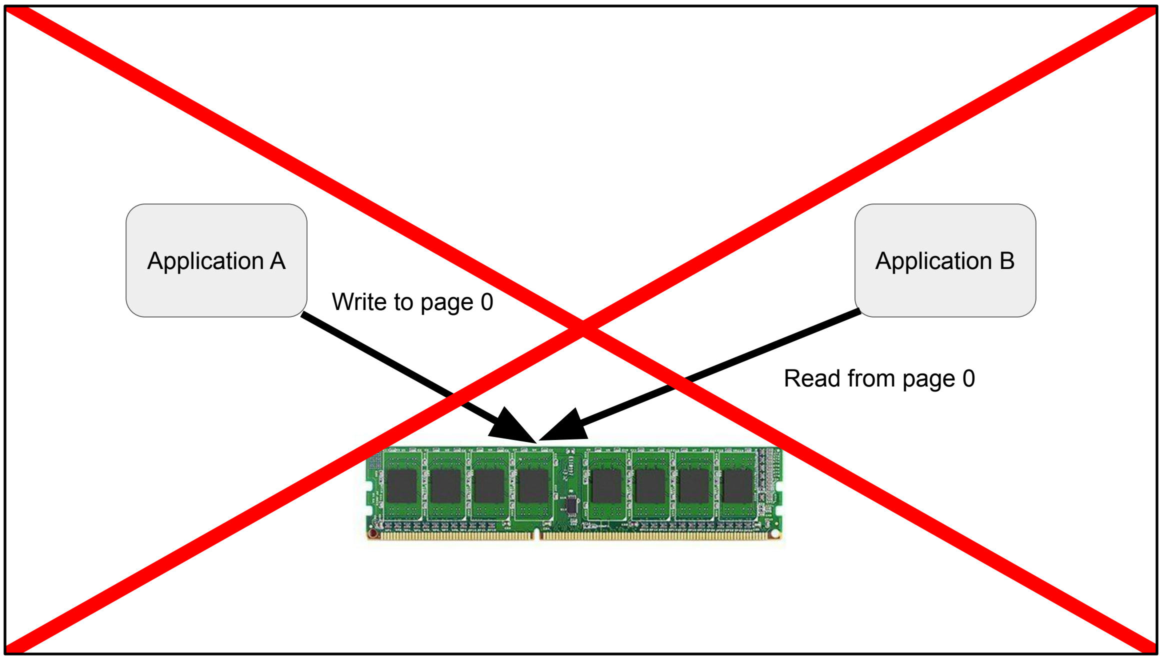
Physical Addressing

- So we introduce physical addressing
- Instead of specifying some memory location based on the DIMM’s geometry, applications can use a *linear address* to refer to a memory location
- Hardware attached to the memory will then translate that linear address into a geometric address
- That’s much simpler and doesn’t require the application to know anything about the DIMM’s geometry
- This is great so far!

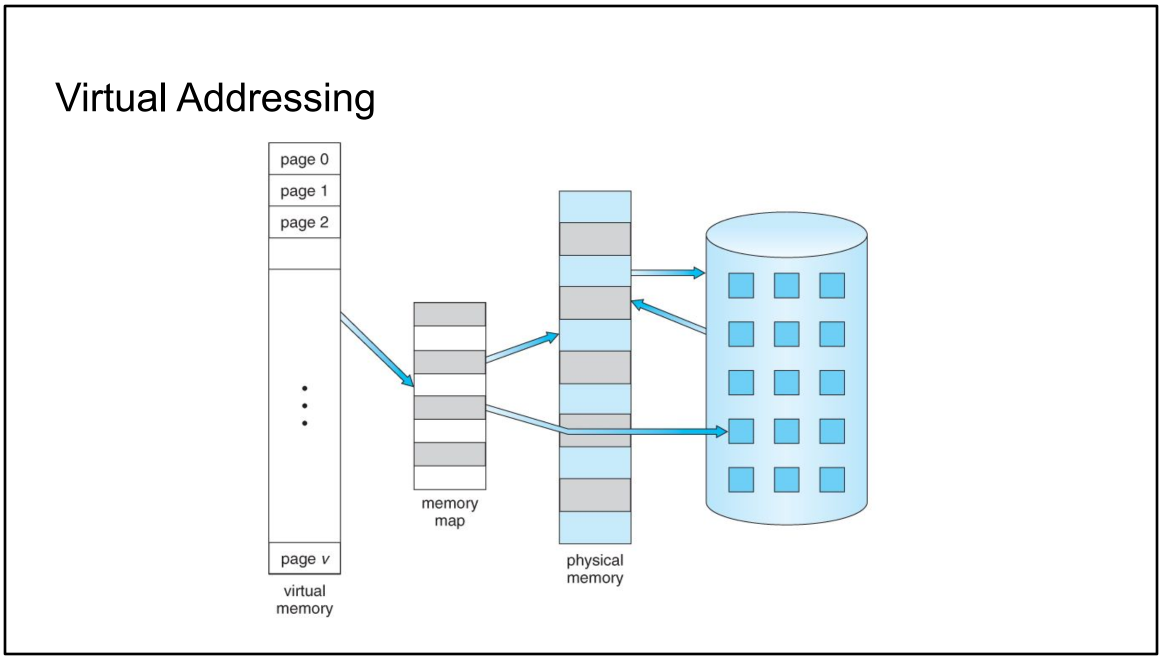


Physical Addressing

- But what if you want to run TWO applications?
- We generally don’t want applications to be able to access each other’s data
- You don’t want some sketchy game that you pirated to have access to the memory that your banking app is using



- So physical memory addressing can be a problem
- If application A wants to write to page 0, and application B wants to read from page 0, then they'll be referring to the SAME location!
- This can result in all kinds of weirdness
- In addition, it can make writing applications more difficult
- Things become much more complex when an application not only has to keep track of its own memory usage, but also the memory usage of the other applications on the system!
- If applications are going to share a single resource, aka multiplex, there needs to be some way for that sharing to be made easy

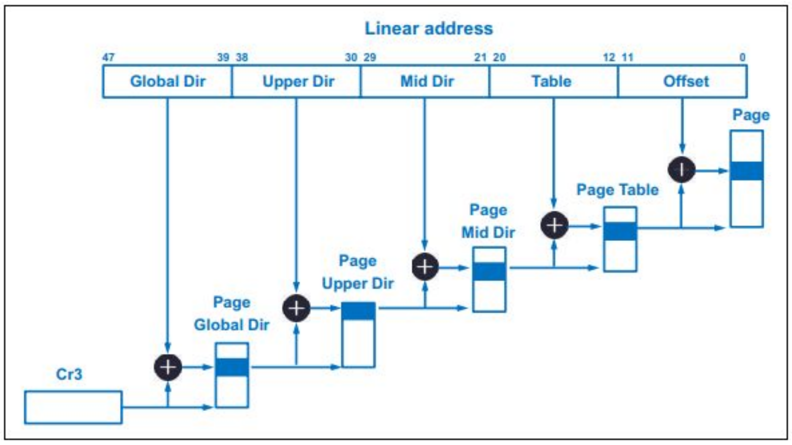


Virtual Addressing

- So we introduce virtual addressing
- Virtual addressing allows multiple applications to use physical memory without interfering with each other
- This scheme accomplishes this by adding an additional level of abstraction
- Now, instead of each application having direct access to physical memory, the operating system keeps a mapping of virtual pages to physical pages for each application!
- Importantly, it will not map virtual pages for two applications to the same physical page!
- When an application attempts to access a memory location, the system will step in and perform the translation from virtual page to physical page on the fly
- The application never knows a thing!
- This provides several benefits:
- Since the system sits between the application and the memory it’s accessing, it can do things like make sure application A doesn’t access application B’s memory
- In addition, it can provide a simpler and easier to work with view of a

