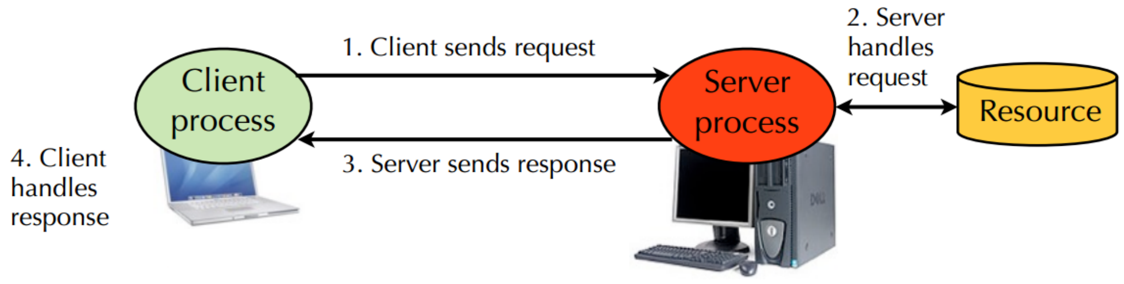
Section 9 Notes

Week 11/26-11/30

Networking

Clients and Servers

Oftentimes applications need to be split over the network, for various reasons. This includes everyday activities like accessing a webpage using a browser. These "network applications" are based on the **client-server model**, where a **server** manages some resource, and one or more **clients** send requests to the server, usually by manipulating the resource that the server manages. A server and clients are specific instances of **endpoints**, since they are on the ends of the communication in this case.



The server and the client(s) send and receive data over the network by some predefined protocol, like HTTP. For example, when you point your browser to the CS61 homepage, your machine sends a request defined by the HTTP protocol to the servers hosting the CS61 homepage, asking for the homepage html. The server extracts the client's request for the page, and sends back the html page to the client, who will then render Hello Kitty in all of its glory.

Question 1: Is it possible for a client process and a server process to be on the same machine?
Answer: Yes, the client-server model is not dependent on the network, and does not make assumptions about how the client and server are separated.

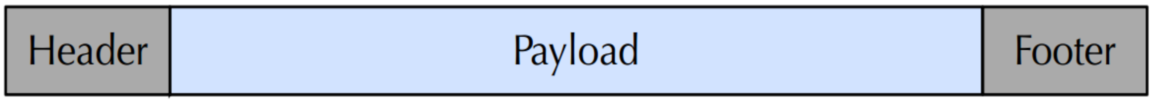
How exactly is data transferred between clients and servers? Data must be **marshalled** into some format suitable for network communication, the unit of which is called a **packet**. For example, if the server wants to transmit some data structure referenced by a pointer to a client, it must fetch the data structure itself, split the data structure up into packets, and send the packets to the client; it cannot simply pass a pointer to the client. In other words, marshalling is the process of transforming the memory representation of something to another data format, which in this case is simply a packet.

Packets

What exactly is a packet, and what does it look like?

A network packet is a **formatted unit of data which contains control information and user information**. They have a max size of **65,535 bytes** (the maximum value of a 16-bit word). It always consists of three fields:

1. **header**: contains routing information and metadata (control information)
2. **payload**: contains application data (user information)
3. **footer**: contains additional metadata (this field can be empty)



The header and footer metadata are determined by **protocols**, which simply specify what the fields are within the header and footer, as well as what type of data is in the payload. Some common protocols include IP, TCP, UDP, HTTP, SMTP, POP, IMAP, and AIM. Multiple protocols can be used in the same packet, as we'll see in a bit.

The payload contains the actual data that we want to transmit between machines. It's important to note that the data is always organized in big endian order in packets by convention.

Question: How would we transmit an integer from one endpoint to another?
Answer: Encode the integer as an ASCII string in the packet.

Before we dive into specific protocols, it's worth noting that the network is not reliable. Specifically, packets can be lost, duplicated, corrupted, and/or delivered out of order.

IP, TCP, and UDP are some of the core protocols in today's communications over networks. IP (the Internet Protocol) is used for routing packets across network boundaries. The most common flavor of IP is IPv4, and its header consists of 14 fields including the total length of the packet, a source IP address (where the packet is coming from), and a destination IP address (where the packet is going) among other things.

TCP (the Transmission Control Protocol) is used for providing endpoints a reliable, ordered, delivery of bytes from one computer to another computer. TCP abstracts the problems of the network (reliability, duplication, corruption, ordering, etc.) away from the endpoints. Specifically, it has the following characteristics:

- **end-to-end**: only the endpoints are responsible for communicating over TCP
- **reliable**: data is guaranteed to be received once in the correct order
- **stream-based**: endpoints don't need to think about splitting data into packets; they can just send bytes!
- **connection-oriented**: endpoints establish a connection before communicating

TCP is generally used as an intermediate level between IP - which dictates where a packet is coming from and where it is going - and some application protocol like HTTP or IMAP - which dictates what type of data will be in the payload. TCP's header consists of 10 mandatory fields and an extra field for options.

