**Review of Introduction**

**User applications** run on top of the **OS** through a **virtual machine interface**. The **OS** runs on the **physical hardware** through a **physical machine interface.**

**Modern Operating System Functionality**

It is the job of the OS to **manage all the processes and threads** going on in the machine. Users start applications which run as processes and processes can have multiple threads. It is the job of the OS to manage these using the limited resources it has available.

The OS must also manage **concurrency** which means doing many things simultaneously. With lots of processes going on at once, there is a need to do multiple things at the same time. We want all users and programs to have the appearance like they have the machine all to themselves. The OS has to manage threads which are the basic unit of OS control. Th CPU can only do one thread at a time but will have many threads active so it must be managed.

The OS has to **manage I/O** which are much slower than the CPU. The CPU can do lots of things while waiting for the I/O devices and this time needs to be used in a clever way. We don’t want to be waiting on slower devices while there is other stuff to be done.

The OS is also in charge of **memory management.** The OS coordinates the allocation of all memory on the device. When there is not enough memory to store all the active processes, the OS must also decide what gets sent to disk space and when it needs to be moved back and forth between memory and disk space.

The OS must also **coordinate** **files** and how disk space is used to store files. It needs to find the best way to store files so they can be found quickly and don’t take up too much space. Also the OS needs to know how to reclaim storage space after a file is deleted. It needs to know how to reuse that space.

When we are talking about multiple machines running on the same hardware, the OS needs to also be able able to allow the machines to work together on the same hardware. This is called **distributed systems and networks.** If we have multiple virtual machines running on the same computer it needs to figure out how to run them all on the same hardware and also handle the case if they need to talk to each other and work together.

**Summary of OS Principles**

**OS as a juggler:** making sure every user and application has the illusion that they are the only machine running on the hardware and that they have infinite memory and CPU. Make sure all users are not constricted.

**OS as government:** protecting users from each other. No users should be able to access or do things to other users that they are not allowed to. Resources also need to be allocated fairly and efficiently. One user shouldn’t get all the good stuff at the expense of another user but at the same time, if it is more efficient to give one user a bit more than another without negatively impacting them, that is whats best. There should also be safe and secure communication options between users.

**OS as a complex system:** Keeping the OS design and implementation as simple as possible. OS are very complex but we want to keep things as simple as possible.

**Generic Computer Architecture**

**CPU:** the processor that performs the actual computations. Modern processors have multiple cores.

**I/O devices:** terminals, disks, video boards, keyboards. How we communicate with the computer and how it communicates with us. This also includes the network card.

**Memory:** RAM containing data and programs used by the CPU. This is where we store instructions and data.

**System Bus:** these are the communications between the different parts of the computer. It is the medium between the CPU, Memory, and the I/O devices.

**OS Servies**

**Protections:**

Protecting programs, the OS, memory, and handling traps.

Protecting the programs and users from each other as well as protecting the OS itself. The CPU has a list of **assembly instructions** that it can do. Some of these instructions are **sensitive or privileged**. The OS will not allow just any program to call a halt for example because one program shouldn’t be able to stop the entire machine. These privileged instructions are also what restrict certain programs from being able to access memory that they shouldn’t be able to.

We have **Kernel Mode** and **User Mode.** This is how the OS restricts some users from certain instructions. User mode will **restrict the user from**:

**Accessing I/O directly**.

Using Instructions that **manipulate the state of memory**.

**Changing the bits that say which mode the application is**. The architecture has a bit that says if you are in user or kernel mode and this bit needs to not be changed.

Using **disable and enable interrupts**.

**Halting** the machine.

But in **kernel mode the OS can do all these things**.

The hardware must be able to at least support user and kernel modes although it is more common for there to be even **more layers** now that fill in the gaps between **allowing more privileges as you get closer to kernel mode** while not allowing everything.

