**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).
  - **AND**
- Processes are almost _____________ from other processes.

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

**Case Study: Chrome**

- On a given system with many tabs open, Chrome will have **dozens and dozens or processes**.
- Why?

---

**Inter-Process Communication (IPC)**

IPC is the broad terminology for all technologies that facilitate real-time communication between threads. Many approaches:

1. ________________

Using a pipe within a terminal:

\[
\text{
\$ ps -aux | grep waf
}\]

Creating pipes in C:

\[
\text{int pipe(int pipefd[2]);}
\]

2. ________________

Allocating shared memory in C (“malloc for shared memory”):

\[
\text{void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);}
\]

3. ________________

Sending a signal within a terminal:

\[
\text{
\$ kill -TERM <pid>
}\]

Listing all available signals:

\[
\text{
\$ kill -l
}\]

Sending a signal in C:

\[
\text{int kill(pid_t pid, int sig);}
\]

Handling a Signal in C:

\[
\text{
\struct sigaction {
    \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \  \}
}\]
\]
4. _________________

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);
```

5. _________________

...IPC is the strong use-case for semaphores and the only time I use them. (Other people's opinions vary. :))

6. _________________

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 Packets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Routers</td>
<td>Hardware</td>
<td>Link Errors</td>
<td>Eavesdropping</td>
</tr>
<tr>
<td>Links</td>
<td>Software</td>
<td>Node Failures</td>
<td>...and more...</td>
</tr>
<tr>
<td>Applications</td>
<td>Bit Errors</td>
<td>Message Delays</td>
<td></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 your VM 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 4006 152e ac16 b24c
10 018 01f6 079e 0000 0101 080a 8146 30a8
20 081d4 3daa 4745 5420 4500 00c6 1e1f 4000
30 006 0060 0060 0060 0060 0060 0060 0060
40 006 0060 0060 0060 0060 0060 0060 0060
50 006 0060 0060 0060 0060 0060 0060 0060
60 006 0060 0060 0060 0060 0060 0060 0060
70 006 0060 0060 0060 0060 0060 0060 0060
80 006 0060 0060 0060 0060 0060 0060 0060
90 006 0060 0060 0060 0060 0060 0060 0060
a0 006 0060 0060 0060 0060 0060 0060 0060
b0 006 0060 0060 0060 0060 0060 0060 0060
```