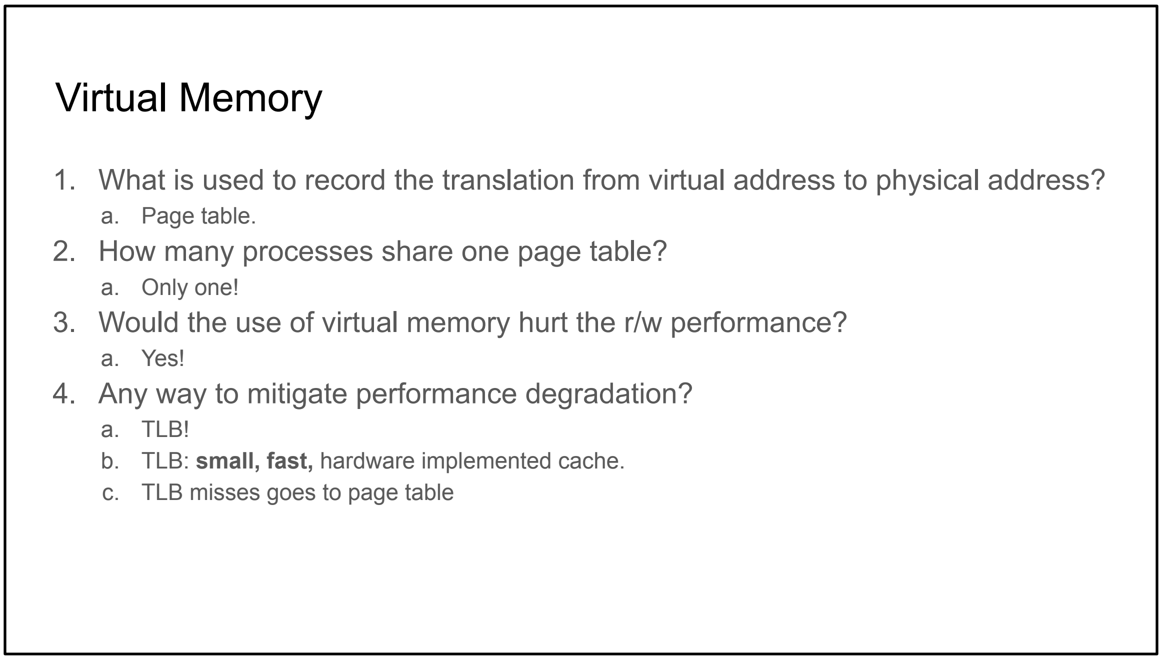
- machine’s memory to an application
- A system that provides virtual addressing essentially lies to each application and tells it that it has a large section of contiguous memory all to itself
- It then works to maintain this lie by keeping a mapping of virtual pages to physical pages and performing the translation between them on the fly whenever an application accesses memory
- So even if there are a ton of running applications using physical memory, your application won’t have to worry about it

Virtual Addressing



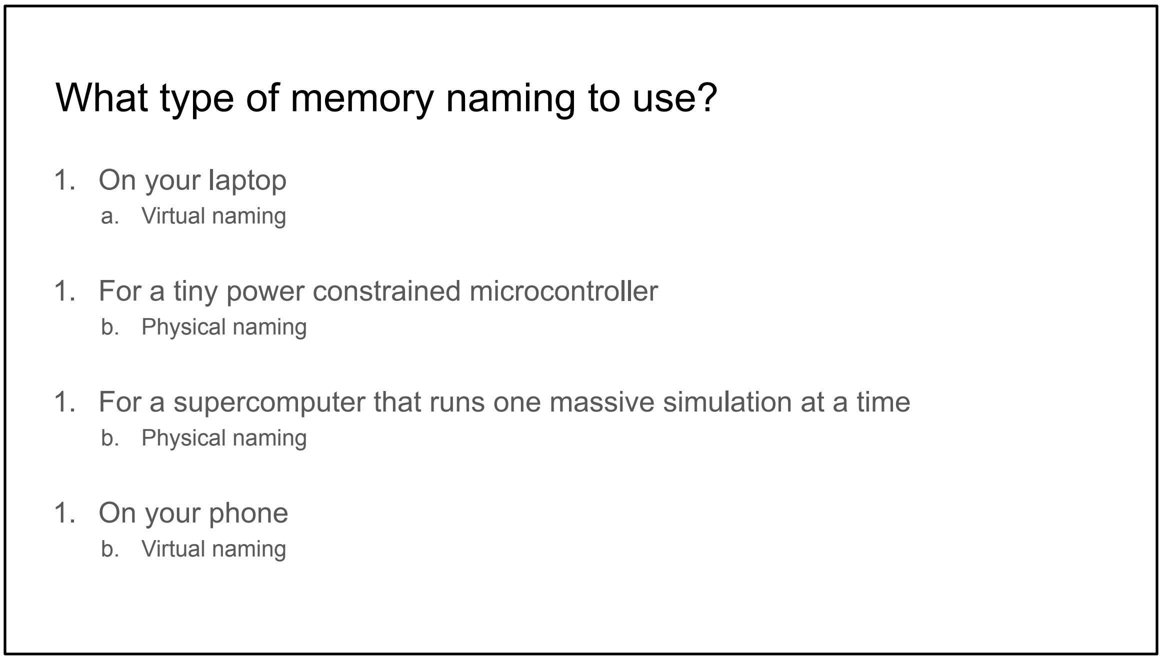
- The data structure used to hold these virtual to physical address translations is aptly called a page table
- When an application is running, its page table is loaded into memory and used by the operating system to perform virtual to physical page translations
- This leads us to a bit of a problem
- While page tables provide a convenient way to write simpler and safer applications, they affect performance
- Memory accesses take time, they are not instantaneous
- Formerly, when an application wanted to access memory, it would have to wait for a single memory access to complete
- But now, the operating system needs to perform the virtual to physical page translation whenever an application wants to access memory.
- For a small page table, this doubles the time spent waiting for the access to complete
- For a large page table, the operating system may need to perform multiple accesses to get the full translation
- This can drastically raise the cost of an application’s memory accesses
- There are things that can be done to mitigate this, like using a TLB

- But in general, a price is paid for virtual memory’s benefits



Virtual Memory

1. What is used to record the translation from virtual address to physical address?
   a. Page table.
2. How many processes share one page table?
   a. Only one!
3. Would the use of virtual memory hurt the r/w performance?
   a. Yes!
4. Any way to mitigate performance degradation?
   a. TLB!
   b. TLB: **small, fast,** hardware implemented cache.
   c. TLB misses goes to page table



What type of memory naming to use?

