CS 240 Computer Systems

Week 6: Multiprogramming and Threads

CS 240, Spring 2021 - Week 6

Wade Fagen-Ulmschneider

From Hardware to Software: Moving to the OS!

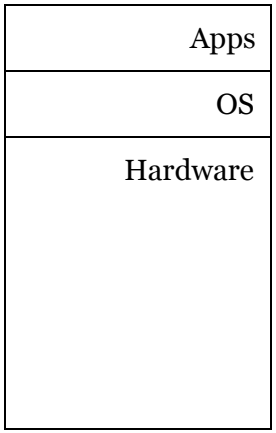
Up until now, everything we have discussed has been focused on the direct interaction with hardware. Today, we escape hardware to explore **controlling** and **using** the hardware!

Operating System (OS):

[1]:

[2]:

[3]:



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]:

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.

  2.

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

1. [POST]:

2. [Bootloader]:

3. [Process #1]:

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

	Threads with a Process	Processes
Overhead		
Context Switching		
Virtual Memory		
CPUs		

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:

```
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:

```
void *_________(void *ptr);
```

...you can use any name for the function name.

Example 1: Launching Fifteen Threads

threads/fifteen.c

```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++) {
    pthread_create(&tid[i], NULL,
                                thread_start, (void *)&i);
  }

  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

threads/fifteen-join.c

```c
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?

threads/count.c

```c
int ct = 0;

void *thread_start(void *ptr) {
  int countTo = *((int *)ptr);

  int i;
  for (i = 0; i < countTo; i++) {
    ct = ct + 1;
  }

  return NULL;
}

int main(int argc, char *argv[]) {
  // Parse Command Line:
  if (argc != 3)  {
    printf("Usage: %s <countTo> <thread count>\n", argv[0]);
    return 1;
  }

  const int countTo = atoi(argv[1]);
  if (countTo == 0) { printf("Valid `countTo` is required.\n"); return 1; }

  const int thread_ct = atoi(argv[2]);
  if (thread_ct == 0) { printf("Valid thread count is required.\n"); return 1; }

  // Create threads:
  int i;
  pthread_t tid[thread_ct];
  for (i = 0; i < thread_ct; i++) {
    pthread_create(&tid[i], NULL,
                   thread_start, (void *)&countTo);
  }

  // Join threads:
  for (i = 0; i < thread_ct; i++) {
    pthread_join(tid[i], NULL);
  }

  // Display result:
  printf("Final Result: %d\n", ct);
  return 0;
}
```

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???