Question: How long does it take an endpoint to transmit a TCP request to another endpoint?
Answer: As long as necessary or until some pre-specified **timeout** has passed.

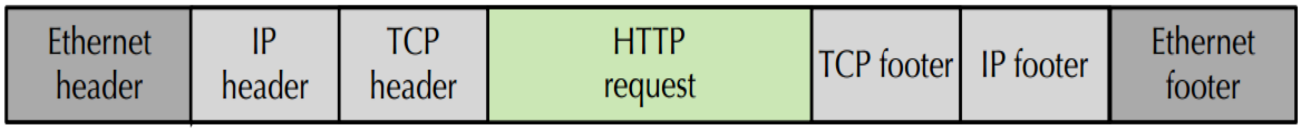
If an endpoint does not require a guarantee of delivery, correct ordering, or duplicate protection when sending data, it can use UDP (User Datagram Protocol) instead of TCP. UDP does not provide any of extra benefits that TCP guarantees, meaning that machines can simply just send a UDP packet without having to establish a connection or treat incoming and outgoing data like a stream. UDP also has a smaller footprint than TCP, as UDP's header only consists of 4 mandatory fields.

Question: For what types of data should UDP be used instead of TCP/How is UDP used?
Answer: UDP is used as an intermediate level between IP and application protocols that don't mind unreliable communication, like remote desktop or video streaming or other applications which use a high volume of packets without explicitly requiring the benefits of TCP.

A normal HTTP (Hypertext Transfer Protocol) packet would actually consist of many layered protocols:

- Ethernet layer
- IP layer - for determining where the packet is from and where it is going
- TCP layer - for providing reliability and a stream abstraction to the endpoints

- HTTP layer - request/response metadata and payload



An HTTP request is contained within a TCP packet, which is contained within an IP packet, which is contained within an Ethernet packet. This is called **packet encapsulation**, and is used for all sorts of protocols, not just TCP, IP, or HTTP.

Want to see what a real packet looks like? Download a **packet sniffer**, which monitors packets sent to and from your machine. You'll be able to analyze individual packets in packet sniffers, observing the different layers of packets and the different fields within layers/protocols. A popular packet sniffer is WireShark, which you can download at http://www.wireshark.org/download.html.

Threads

Inherently, networking introduces slow connections between the client and the server. Slow connections, unless handled in parallel, results in **low utilization**, meaning the process spends a lot of time doing useless work.

How can we solve this problem of low utilization? After all, many programs want to do many things all at the same time, each of which requires its own connection. For example, web browsers want to download web pages while chat clients want to send and receive messages. One answer is to run a new server for each new connection. However, this means there will be low utilization in terms of processes and memory, since each process needs its own address space; there will still be low utilization for every server that we use!

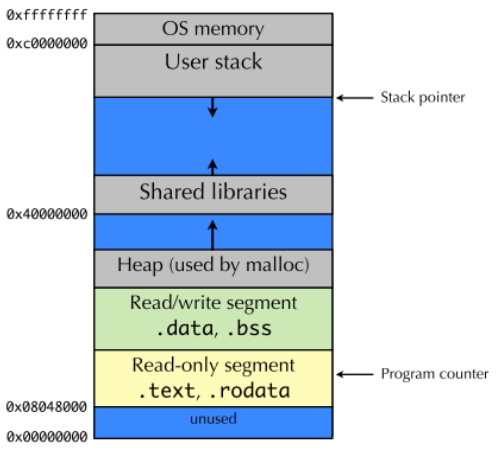
Using processes as a solution isn't very efficient. Recall that there is typically high overhead cost associated with each new process we create because processes have their own address space. Creating a new server or process requires copying the resource that the server is managing, which is pretty expensive. Instead, **a better idea is to handle connections in parallel within a single process** since handling connections is cheap. As a result, we don't reduce the server utilization. Using threads is the solution!