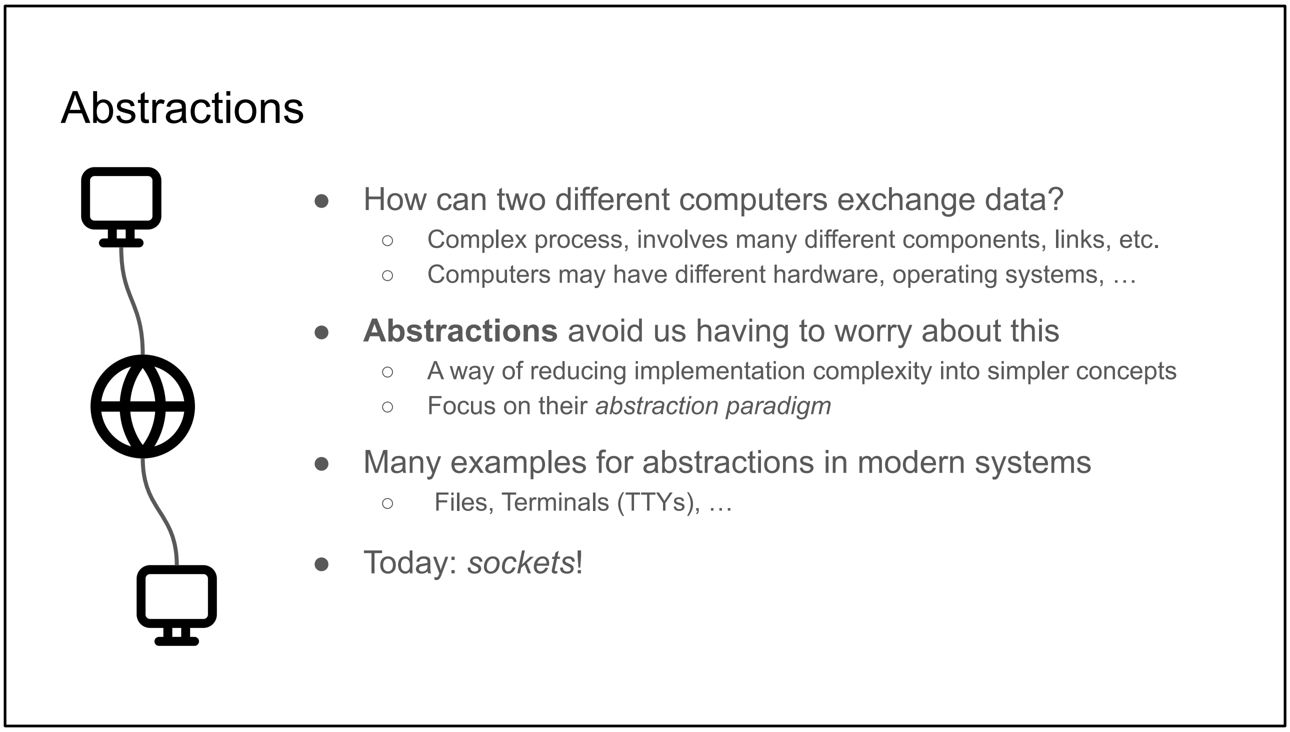
1. On your laptop
   a. Virtual naming

1. For a tiny power constrained microcontroller
   b. Physical naming

1. For a supercomputer that runs one massive simulation at a time
   b. Physical naming

1. On your phone
   b. Virtual naming

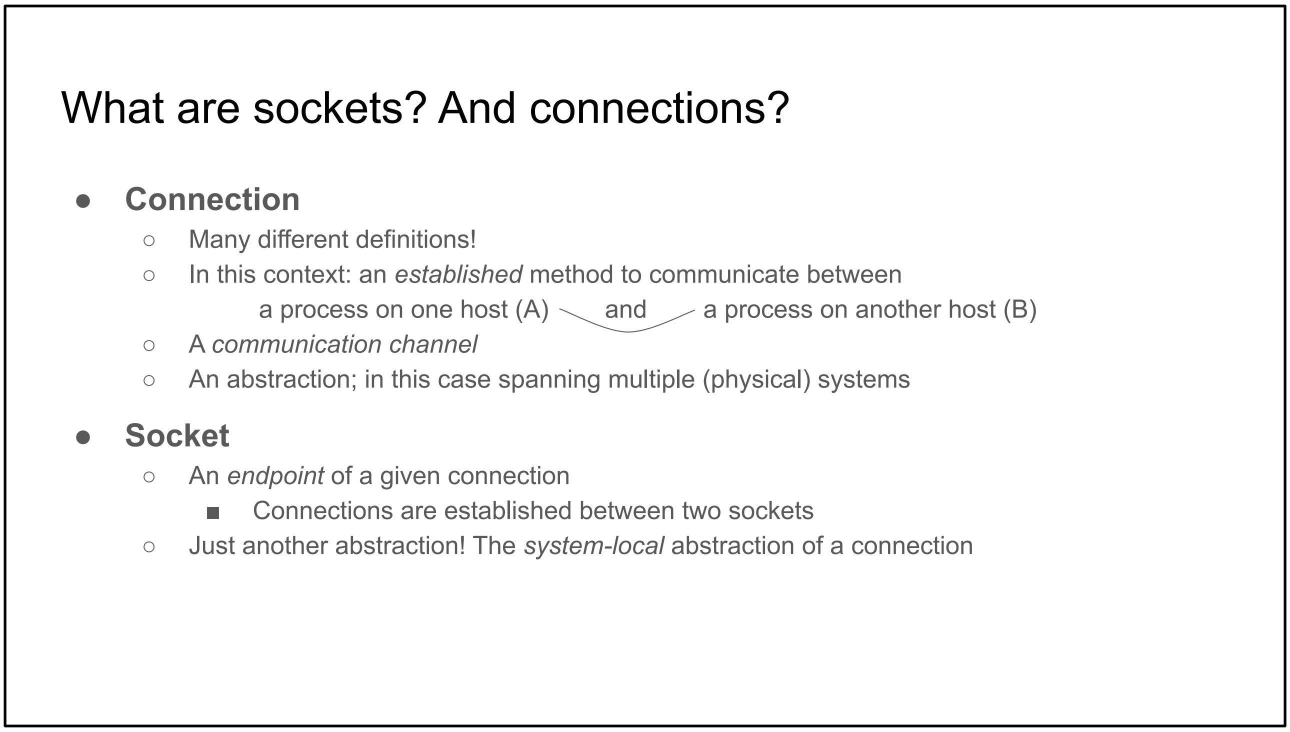
- Let’s take a look at these questions from lecture and try to answer it, given what we know about physical and virtual addressing
- For a laptop, what kind of of addressing should we use? Virtual or physical?
  - Virtual: Many applications need to run concurrently and share the physical memory on the machine
- Tiny power constrained microcontroller
  - Physical: Only a single application may need to run at once. Also, translating virtual addresses to physical addresses requires computation and energy, something a tiny microcontroller may not have enough of to spare
- A super computer that runs one massive simulation
  - Physical is more performant since it doesn’t need to waste CPU cycles on translating, and it’s running one application so there’s no multiplexing of the physical memory
- Phone:
  - Virtual: Same as laptop



