Note 19: OS and Virtual Memory

Kevin Moy

1 Introduction

The OS allows you to run multiple programs at the same time through parallel execution.

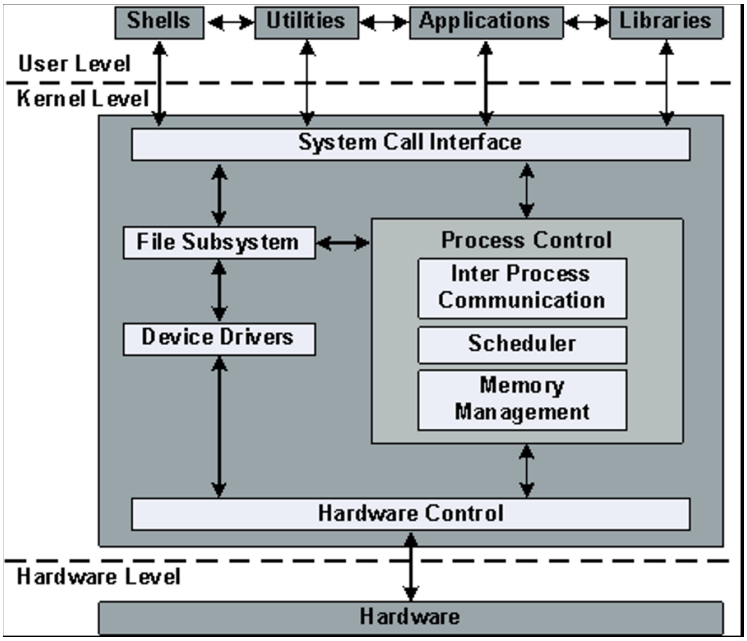
2 Operating System

Operating systems control HOW software accesses and uses hardware. It is the interface for actions such as reading and writing to disk. It also allows for **software portability**- being able to run on any machine.

What does the OS do? The main point of the OS is to **load, run, and manage programs**. When multiple programs are running at the same time, we call this **time-sharing**. It manages this by not only running them at the same time but ensuring their execution stays isolated. It also handles multiplexing between applications. The OS also manages our file system as well as network stack. Finally, the OS utilizes *device drivers* to manage all devices on the machine.

2.1 OS Levels

There are three sub-levels to the OS: user level, kernel level, hardware level. Below is a diagram of these levels as well as a display of how the OS interacts with software and hardware:



This is a representation of a *microkernel*, a very small kernel controlling the processes. It is important to also abstract these layers from each other for security purposes.

What's different about these operating systems? A particular OS has a specific design for:

- Appearance/Layout (compare MacOS vs Windows)
- Interfaces for applications (Windows applications for Windows only)
- Licensing, availability, hardware support (compare Linux open source to Mac which is not)

CS prefers a UNIX-based machine.

3 OS Boot Sequence, Operations

Let's look at the general machine startup sequence. When your computer boots, CPU executes instructions from some start address (PC), which is stored in flash ROM. Usually the first instructions in the boot sequence is loading the BIOS- finding a device and loading it into sector 1 (first block

of data). Next is the bootloader, which is stored on disk and loads the OS kernel from disk into memory. Next is the OS boot sequence: initializing the OS, services, and drivers necessary to interface with the computer. Finally it launches `init` which allows us to launch apps and wait for input.

In most OS, apps are called *processes*. The method for launching a new process is system-dependent and thus OS-dependent. For example, UNIX/Linux uses `fork`for new processes and `execve` for new apps.

Loading applications simply loads its executable from disk and places its instructions/data into memory- sort of like the assembler directives `.data` and `.text`. The OS prepares the stack and heap, then set `argc` and `argv` then jumps into `main`.

The OS also needs to enforce isolation of processes and resource constraints (like device access), to mitigate damage from faults or malware. To protect the OS from applications, CPUs have a bit to turn on **supervisor mode**, which allows the OS to access and run certain dangerous instructions and unlock some memory. This is a mode ONLY allowed to run when upon some interrupt or trap- no arbitrary jumps into it!

A syscall is a process where we set up function arguments then raise a software interrupt. This allows enabling of supervisor mode (from user mode). Once the OS performs the operations it needs it returns to user mode. This way, OS can mediate all resource access.

Venus actually provides many simple syscalls using the`ecall` RISC-V instruction. We place appropriate syscall number in `a0`, place syscall *arguments* in `a1`, then call `ecall`.

4 Multiprogramming and Time-Sharing

Parallel application execution is actually just an illusion!

The OS actually is able to switch between processes extremely quickly- a **context switch**, as we are changing program contexts upon switching.

The OS also has to dynamically decide which program to run- called **scheduling**. Different program schedules have various resource requirements, runtime speed, importance (priority programs first), etc.

Virtual memory gives each process the *illusion* it has more memory accessible than actually exists. It gives each process its own memory space, only accessible to itself. This solves issues of memory overlap and overwriting!

Virtual memory (VM) is a very large, near infinite memory space only **specific to a single process**. Physical memory (PM) is the limited RAM the computer must share among all processes. Virtual memory, when implemented effectively, does three main important things:

- Isolates processes
- Translates a potentially infinite address space to finite without much notice from program
- Translates VM -¿ PM addresses.

Each process will actually think it has the entire block of RAM to itself (and more: actually, the entire address space). Virtual memory actually connects main memory and disk. It's the same idea as caches.