Threads

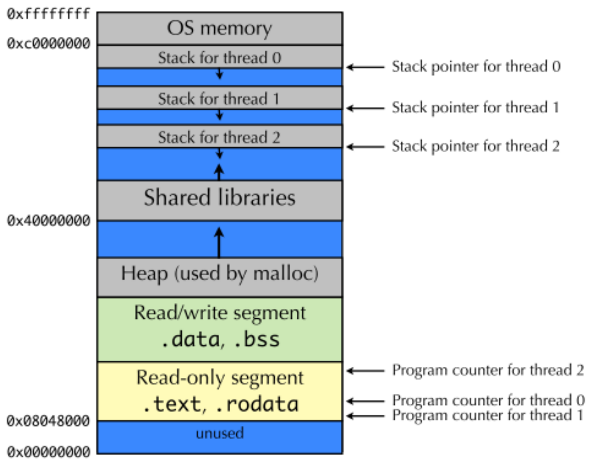
- A **thread** is a logical flow of control that runs in the context of a process.
    - At the machine level, if a process is an abstract/virtual computer, then a process with multiple threads is essentially an abstract multi-core computer.
- Each process has one of more threads contained within it:

    - Each process starts with a main thread.
    - New threads can then be created by older threads.
- Each thread has its own stack, stack pointer, program counter, and CPU registers.
- They share the same address space and OS resources as the process they are within, so they can communicate directly.

Old Process Address Space



New Address Space with Threads



The idea behind using threads on a server is that the memory state of the server is expensive to copy. We should have multiple virtual CPUs (threads) use the same memory state to get work done, thus avoiding expensive memory state copying.

How do we manage threads? The OS has a **process control block (PCB)** [one per process] and a **thread control block (TCB)** [one per thread] data structure. Each TCB contains information on a single thread, such as processor state and registers, and points to the corresponding PCB. The PCB contains information on the process address space and OS resource, but not on the processor state!

Using this concept, the process can be viewed as a container for the threads it owns; threads are now what the OS needs to schedule on the processor, not processes. Each CPU of the processor can run one thread at a time, but all the CPUs can be used to run threads in parallel.

Question: Which is faster: switching between two threads in the same process, or switching between two threads in different processes? Why?

Answer: Switching within the same process.

- Switching within the same process means we only need to save/restore CPU registers and the program counter.
- Switching across processes requires us to update to a new address space! We need to update the MMU and the TLB, and will have cache misses. (Check out virtual memory notes if you forgot what these mean).

Pthreads

There's a standard API for working with threads, also known as POSIX threads, or **pthreads**, in this pthread API. Here are just a few useful functions that are part of the pthread API:

- `int pthread_create(pthread_t *tid, pthread_attr_t *attr, void *(*start_routine)(void *), void *arg);`
    - tid: returns thread ID of new thread
    - attr: set the attributes for the new thread (scheduling)
    - start_routine: a pointer to the function for this new thread to begin execution
    - arg: argument to the start_routine()
- `void pthread_exit(void *retval)`
    - explicitly terminates the current thread
- `int pthread_cancel(pthread_t tid)`
    - terminate thread with specific TID

To kill a thread, programs can invoke pthread_exit to kill the current thread or pthread_cancel to kill a specific thread. Threads also terminate when the top-level process or the main thread terminates.

Code Example 0 - Creating Multiple Threads

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

volatile int var = 0;
void *run_thread1(void *arg);
void *run_thread2(void *arg);

void *run_thread1(void *arg){
    while(1)
        var++;
}

void *run_thread2(void *arg){
    while(1){
        printf("In thread2, var is %d.\n",var);
        sleep(1);
    }
}