Abstractions

- How can two different computers exchange data?
  - Complex process, involves many different components, links, etc.
  - Computers may have different hardware, operating systems, …
- **Abstractions** avoid us having to worry about this
  - A way of reducing implementation complexity into simpler concepts
  - Focus on their *abstraction paradigm*
- Many examples for abstractions in modern systems
  - Files, Terminals (TTYs), …
- Today: *sockets*!

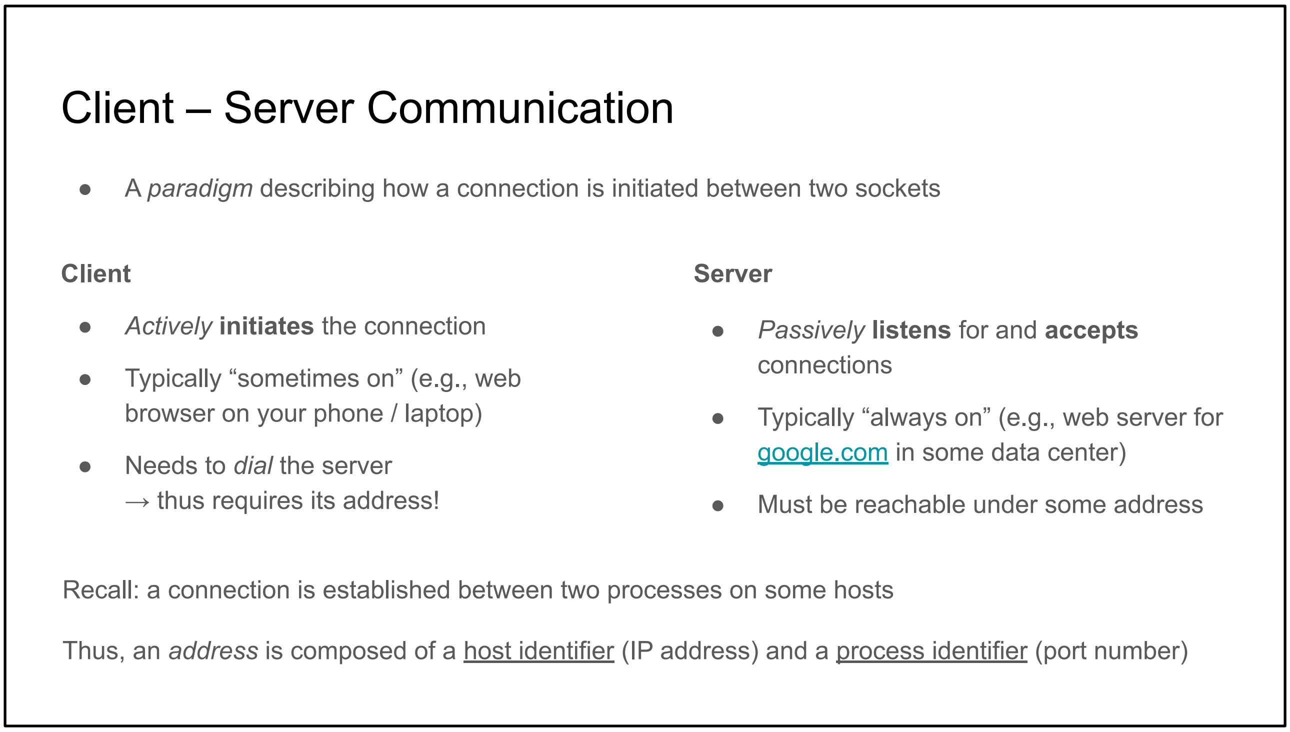
- First, let’s talk about abstractions
- Computers are composed of components with varying degrees of complexity.
- Abstractions provide a more friendly way to interact with that complexity
- For a computer-centric example, when you want to perform some calculations, say adding numbers together, you don’t have to think about which transistors need to be turned on and off; you just pop open your calculator and press some keys
- Similarly, when developers (in this case you) write applications that need to communicate over the network, they don’t want to think about how communication protocols work or how their application will share a single physical interface with other processes that are running on the machine. They want some abstraction that makes all of that easier so that they can focus on their application logic.
- Today, we’ll talk about the most commonly used networking abstraction, sockets!



What are sockets? And connections?

- **Connection**
  - Many different definitions!
  - In this context: an *established* method to communicate between
    a process on one host (A) and a process on another host (B)
  - A *communication channel*
  - An abstraction; in this case spanning multiple (physical) systems
- **Socket**
  - An *endpoint* of a given connection
    - Connections are established between two sockets
  - Just another abstraction! The *system-local* abstraction of a connection

- In this context, a *connection* refers to a communication between two processes.
- Those two processes can be on the same machine or on different machines that are connected via a network.
- A socket is the local abstraction that’s used to interact with that connection.
- So imagine a communication channel with a host/process on each end. Each host’s view of the channel is that of a socket.



Client – Server Communication

- A *paradigm* describing how a connection is initiated between two sockets

Client

- *Actively* **initiates** the connection
- Typically "sometimes on" (e.g., web browser on your phone / laptop)
- Needs to *dial* the server
  → thus requires its address!

