Cpu API - SistOp

Lautaro Bachmann

Contents

Interlude: Process API

The fork() System Call

The fork() system call is used to create a new process

process identifier,

also known as a PID. the PID is used to name the process if one wants to do something with the process, such as (for example) stop it from running.

fork() system call,

the process that is created is an (almost) exact copy of the calling process. The newly-created process doesn't start running at main(), rather, it just comes into life as if it had called fork() itself.

nondeterminism,

leads to some interesting problems, particularly in multi-threaded programs;

The wait() System Call

Sometimes, as it turns out, it is quite useful for a parent to wait for a child process to finish what it has been doing. This task is accomplished with **the wait() system call** (or its more complete sibling waitpid()); the parent process calls **wait()** to delay its execution until the child finishes executing. When the child is done, **wait()** returns to the parent.

The exec() System Call

This system call is useful when you want to run a program that is different from the calling program.

What it does:

given the name of an executable and some arguments it loads code (and static data) from that executable and overwrites its current code segment (and current static data) with it;

it does not create a new process; rather, it transforms the currently running program into a different running program

a successful call to exec() never returns.

Why? Motivating The API

the separation of fork() and exec() is essential in building a **UNIX shell**, because it lets the shell run code after the call to fork() but before the call to exec();

The shell

- It shows you a **prompt** and then waits for you to type something into it.
- You then type a command into it; in most cases, the shell then figures out where in the file system the executable resides, calls **fork()** to create a new child process to run the command, calls some variant of **exec()** to run the command, and then waits for the command to complete by calling **wait()**.
- When the child completes, the shell returns from **wait()** and prints out a prompt again, ready for your next command.

Redirecting Output

The way the shell accomplishes this task is quite simple: when the child is created, before calling **exec()**, the shell closes standard output and opens the file By doing so, any output from the soon-to-be-running program are sent to the file instead of the screen.

file descriptors.

UNIX systems start looking for free file descriptors at zero. In this case, **STDOUT_FILENO** will be the first available one and thus get assigned when open() is called.

UNIX pipes

UNIX pipes are implemented in a similar way, but with the pipe() system call. In this case, the output of one process is connected to an in-kernel pipe (i.e., queue), and the input of another process is connected to that same pipe; thus, the output of one process seamlessly is used as input to the next,

Process Control And Users

there are a lot of other interfaces for interacting with processes in UNIX systems.

For example,

the kill() system call is used to send signals to a process, including directives to pause, die, and other useful imperatives.

UNIX shells,

certain keystroke combinations are configured to deliver a specific signal to the currently running process;

for example, control-c sends a SIGINT (interrupt) to the process (normally terminating it) and control-z sends a SIGTSTP (stop) signal thus pausing the process in mid-execution

signals subsystem

provides a rich infrastructure to deliver external events to processes, including ways to receive and process those signals within individual processes, and ways to send signals to individual processes as well as entire process groups.

signal() system call

It is used to "catch" various signals; doing so ensures that when a particular signal is delivered to a process, it will suspend its normal execution and run a particular piece of code in response to the signal.

who can send a signal to a process, and who cannot?

modern systems include a strong conception of the notion of a user.

The user,
after entering a password to establish credentials, logs in to gain access to system resources. The user may then launch one or many processes, and exercise full control over them **Users generally can only control their own processes;** it is the job of the operating system to parcel out resources to each user to meet overall system goals.

Useful Tools

ps command

allows you to see which processes are running;

The tool top

it displays the processes of the system and how much CPU and other resources they are eating up.

The command kill

can be used to send arbitrary

signals to processes, as can the slightly more user friendly killall.

KEY PROCESS API TERMS

- Each process has a name; in most systems, that name is a number known as a **process ID (PID).**
- The **fork()** system call is used in UNIX systems to create a new process. The creator is called the **parent;** the newly created process is called the **child.** the child process is a nearly identical copy of the parent.

- The **wait()** system call allows a parent to wait for its child to complete execution.
- The **exec()** family of system calls allows a child to break free from its similarity to its parent and execute an entirely new program.
- A UNIX **shell** commonly uses fork(), wait(), and exec() to launch user commands; the separation of fork and exec enables fea- tures like **input/output redirection, pipes,**
- Process control is available in the form of **signals,** which can cause jobs to stop, continue, or even terminate.
- Which processes can be controlled by a particular person is encapsulated in the notion of a **user;** the operating system allows multiple users onto the system, and ensures users can only control their own processes.
- A **superuser** can control all processes this role should be assumed infrequently and with caution for security reasons.