From Hardware to Software: Moving to the OS!
Up until now, everything we have discussed has been focused on the
direct interaction with hardware. Today, we escape hardware to
explore controlling and using the hardware!

Operating System (OS):

<table>
<thead>
<tr>
<th>Apps</th>
<th>OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td></td>
</tr>
<tr>
<td>[2]</td>
<td>Hardware</td>
</tr>
<tr>
<td>[3]</td>
<td></td>
</tr>
</tbody>
</table>

Pressing the Power Button
When you press the power button on ANY computing device, not much is available:

- Your ________________ after being off (ex: does not contain your Operating System, that’s on your disk!)
- Goals of a computer at boot:
  1. [POST]:
  2. [Bootloader]:

To do this, almost all modern systems perform three tasks:

1. [POST]:

2. [Bootloader]:

Running a Program
A _____________ is an instance of a running program.
  - NOT the binary source code; it’s an instance of it running.

A _____________ provides two key abstractions:

1. [Memory]:

2. [Execution]:

Once one process exists, it can spawn new processes through the
fork() command. You can explore every process on a system:

- Linux command: **ps**
- ...options for **all** processes with details: **ps -aef**
**Multiprogramming**
On a modern computer, there are dozens of different processes running simultaneously -- but only a few CPUs.

- In the period of microseconds, the OS rapidly switches between all processes to **allow each process to run on one or more of the CPUs**.
- When the OS swaps out one process from one CPU and allows a new process to run, this is called a _________________.

**Context Switching**
What is required during a context switch?

[CPU]:

[Caches]:

[Page Table]:

...overall cost?

---

**A Process' Execution Unit: Threads**
Each process contains one or more ________________ that can run concurrently with other threads:

[1]: Main Thread:

[2]: Uses for Additional Threads:

---

**Threads vs. Processes**

<table>
<thead>
<tr>
<th></th>
<th>Threads with a Process</th>
<th>Processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overhead</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Context Switching</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Virtual Memory</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPUs</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Creating Threads in C**
The pthread library is the POSIX thread library allowing you to create additional threads beyond the main thread.

Creating a new thread is a complex call with four arguments:

```c
int pthread_create(
    pthread_t *thread,               /* thread struct */
    const pthread_attr_t *attr,      /* usually NULL */
    void *(*(start_routine) (void *)), /* start func */
    void *arg                          /* thread start arg */
);
```

The start_routine variable is a function pointer and requires the argument to be a function with the prototype:

```c
void *__________(void *ptr);
```

...you can use any name for the function name.
Example 1: Launching Fifteen Threads

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

const int num_threads = 15;

void *thread_start(void *ptr) {
    int id = *((int *)ptr);
    printf("Thread %d running...\n", id);
    return NULL;
}

int main(int argc, char *argv[]) {
    // Create threads:
    int i;
    pthread_t tid[num_threads];
    for (i = 0; i < num_threads; i++) {
        int *val = malloc(sizeof(int));
        *val = i;
        pthread_create(&tid[i], NULL,
                        thread_start, (void *)val);
    }
    printf("Done!\n");
    return 0;
}
```

Q1: What is the expected output of this program?

Q2: What actually happens?

Q3: What do we know about threads in C?

Example 2: Joining Threads

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

int main(int argc, char *argv[]) {
    // Create threads:
    int i;
    pthread_t tid[num_threads];
    for (i = 0; i < num_threads; i++) {
        int *val = malloc(sizeof(int));
        *val = i;
        pthread_create(&tid[i], NULL,
                        thread_start, (void *)val);
    }
    // Joining Threads
    for (i = 0; i < num_threads; i++) {
        pthread_join(tid[i], NULL);
    }
    printf("Done!\n");
    return 0;
}
```

In the above program, we use `pthread_join`. This call will block the CPU from running the program further until the specified thread has finished and returned.

Q4: What happens in this program?

Q5: Does the order vary each time we run it? What is happening?

Q6: What can we say about the relationship between “Done” and “Thread %d running...” lines?
Q7: What do we expect when we run this program?

Q8: What is the output of this program when it’s running as:
   
   ./count 100 2

Q9: What is the output of this program when it’s running as:
   
   ./count 100 16

Q10: What is the output of this program when it’s running as:
    
    ./count 10000000 2

Q11: What is the output of this program when it’s running as:
    
    ./count 10000000 16

Q12: What is going on???