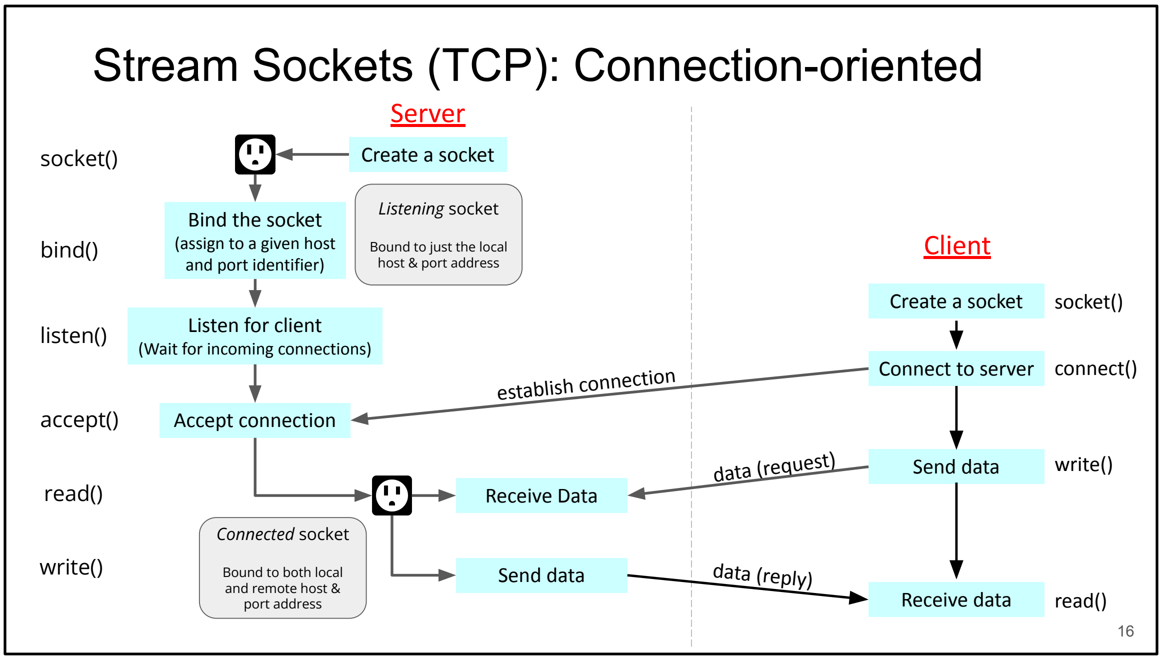
Server

- *Passively* **listens** for and **accepts** connections
- Typically "always on" (e.g., web server for google.com in some data center)
- Must be reachable under some address

Recall: a connection is established between two processes on some hosts

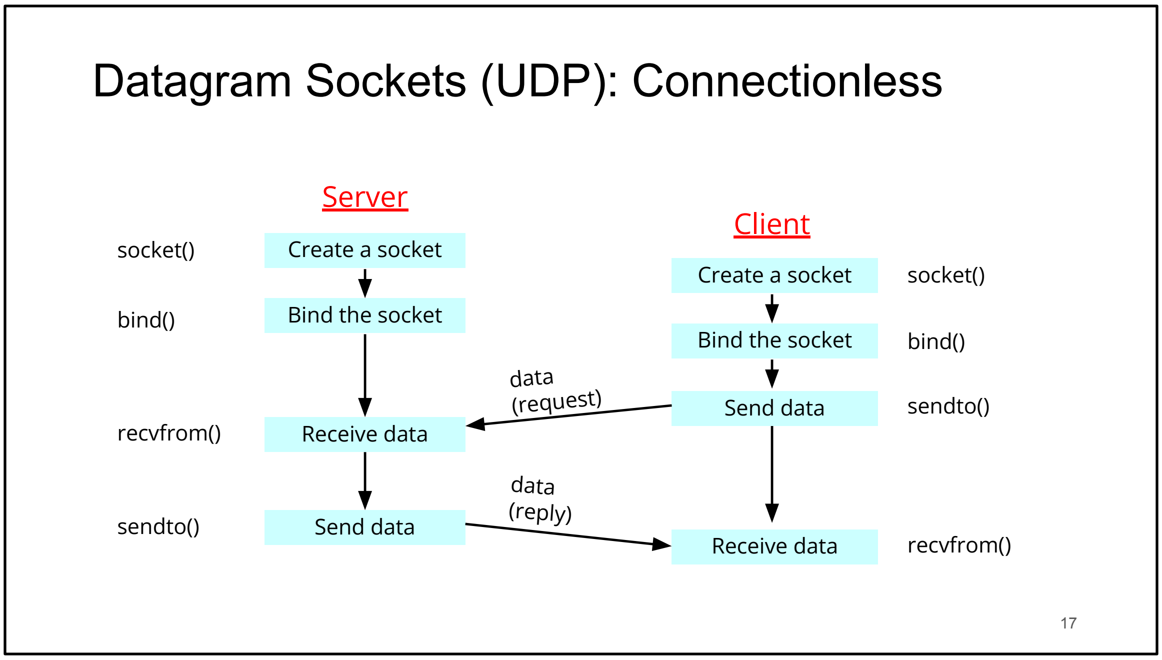
Thus, an *address* is composed of a host identifier (IP address) and a process identifier (port number)

- Next, let's discuss a common communication paradigm.
- Oftentimes, the relationship between two hosts is asymmetrical.
- The way that we normally interact with other machines follows the client-server paradigm
- In this paradigm, there is a server that's constantly listening for new attempts to connect.
- Conversely, there are clients that try to connect for temporary sessions.



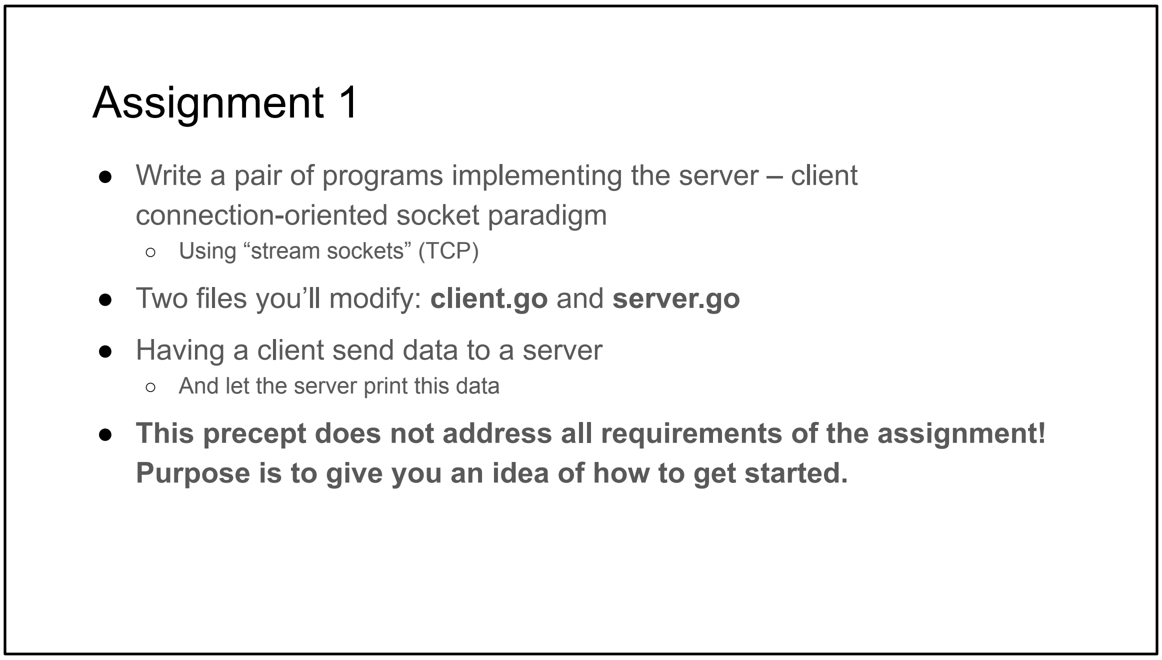
Stream Sockets (TCP): Connection-oriented

