**Review of OS and Computer Architecture**

You have a **CPU** doing most of the work, **main memory** where you store everything, I/**O devices**, and everything is connected by the **system bus**.

The main OS services are: **Protection** through kernel and user mode, protected instructions and base/limit registers. **Interrupts** which use interrupt vectors. **System calls** using trap instructions and trap vectors. **I/O** which interrupts when done and also uses memory mapping. **Scheduling, error recovery, accounting** which are implemented using the timer to change processes on a given timeframe. **Synchronization** which is implemented through Atomic Instructions that cannot be interrupted. **Virtual Memory** using the disk space to give the illusion that we have infinite memory.

**System Call Implementation**

Every system call has a number associated with it. These numbers are used by the system call interface as a way of maintaining its table for system calls. It uses the number to invoke the correct system call, the actual implementation of the system call is handled behind the scenes.

If the application wants to open a file, the **system call interface** looks up the number for the open system call, has that system call executed in **kernel mode** and then returns back to the **user mode** with a **return value**.

**OS Structure**

In the basic structure of the OS we have a few layers that work together. We have all the user programs at the top where the user is running programs and these programs are using their compilers, interpreters, and the system libraries that they need. They communicate to the kernel through the **system call interface to the kernel.** This is where we move into the kernel mode operations. We call the protected part of the OS that runs in kernel mode the **kernel** and it basically runs as one large process. The kernel communicates to the hardware through the **kernel interface to the hardware.**

There is a lot of the OS that sits above the kernel. We may typically think about the kernel when we talk about the implementation of the OS but the majority of the OS is the APIs that sit above it. When we are users, we aren’t even interacting with the kernel, we are interacting with the GUI that interacts with the API that ultimately talk to the kernel.

More commonly we use **Layered OS Design** over **Monolithic OS design.** With the layer design we have more organization where each layer just talks to the layers directly above and directly below. An example would be:

User programs

|

Device Drivers

|

Virtual Memory

|

I/O channel

|

CPU Scheduler

|

Hardware

We do sacrifice some efficiency by using layered design. If we want a system call that involves the CPU Scheduler, going through virtual memory when we don’t have to is not efficient. Or what if the CPU Scheduler wants to use virtual memory, it has to go through multiple steps to get there. **Layered Design is not as common**.

In the **Monolithic design**, we basically give the kernel everything. All of the functionality exists within the kernel and it call all talk to each other. We try to take as much out of user mode processes as we can.

A **Microkernel** is the exact opposite of the Monolithic design. We try to put as much as we can with user processes and make the kernel as small as we can. There are some processes that have to be done in the kernel, but a lot of what you would expect to be in there are actually running in user mode above the microkernel. Things like file systems, thread systems, and high level scheduling. We see increased resilience with this as well, if the file system goes down due to a bug, the rest of the computer might still be able to run. In a layered design, everything downstream would be effected.

The downside of the microkernel is that all processes need to be able to communicate with each other and they have to go through he kernel to do it. They can’t directly communicate with each other, they have to talk to a third part first essentially. This reduces efficiency.

The design that is **used today** is called the **kernel module.** We have multiple, **independent components** that communicate with each other over **premade interfaces**. With this we can load the modules to the kernel as needed. This means they **lie dormant** until they are needed. This excludes the necessary parts of the OS that have to be running at all times.This is almost like running a core kernel inside of the larger modular kernel.

The disadvantage of this is that we are running all of these modules within the same kernel space. This means if we have one module that is compromised with bad or malicious code, the **entire kernel is vulnerable**. The entire kernel can be taken down by one module.

**TLDR**

We are always working with a tradeoff when we are structuring OS. We want the OS to be simple but also efficient and that is the tradeoff we work with. Simplicity is the stronger focus unless we believe that its work it to be more complicated in order to increase performance.