User mode can ask the kernel to do stuff for it. These are called **system calls.** System calls are an OS procedure that a user program calls on to perform an instruction that it is not allowed to do on its own. Its an API that the kernel exposes for the user to use. This issues what is called a **trap** which is when the kernel takes over from the user for a bit. When this happens, the mode bit that determines which mode we are in **flips** to make it kernel mode, the **system call then executes the instruction** on your behalf and then **returns** you to user mode. The OS does this so **it can check what you want to do** before you do it. It wants to make sure you aren’t doing anything bad and then executes it.

There are multiple system calls that can be used for both **process and file management.**

**Memory Protection**

The OS must be able to protect programs from each other as well as protecting itself from programs. The OS does this by implementing **Base** and **Limit Registers.** The **base register is the memory location where the valid memory addresses start and the limit register is where the valid memory addresses end**. These are both **loaded by the OS before** the program is even started so it can easily and quickly tell if the memory you are trying to access is allowed or not. **You are only allowed to access the memory addresses between those two registers.**

Processes have three segments in memory: **text, data, and stack. Text** is where the **actual code** of the program lives. Since the actual code of the program is not going to change, this is statically allocated at the end the space allocated in memory. The **stack** is where the **temporary variables are declared, stored, and initialized**. The stack is used a **scratch paper** for the program. The values here are not saved permanently and are **reallocated** once the program ends. The **data or heap** is where data is saved. If you declare an object, this is where it is saved. Between the stack and data is a gap that allows for each to expand. The stack expands down and the data expands up.

**Registers** are a basic unit of memory. They are **one word of memory** that is managed by the CPU. There are **general purpose** registers which are used to just store whatever data you are using. There are also **Special Purpose** registers like **stack pointers, frame pointers, and program counters**. Stack and Frame are used to manage the actual **layout of the memory**. While the program counter is telling you what instruction you are on and what instruction is next. When we are chaining processes or switching between them, we do **context switches** which is when we **save the current registers**, **load the registers** of the process we are switching to, once we want to switch back we do another context switch to load the registers of the original program. The OS handles context switches.

We also have a **memory hierarchy** which is the levels of memory we have in the machine**.** Memory that is stored higher up is **smaller and faster to access**. This is revered for registers and they can be accessed in one cycle. It is very small.

As you go down the hierarchy it gets slower but the memory space gets larger. Below the registers you get the **L1** **cache**, then **L2** **cache**, then **RAM**, then **disk** space. We rely on the caches to keep things fast. The slowest we have is the **network**. The caches are actually stored in the CPU to make them as fast as possible. L1 and L2 are managed by the hardware not the OS.

**Traps** are **special conditions** that are detected by the architecture. These are conditions like system calls or when you try to write to a read only page. Basically the program wants to do something that the architecture flags to say “I don’t know about that”. When a trap is detected, the **hardware saves the state of the process** (the program counter, the stack, etc.) and transfers control to the **appropriate trap handler**. The CPU has a **memory mapped trap vector** which has the types of traps listed and the memory address of that specific type of traps handler. Once it finds the address of that specific handler, it **executes the handler at that address** and once it finishes executing the trap handler it **returns to the original process** from that saved spot.

**I/O Control**

I/O devices run **autonomously**, they have their **own processors**. The CPU sends a command to the I/O device and then **continues onto the next task** because I/O devices are very slow.

The CPU is actually going to do other things until it hears back from the device. When the device does end up completing its task, it issues a **command to interrupt**. The CPU then stops whatever it was doing and the **OS processes the devices interrupt**. Interrupts are also a usual part of your machines operation. Something to think, in a lot of cases when we are talking about I/O we are referring to **getting files and data from disk space**. Not waiting on a keyboard or monitor or something.

There are three different ways to handle I/O. We have **synchronous, asynchronous, and memory mapped.** When we are talking about these ways, we are talking about the way the individual processes handle I/O. The processor will almost always be working asynchronously.