- There are two types sockets/connections that are generally used
- The first is a TCP or connection-oriented socket
- TCP stands for transmission control protocol, and it's used for when a client and server want their connection to *persist*
- This means that once established, the connection between the client and server will continue to exist until one of them terminates it
- And they can continually exchange data over this connection
- Implementations of the TCP protocol on the client and server work with each other to ensure that data arrives intact and in order
- Weird things can happen over a network, and sometimes things get rearranged or lost entirely
- The TCP protocol acts to mitigate these occurrences
- These types of connections are used when its important for all of the data to arrive intact and in order
- For example, streaming a movie or downloading a file



Datagram Sockets (UDP): Connectionless

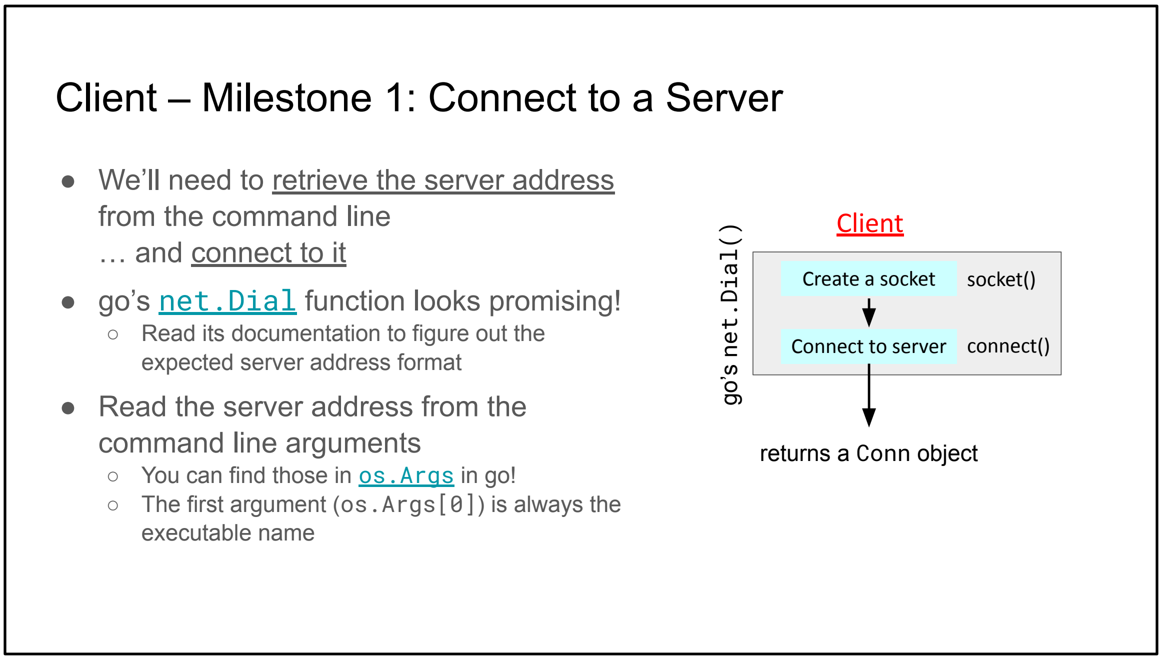
- The next type of socket is a UDP socket, or a connectionless socket
- UDP stands for User Datagram Protocol
- Unlike a TCP connection, this connection does not persist
- The client and server's implementations of these protocols don't keep track of order or arrival
- When sending the data, the sender essentially thinks: "I hope it gets to its destination, but I'm not going to worry about it"
- These types of connections are used when its not important that data arrives intact or in order
- For example: Video conferencing. "I really hope that last image of the user's face arrived. But if it doesn't, I'm not going to worry about it. I have new images to send!"



Assignment 1

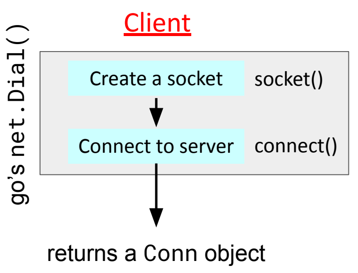
- Write a pair of programs implementing the server – client connection-oriented socket paradigm
  - Using “stream sockets” (TCP)
- Two files you’ll modify: **client.go** and **server.go**
- Having a client send data to a server
  - And let the server print this data
- **This precept does not address all requirements of the assignment! Purpose is to give you an idea of how to get started.**

- For assignment 1, you’ll implement a client and server in go
- They will communicate with each other using sockets
- In this precept, we’ll walk you through a few useful functions that will help you start the assignment when it’s released

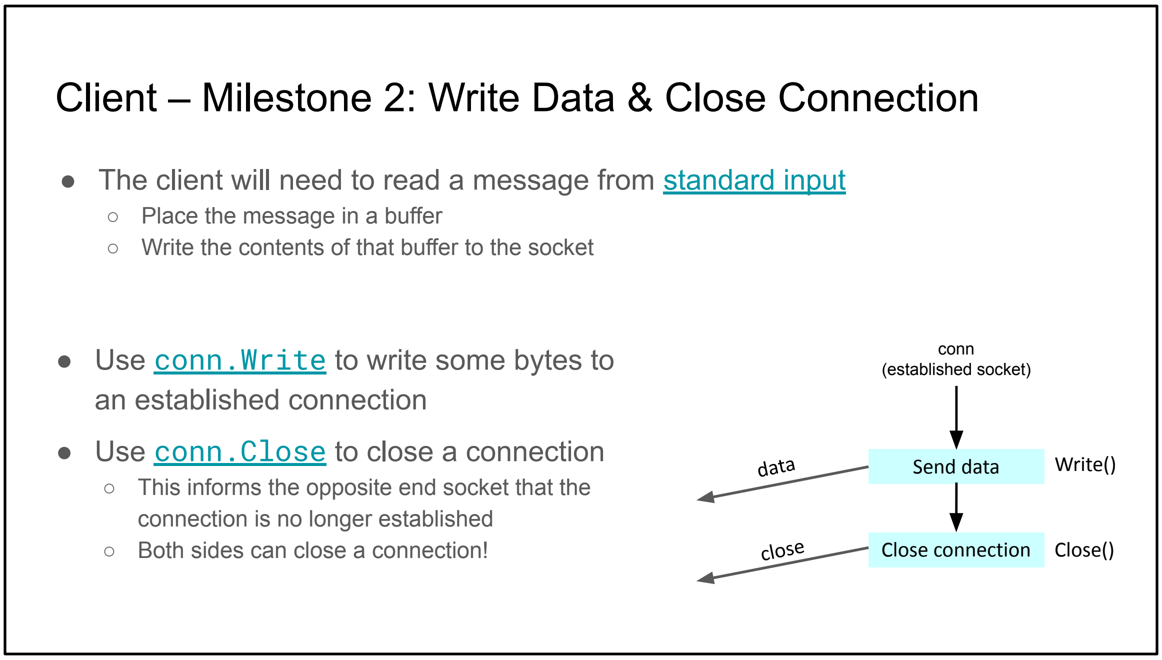


Client – Milestone 1: Connect to a Server

- We'll need to retrieve the server address from the command line
  … and connect to it
- go's `net.Dial` function looks promising!
  - Read its documentation to figure out the expected server address format
- Read the server address from the command line arguments
  - You can find those in `os.Args` in go!
  - The first argument (`os.Args[0]`) is always the executable name



