Lecture 3

Main goal: Thread

Dual Mode Operation

- Processes execute in **user mode**
  - To perform privileged actions, processes request services from the OS kernel
  - Carefully controlled transition from user to kernel mode
- Kermel executes in **kernel mode**
  - Performs privileged actions to support running processes
  - And configures hw to use it
- Carefully controlled transitions between **user** & **kernel** mode.
  - Calls, interrupts, exceptions
- **Ex.** Let's say we're trying to switch between processes P0, P1
  - Save state into PCB0
    - PCB = **process control block**
  - Reload state from PCB1
  - Run Process P1
  - Save state into PCB1
  - Reload state from PCB0
  - Continue P0
- Running many programs
  - We have the basic *mechanism* to
    - switch b/w user processes and kernel
    - kernel can switch among user processes
    - protect OS from user processes and processes from eachother
  - Questions
    - How to represent user processes in OS?
    - How to decide which proc to run?

        - How to pack up the process and set aside?
        - How to get a stack and heap for the kernel?
    - Muxing Processes - PCB
        - Kernel represents each process w/ a PCB
            - status (running, ready, blocked, ..)
            - registre state (if not ready)
            - process id (PID), user, executable, priority
            - exec time
            - memory space translation
        - Kernel *Scheduler* maintains a data structure containing the PCBs
            - Give out CPU to diff processes
            - Not a policy decision
        - Give out non-CPU resources
            - Memory/IO
            - Another policy decision

Scheduling

- Scheduling: Mechanism for deciding which procs/threads receive hardware CPU time, when, and for how long
- SMT/Hyperthreading
    - Hardware scheduling technique
        - Avoids software overhead of multiplexing
        - Superscalar processors can execute multiple independent instructions
        - Hyperthreading duplicates register state to make a second "thread", allowing more instructions to run
    - Can schedule each thread as if it was a separate CPU
        - However, the speed-up is sublinear
- A couple definitions
    - **Multiprocessing**: Multiple CPUs
    - **Multiprogramming**: Multiple jobs / processes

    - **Multithreading**: Multiple threads / processes
- Motivation: There are some processes that take orders of magnitude longer, and we should split up threads so that we aren't hung up on one job that takes a while
    - Masks the I/O latency because you have other things running in the meantime
- To create more threads, we'll need to make **syscalls** (system calls)
    - Usually created by an API, such as pthreads
- Fork-Join method (used in pthreads)
    - Main thread creates / forks a collection of subthreads and gives them args to work on. Then, it joins with them to collect the results.

OS Library API for Threads: `pthreads`

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *
(*start_routine)(void*), void *arg);
```

- creates thread executing `start_routine` w/ `arg` as its sole argument
- return is an implicit call to `pthread_exit`

```
void pthreaed_exit(void *value_ptr);
```

- terminates the thread and makes `value_ptr` available to any successful join

```
int pthread_join(pthread_t thread, void **value_ptr);
```

- suspends exec of the calling thread until `thread` terminates
- on return with a non-NULL `value_ptr`, the value passed by `pthread_exit` by the terminating thread is made available in the location ref'd by `value_ptr`.

TL;DR

1. Threads are the OS unit of concurrency

    1. Abstraction of a virtual CPU core
    2. Can use `pthread_create`, etc to manage threads within a process
    3. They share data, so need synchronization to avoid data races.
2. Processes consist of ≥ 1 threads in an address space.
    1. Abstraction of the machine: execution env for a program
    2. Can use `fork` (new thread), `exec`, `join`, etc to manage threads within a process
    3.