Threads vs. Processes
Up until now, we’ve discussed **threads** -- the fundamental unit of computation -- and we know they’re organized into **processes**.

- Threads within a process share nearly all resources (exceptions are few, like the PC and their stack frames).
- Processes are almost completely independent from other processes.

<table>
<thead>
<tr>
<th>Threads</th>
<th>Processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creation</td>
<td></td>
</tr>
<tr>
<td>Overhead</td>
<td></td>
</tr>
<tr>
<td>Context Switching</td>
<td></td>
</tr>
<tr>
<td>Virtual Memory</td>
<td></td>
</tr>
</tbody>
</table>

Case Study: Chrome

Inter-Process Communication (IPC)
IPC is the broad terminology for all technologies that facilitate real-time communication between processes.

**Approach #1:**
Using a pipe within a terminal:

```
$ ps -aux | grep waf
```

Creating pipes in C:

```
int pipe(int pipefd[2]);
```

**Approach #2:**

**Approach #3:**
Approach 4: _________________
Allocating shared memory in C (“malloc for shared memory”):

```c
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

Approach 5: _________________
Functions in C:

```c
mqd_t mq_open(const char *name, int oflag);
int mq_send(mqd_t mqdes, const char *msg_ptr,
            size_t msg_len, unsigned int msg_prio);
ssize_t mq_receive(mqd_t mqdes, char *msg_ptr,
                   size_t msg_len, unsigned int *msg_prio);
int mq_close(mqd_t mqdes);
```

Approach 6: _________________

Approach 7: _________________

Creating a new socket interface:

```c
int socket(int domain, int type, int protocol);
```

Binding a socket interface to an address and port:

```c
int bind(int sockfd, const struct sockaddr *addr,
         socklen_t addrlen);
```

Connecting to a remote socket:

```c
int connect(int sockfd, const struct sockaddr *addr,
            socklen_t addrlen);
```

Begin listening for a remote socket connection:

```c
int listen(int sockfd, int backlog);
```

Start a new socket channel with a remote host:

```c
int accept(int sockfd, struct sockaddr *restrict addr,
           socklen_t *restrict addrlen);
```

**Networking**

Q: What do we expect out of networking?

...making this happen is **insanely complex**:

<table>
<thead>
<tr>
<th>Hosts</th>
<th>Protocols</th>
<th>Packet Errors</th>
<th>Out-of-Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>Routers</td>
<td>Hardware</td>
<td>Link Failures</td>
<td>Packets</td>
</tr>
<tr>
<td>Links</td>
<td>Software</td>
<td>Node Failures</td>
<td>Eavesdropping</td>
</tr>
<tr>
<td>Applications</td>
<td>Bit Errors</td>
<td>Message Delays</td>
<td>...and more...</td>
</tr>
</tbody>
</table>

We define common ____________ -- a message format and rules for exchanging messages. You know many protocols already:

**Network Packets**

At the core, network data is simply a series of 0s and 1s, which we represent in hex. (You can view all of the network packets on Linux using `tcpdump -x`.) For example, here one of many packets used in a request for me to view waf.cs.illinois.edu:

```
00 4500 00c6 1e1f 4000 4000 152e ac16 b24c
10 12dc 95a6 bafa 0050 0f60 c9b4 356a 523f
20 8018 01f6 079e 0000 0101 080a 8146 30a0
30 31d4 daac 4745 5420 2f20 4854 5450 312e
40 2e31 0d0a 5573 6572 2d41 6765 6e74 3a20
50 5767 6574 2f31 2e32 302e 302e 302e 302e
60 7578 2d67 6e75 290d 0a41 6363 6572 747a
70 282a 2f2a 0d0a 4163 6365 7474 2d4d 6663
80 6f64 6f6e 677a 2069 6465 6e74 6974 790d
90 0a48 6f73 743a 2077 0d0a 436f 6e6e 6563
a0 6c69 6765 6773 2e67 6f73 743a 2077 0d0a
b0 4653 7469 666e 3a20 4b65 6570 2d4d 6663
```

```
c0 7665 0d0a 0d0a
```