- We know that the client will need to connect to the server
- We can use the Dial function in the net package for that
- Looking at the examples in the function's docs, we can see that the address and port are passed as a single string, being separated by a colon
- A server will listen on an address and a port, and a client will try to contact a server on an address and a port
- We should take in the address and port of the server from the command line
- To accept input, we can use the Args variable in the os package
- Like C, Args is an array whose first element is the name of the executable
- All other elements are arguments that were passed to the executable when it was launched
- From the documentation, you can see that net.Dial returns a Conn object and an Error object
- The conn object is what the client will use to communicate with the server
- The value of the error object should be checked and handled if necessary



Client – Milestone 2: Write Data & Close Connection

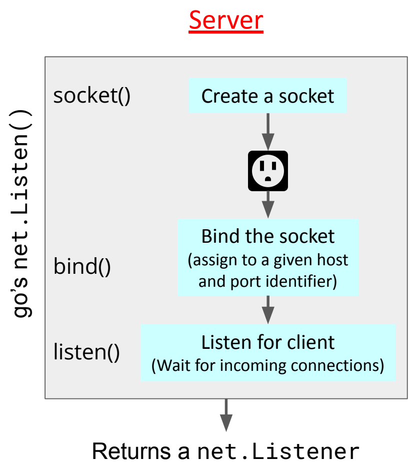
- The client will need to read a message from standard input
  - Place the message in a buffer
  - Write the contents of that buffer to the socket

- Use `conn.Write` to write some bytes to an established connection
- Use `conn.Close` to close a connection
  - This informs the opposite end socket that the connection is no longer established
  - Both sides can close a connection!

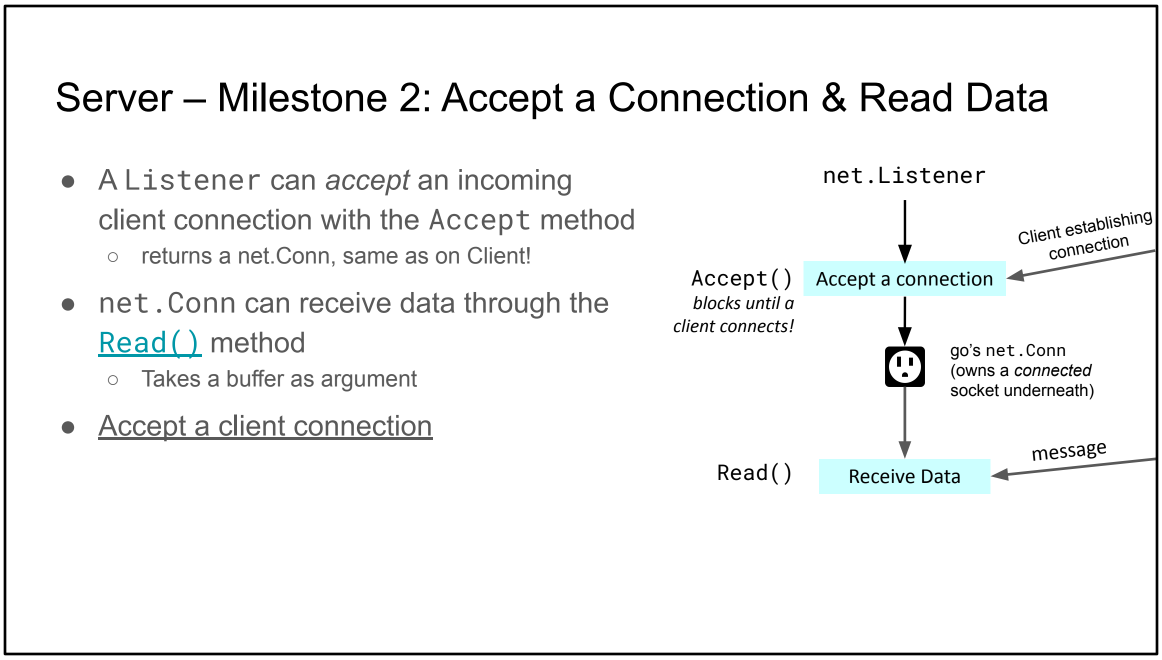
- Now that we have a socket that we can use to communicate with the server, we should take in input from the command line that we will then send to the server
- There are any number of methods. If you're unsure of how to do that, some googling should suffice.
- Once you have input saved in a buffer, you can call the Write method of the conn object to send it to the server
- Once the client sends the message to the server, it should exit. Before doing this, the client should invoke conn's Close method to close the connection

Server – Milestone 1: Create a Listening Socket

- To accept connections, our server must create a *listening* socket
  - The `net.Listen` function does that!
  - Returns a `Listener`, which owns a socket
- net.Listen takes a *listen* address
  - *Host*- and *process*-address of server (IP & port)
  - A server can have multiple host addresses! Listening on "localhost" or "127.0.0.1" only allows local connections.
- Use `fmt.Sprintf` to combine the host-address and port number



- Next, we'll go over some things that will be useful for implementing a server
- Remember that the relationship between a client and server is asymmetrical, a client proactively establishes connections while a server *listens* for incoming connection requests
- For the server, we need to create a listening socket that waits for incoming connection requests
- Let's take a look at the documentation to see what we need to pass to net.Listen
- Similar to net.Dial, listen takes a network, in our case tcp, and an address string, which is a concatenation of an ip address and port number
- It then returns a listener object, which the server can use

