UCB CS 162 - Operating Systems and Systems Programming "Lecture 3 - Abstractions 1- Threads and Processes"

Lecture 3 - Abstractions 1: Threads and Processes

Multiprocessing vs. Multiprogramming

Multiprocessing vs. Multiprogramming is similar to the idea of concurrency vs. parallelism. Multiprocessing is when you have multiple physical cores, and you use them to perform operations at the same time completely independently. Multiprogramming is when you divide a program into multiple jobs/processes/threads.

Concurrency is when you are *handling* multiple things at once, not necessarily *doing* them. Parallelism is when you do multiple things *simultaneously*.

Threads

A thread can be in one of 3 states - running, ready, or blocked. A running thread is a thread whose instructions are actually being executed, a ready thread is a thread that is waiting for CPU time, and can be swapped out with the current running thread, and a blocked thread is a thread that is waiting for something to happen before it can run (typically i/o).

To write a multithreaded program in C, we usually use pthreads. The "P" in pthreads stands for POSIX. POSIX is an attempt at standardizing the programming interface that different operating systems provide to programs running in user mode.

Usually the structure of a multithreaded program is that we have a "main" thread that is created when the process is created. The main thread creates as many new threads as it needs. These threads run, do their own thing, and then "join" with the main thread. The main thread then continues to run.

All threads in a process share the process's address space, but they have their own independent stack. These stacks grow and shrink independently, and sometimes may

run into each other. In this case, the OS decides what to do. The thread may be killed, or its stack may be reallocated.

The important part about multithreaded programming is correctness. The program you write must be able to handle any interleaving of thread execution that the OS scheduler gives you. One way to ensure correctness is mutual exclusion.

If more than one thread is going to access some data, the thread accessing that data acquires a "lock" over that data. This ensures that only that thread can read or write to that location. Once the thread has done whatever it needs to do with that data, it releases the lock. The part of code that exactly one thread can access at once is called the critical section.

Processes

All processes are created by other processes. If that is the case, how is the first process created? The first process is called the "init" process, and it is created by the kernel.

The OS also provides a process management API, that allows our programs to manage other processes. The common functions provided by this API are -

1. `exit` - Terminate a process
2. `fork` - Create a complete copy of the current process
3. `exec` - Change the program being run by the current process
4. `wait` - Wait for a process to finish
5. `kill` - Send an interrupt-like notification to another process
6. `sigaction` - Set handlers for signals