CS 350 Operating Systems

Keven Qiu
Instructor: Bernard Wong
Winter 2024

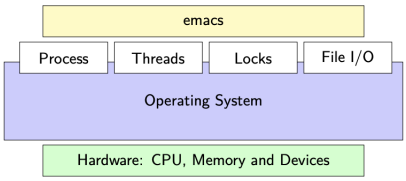
Chapter 1

Introduction

Definition: Operating System (OS)

The layer between applications and hardware.

It usually provides abstractions for applications. This includes managing and hiding details of hardware and can access hardware through low level interfaces unavailable to applications. It often provides protection.



Primitive OS are just a library of standard services (no protection). The system runs one program at a time and there are no bad users or programs.

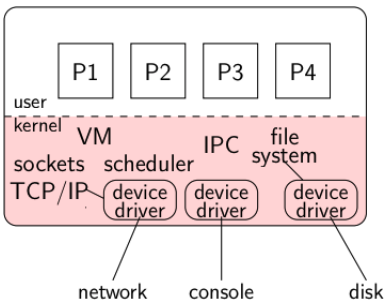
Multitasking is the idea of running more than one process at once. An ill-behaved process can go into an infinite loop and never relinquish the CPU. It can also scribble over other processes' memory. The OS provides mechanisms to address these problems:

- Preemption: take CPU away from looping.
- Memory protection: protect process' memory.

Multi-user OS use protection to serve distrustful users/apps. With n users, the system is not n times slower. Users can use too much CPU (need policies), total memory usage is greater than in the machine (must virtualize), or a super-linear slowdown with increasing demand (thrashing).

The OS structure: Most software runs as user-level processes and the OS kernel runs in privileged mode.

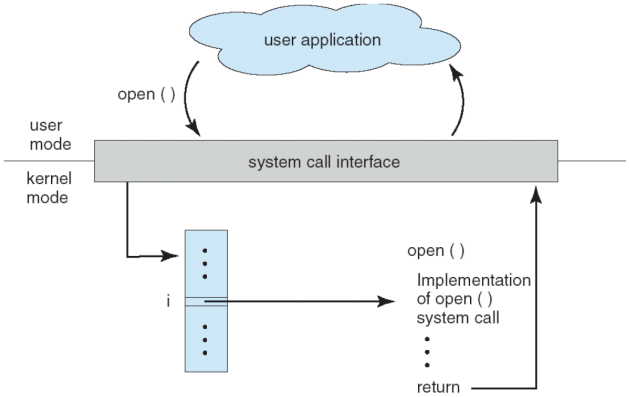
Protection:



- Pre-emption: give application a resource, take it away if needed elsewhere.
- Interposition/mediation: place the OS between application and "stuff". Track all pieces that the application is allowed to use. On every access, look in the table to check that access is legal.
- Privileged & unprivileged modes in CPUs: applications are unprivileged and the OS is privileged. Protection operations can only be done in privileged mode.

Definition: System Calls

Applications can invoke the kernel through system calls. These are special instructions that transfers control to the kernel.



The goal for system calls is to do things applications cannot do in unprivileged mode. The kernel supplies well-defined system call interface.

Applications set up syscall arguments and trap to kernel. The kernel performs operation and returns result.

Higher-level functions are built on the syscall interface. E.g. `printf`, `scanf`, `gets`, etc.

These call the POSIX/UNIX interface. E.g. `open`, `close`, `read`, `write`, etc.

Chapter 2

Processes

Definition: Process

An instance of a program running and the environment for running the program.

Modern OSes run multiple processes simultaneously.

- Multiple processes increase CPU utilization: overlap one process' computation with another.
- Multiple processes can reduce latency: running A then B may require more time than running A and B concurrently.

Each process has its own view of the machine. It has its own address space, open files, and virtual CPU.

2.1 User View of Processes

2.1.1 Creating Processes

`int fork(void);` creates a new process that is an exact copy of the current one. It returns the process ID of the new process in the parent. It returns 0 in the child.

The only part that is shared between the parent and child are the open files, otherwise the address space and code are its own.

`int waitpid(int pid, int *stat, int opt);` waits for a child process to terminate. `pid` is the process to wait for, or -1 for any. `stat` contains the exit value, or signal. `opt` is usually 0 or `WNOHANG`. `waitpid` returns the process ID that it waited for, or -1 if error.

2.1.2 Deleting Processes

`void exit(int status);` exits the current process. By convention, status of 0 is success and non-zero is error.

`int kill(int pid, int sig);` sends the signal `sig` to process `pid`. `SIGTERM` is the most common value and kills the process by default. `SIGKILL` is stronger and always kills the process.

2.1.3 Running Programs

`int execve(char *prog, char **argv, char **envp);` executes a new program. `prog` is the full path name of the program to run, `argv` is the argument vector that gets passed to main, and `envp` are the environment variables (PATH, HOME, etc.).

It is generally called through wrapper functions.

- `int execvp(char *prog, char **argv);` searches PATH for `prog` and use current environment.
- `int execlp(char *prog, char *arg, ...);` list arguments one at a time and finish with `NULL`.

2.1.4 Manipulating File Descriptors

`int dup2(int oldfd, int newfd);` closes `newfd`, if it was a valid descriptor and makes `newfd` an exact copy of `oldfd`.

Two file descriptors will share the same offset, i.e. `lseek` will affect both.

2.1.5 Pipes

This allows reading and writing between two processes like the parent and child.

`int pipe(int fds[2]);` returns two file descriptors in `fds[0]` and `fds[1]`. `fds[0]` reads and `fds[1]` writes. When the last copy of `fds[1]` is closed, `fds[0]` will return EOF. `pipe` will return 0 on success and -1 on error.

Operations on pipes:

- read/write/close.
- When `fds[1]` is closed, `read(fds[0])` returns 0 bytes.

- When `fds[0]` is closed, `write(fds[1])` kills the process with SIGPIPE or if the signal is ignored, it fails with EPIPE.

Most calls to fork are followed by execve. We can also combine into one spawn system call.