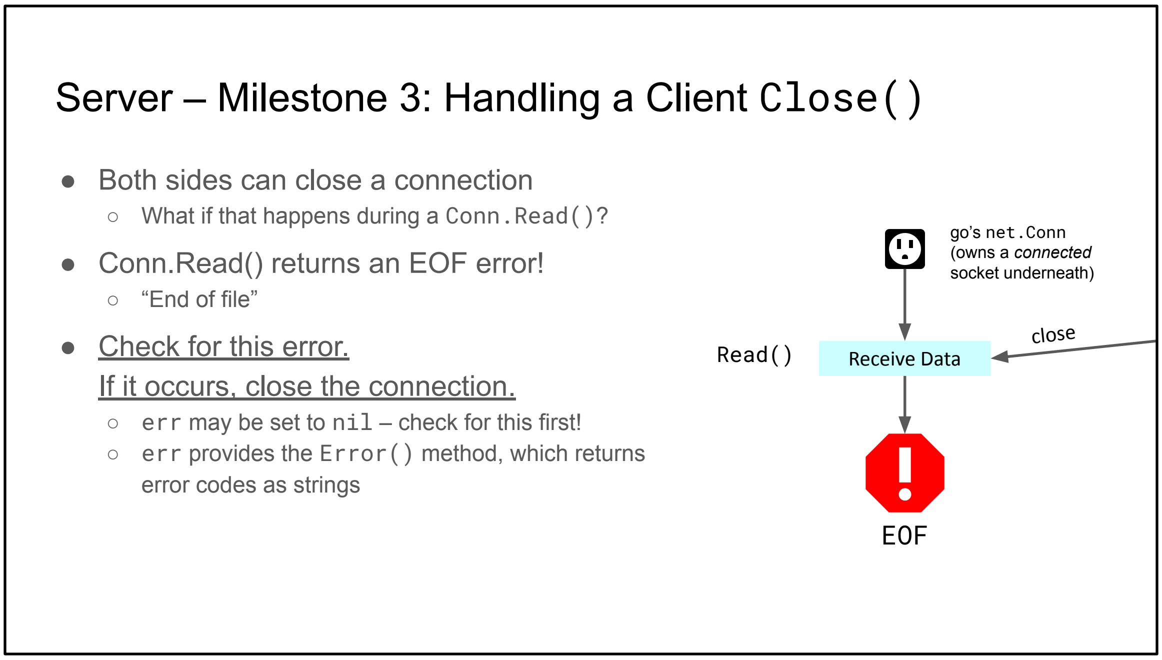


Server – Milestone 2: Accept a Connection & Read Data

- A `Listener` can *accept* an incoming client connection with the `Accept` method
  - returns a net.Conn, same as on Client!
- `net.Conn` can receive data through the [`Read()`] method
  - Takes a buffer as argument
- Accept a client connection

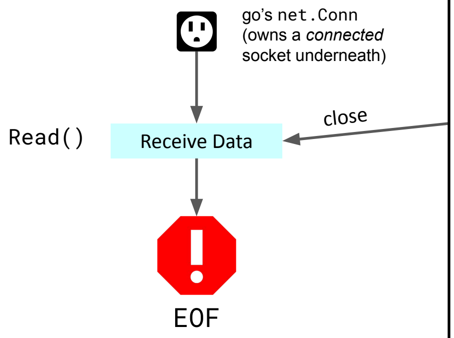
net.Listener
Accept()
blocks until a client connects!
Accept a connection
Client establishing connection
go's net.Conn (owns a connected socket underneath)
Read()
Receive Data
message

- Now that we've created a listening socket, we can invoke its *accept* method. This method blocks until a new connection attempt is received
- When it returns and execution of the program is allowed to proceed, it returns both a connection object and an error object
- We can check the value of this error object to see if anything went wrong
- And we can use the returned connection object to communicate with the client that requested the connection
- Note that there are now two sockets on the server side
- There's the listening socket that is always on the lookout for incoming connection requests
- And there's the socket for the established connection, which is used to send and receive data to a client that has already connected
- We can then use the conn object's read method to take in data that has been sent by the client
- From the documentation, we can see that the read method takes in a buffer and returns the number of bytes that were read from the connection and saved to that buffer, in addition to an error object
- We should then create a buffer to pass into the read function

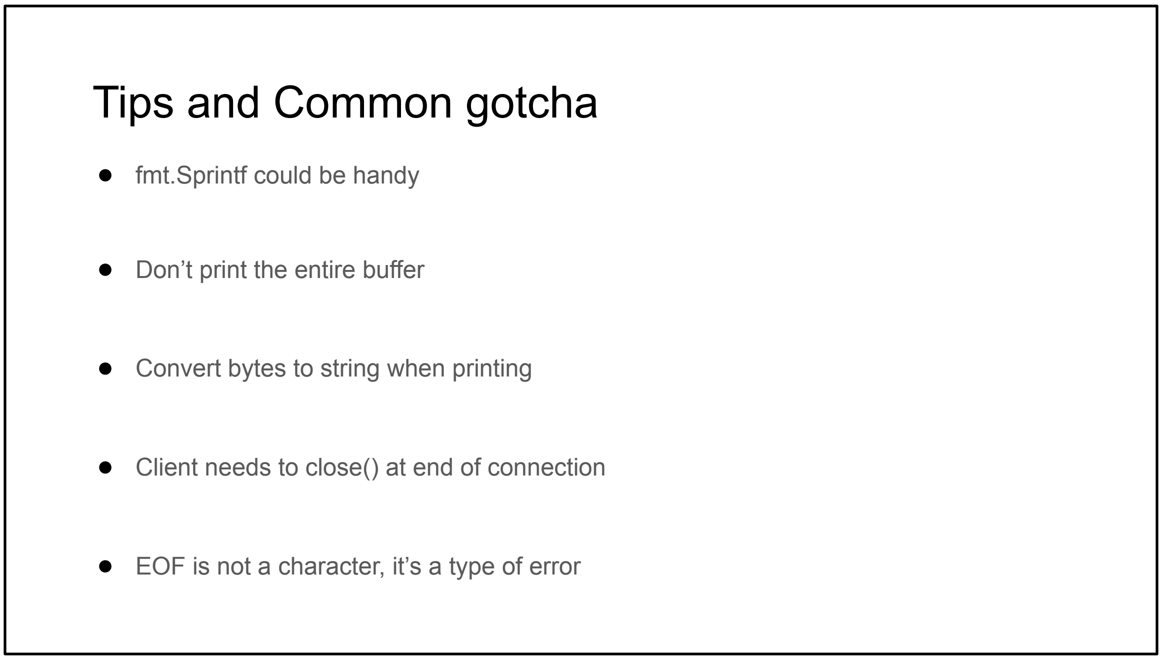


Server – Milestone 3: Handling a Client `Close()`

- Both sides can close a connection
  - What if that happens during a `Conn.Read()`?
- Conn.Read() returns an EOF error!
  - "End of file"
- Check for this error.
  If it occurs, close the connection.
  - `err` may be set to `nil` – check for this first!
  - `err` provides the `Error()` method, which returns error codes as strings



- Both the client and server are able to close a connection
- What should the server do if the client closes a connection?
- Remember that the read method returns an error object
- We can check the value of this error object to determine if the client has closed the connection from the other side



Tips and Common gotcha

- fmt.Sprintf could be handy
- Don’t print the entire buffer
- Convert bytes to string when printing
- Client needs to close() at end of connection
- EOF is not a character, it’s a type of error

- fmt.Sprintf can help you concatenate strings!
- Don’t print the entire buffer. You will make buffers with a fixed size and if a message doesn’t fill the buffer completely, then printing it out in its entirety may result in weird things happening
- The buffers that you’ll make are basically byte arrays. Convert them to strings before printing!