int main(int argc, char **argv){
    pthread_t thread1, thread2;
    pthread_create(&thread1, NULL, run_thread1, NULL);
    pthread_create(&thread2, NULL, run_thread2, NULL);
    pthread_exit(NULL);
}
```

Code Example 1 - Merge Sort!

Hopefully you understand the merge sorting algorithm!

Question: How can we implement merge sort using threads?
Answer:

- Recursively break the array into smaller blocks, and have **separate threads** sort the blocks in parallel.
- The smallest blocks will be sorted using an insertion sort.
- Then, we merge the sorted results from the threads, working our way back up to the completed result.

Code Example 2 - Sieve of Eratosthenes

The Sieve of Eratosthenes is a simple algorithm for finding all primes up to a given number. The algorithm filters all composite numbers that are multiples of the next prime it finds, starting from 2. For example, the algorithm will declare 2 a prime, and then filter out all multiples of two, since they are not prime. The algorithm will then go to the next unfiltered number, 3, filter out all multiples of 3, do the same for the next unfiltered number (5), and so on until all the primes have been found up to a given number.

How can we use threads to make a simple version of the sieve?

- Every prime has its own thread that is used for filtering out its multiples.
- Start with a single thread that is assigned the prime number 2. We will call this thread the 2-thread.
- Every thread has a stream of inputs, which has been filtered by the previous thread.
    - For instance, the main thread will pass all numbers to the 2-thread. The 2-thread will pass all numbers not multiples of 2 to the next prime thread, which will now be designated as the 3-thread. The 3-thread will then pass those numbers less the multiples of 3 to the next prime thread, the 5-thread, and this process will repeat until some threshold.
- The "next prime thread" is really a thread waiting to receive its prime assignment. For example, there is initially only one thread with a prime assignment (2); the 2-thread passes on the next prime (3) to the next prime thread, so the current "next prime thread" becomes our 3-thread. This prime assignment mechanism continues for each prime under the threshold.

Question: While this method is easy to understand, it's actually really inefficient. Why?
Answer: Threads early in the chain will be doing work on almost every number, so they enjoy good utilization. As we get to larger primes though, threads will continue to run but will be doing no useful work until they get a number that happens to be filtered down to them. They will continue to run and check that they have an assignment (**busy loop**) until they get an input that they can do work on.

Never fear! We can improve performance by doing a few tweaks and using...

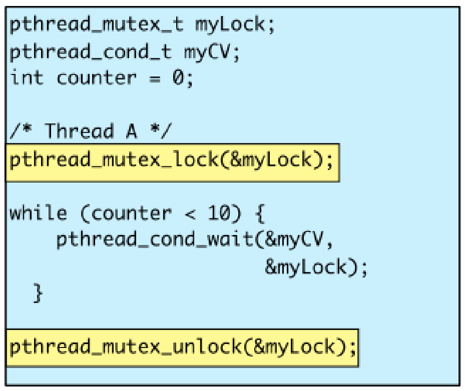
Locks and Condition Variables

The threads associated with larger numbers will spend a lot of time in a busy loop checking for an assignment, resulting in low utilization. Additionally, these threads will hog cycles away from the threads who are actually doing useful work, most notably the really early threads like the 2-thread and the 3-thread. Ideally, these large number threads should be ignored or put to sleep until there is a prime to be assigned to a thread; at that point, a thread should be woken up and should be allowed to run with its new prime number assignment.

Turns out there's a mechanism for this behavior called **condition variables**. A condition variable, or CV, represents a condition for a thread, and isn't itself a condition. A thread can **wait** (or go to sleep) until some condition occurs, and a thread can also **signal** to other waiting threads that some condition has occurred.

For example, let's say that we want to spawn new threads and tell them to wait until they have received a prime assignment (a condition that hasn't occurred); all new threads will then immediately go to sleep unless they already have an assignment. When an early thread, like the 2-thread or the 3-thread, have found a new prime to be assigned (the condition has occurred), they can signal to one of the waiting threads that a prime is ready to be assigned.

- The reason we use CVs are for **mutual exclusion**, which allows us to synchronize access to shared memory
    - **Example**: threads want to decrement a global counter. Decrementing a counter consists of three assembly instructions:
        - First, they must get the value from the shared memory.
        - Next, they decrement the value.
        - Finally, they store the new value.
    - Without mutual exclusion, **we don't know the order that these events occur in** across multiple threads, and we can't reliably get a result that we want.
- All CV operations require that a thread performs a job using a **lock**. **Why are locks necessary?**
    - **Answer**: We want to guarantee that events will occur in some proper order in the larger picture.
    - **Example**: decrement a counter while the counter != 0. Without a lock, we may just skip 0 and go into negative values! There is no guarantee that the threads will increment or decrement the counter properly.

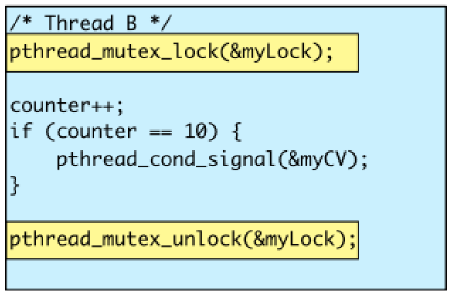


```
pthread_mutex_t myLock;
pthread_cond_t myCV;
int counter = 0;

/* Thread A */
pthread_mutex_lock(&myLock);

while (counter < 10) {
    pthread_cond_wait(&myCV,
                      &myLock);
  }

pthread_mutex_unlock(&myLock);
```



```
/* Thread B */
pthread_mutex_lock(&myLock);

counter++;
if (counter == 10) {
    pthread_cond_signal(&myCV);
}

pthread_mutex_unlock(&myLock);
```

- If there is no lock for Thread A, it might wait until after another thread sets the counter to 10 before proceeding.
- If there is no lock for Thread B, there is no guarantee that incrementing will occur properly!

- **Locks** are objects in memory that allow us to **enforce mutual exclusion** and thus make use of CVs.
- Remember that address spaces and thus memory is shared among threads, so CVs require that **threads acquire a lock** in order to access or change some condition, like a counter for instance.
- **The lock is released** when the job is done (condition has been checked or changed), and the next thread is allowed to obtain the lock to check or change the condition.
- If a thread doesn't own the lock, it can **sleep** so it doesn't do useless work. We get improved utilization!

Question: How can we improve the sieve implementation from before?
Answer: Using **CVs** and **locks**, we can make sure each thread does work only when it gets a value. A thread gets a lock when it has a value to filter, and sleeps when there is no work to be done.