codecademy

Concurrent Programming

Computer Programs

A *computer program* is a collection of instructions used to perform a certain task.

Computer Process

A *process* is an abstraction used to represent a program while it is in execution.

Process States

A process typically exists in five states: New, Ready, Running, Blocked, or Finished

Context Switching

Context switching allows CPU cores to alternate between ready and blocked processes to best take advantage of limited computing resources.

Preemption

Preemption occurs when a process is temporarily interrupted by an external scheduler to prioritize a more important task.

Blocked Process

A process is *blocked* when it has to wait for a contested, limited, or slow resource, such as accessing a specific file or waiting for a network request.

Process Layout

The *layout* of a process in memory has four distinct sections:

- A text section for the compiled code
- A data section for initialized variables
- A stack for local variables
- A heap for dynamic memory allocation

Process Control Block

Every process is initialized with a *process control block* that is required by the operating system to be able to identify and control the process.

Process Parent-Child Relationship

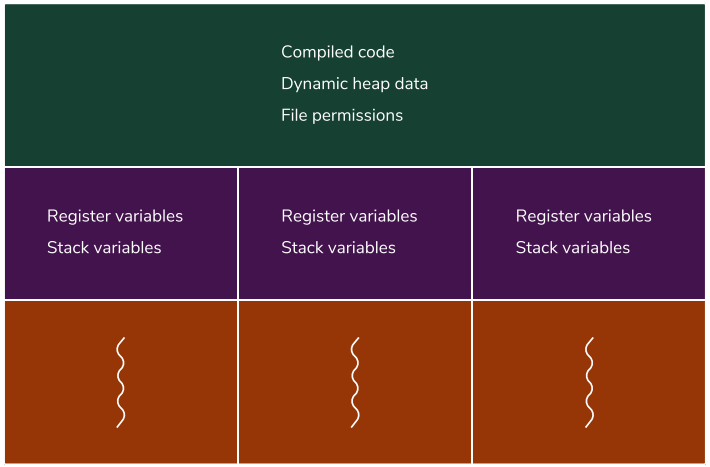
When a process launches another process, the original process enters a *parent-child relationship* with the new process. This relationship facilitates the sharing of common data and signals along the hierarchy as well as the arrangement of which process may terminate first.

Process Thread

A *thread* represents the sequence of programmed instructions that are actively being executed. They share resources which allows for faster communication and context switching as well as requiring fewer system resources when compared to processes.

Process Multithreading

Multithreading is the capability for a single CPU core to execute multiple threads at once. This improves system utilization and responsiveness by more efficiently splitting up tasks



Kernel Threads

Kernel threads are threads created in kernel space using kernel code and libraries through a system call. The kernel is fully aware of these threads and can properly manage them.

User Threads

User threads are threads created in user space using local code and function calls. The kernel is not aware of these threads and cannot directly control them. User threads allow for more fine-grained control by developers and are more efficient than kernel threads as they do not need to make system calls.

User vs Kernel Threads

User threads can be mapped to kernel threads in a variety of ways: 1:1 Kernel-Level threading, N:1 User-Level threading, or M:N Hybrid threading.