In **Asynchronous** I/O, the requesting process (the process requesting I/O) issues a request through the device driver. While the device driver is processing this, the request process goes and **starts working on its next steps** while it waits for the I/O. Once the hardware is done processing the I/O, the return data is sent to the device driver and an **interrupt** is sent to the requesting process and it goes back to the part where it was waiting and receives the data.This is also called **non-blocking I/O.**

**Synchronous I/O** is when the requesting process does not proceed to the next step and waits until the data is returned before it moves on. We might have to do this if the rest of the process is reliant on the data we receive. This is also called **blocking I/O.** This isn’t necessarily that bad for efficiency because **the processor can still run asynchronously** and do other things, this is just on the level of the single program itself which may have needed to wait anyways while the processor was doing other things.

**Memory-Mapped I/O** is when we want to read a **very large file** and don’t want to go one byte at a time. The data is most likely not stored all in one place and if we have a very large file, we would have to send an interrupt for every byte we get from I/O. This would mean **processing a ton of interrupts**. Rather than that, we have the I/O controller talk **directly to memory** without talking to the CPU. By doing this we can have the I/O controller move all the needed data directly into memory **without having the CPU get involved**. Once the entire file is moved into memory, then we send **one interrupt** to the CPU who can then **access everything super quick** because its all in memory which is fast.

In Synchronous and Asynchronous, the CPU is having the data given to it from (most likely) disk space. This works for smaller files where we only have to move a couple bytes of data because we only have to through a few interrupts as they are received. Only a couple of moving trucks are needed. In Memory-Mapped, we are moving a large file to memory from disk space because it will take a ton of moving trucks and we don’t want to have to send an a ton of interrupts which would slow the process down. We only have to send one interrupt once the file is loaded and it can be accessed in its entirety.

The way the CPU handles these interrupts is with the following steps:

1. **Saves the CPU state** (the hardware state) this includes the registers.

2. We then have to **disable interrupts** so we don’t receive anymore. We do this so we don’t keep getting interrupts that push the first one back.

3. **Save the software state**. We don’t want to override any data that the user program was using.

4. **Invoke the interrupt handler** using the in-memory interrupt vector. This is where the CPU “looks up” how to handle this type of interrupt.

5. **Restore software state** so no data the user program was using is lost.

6. **Re-enable interrupts** so the next ones can be processed.

7. **Resort hardware state** and **continue the execution of interrupted processes**. Anything that was stopped needs to be restarted.

**Timer & Atomic Instructions**

Timers are needed for things like **time of day**, as well as **accounting and billing**. Accounting and billing is used when we are scheduling multiple processes, we don’t want to be giving all of the time to certain processes so we measure how much time each one is getting. This helps with the OS governance. We do this by setting an arbitrary time where every 100 microseconds or so the **timer issues an interrupt** and the CPU is told to **work on something else**. This way it never spends too more than 100 microseconds on one task in one go. **Every timer interrupt, the OS gives the CPU something else to work on**.

There are some sequences in processes we might want to run without being interrupted by the timer or something else. This is called **Synchronization** and we have two ways we can do it.

1. We can just **disable interrupts** while the sequence is executed then turn them back on.

2. We could give special hardware instructions that **execute atomically**. There are special instructions that are **immune to interrupts** such as test and set.

**Virtual Memory**

We want to give the processes the **illusion that they have unlimited memory** so they aren’t restricted but we do actually have limited physical memory that we can give them. The answer to this is virtual memory. If we have a program saved on the disk, when we run it we actually **don’t have to move the entire program to memory**, we can just load it piece by piece as needed. If we have executed the first line of the program, why would we keep it in memory? Once we are done with a part of the program, we can loaded the next part where it is since we don’t need it anymore. We don’t have the entire program in memory but the process doesn’t know that it. **We take the parts of the program we need from disk space as we need them and save the next parts overtop of them once we are done.**

The OS has to keep track of what pieces of virtual memory are actually in physical memory, it has to keep track when its done with them, and it needs to keep track of when it needs to load the next part into physical memory. It is constantly swapping data between memory and the disk to give the illusion of infinite memory.

**TLDR**

The OS provides an interface to the machines physical architecture but also requires the architecture to support the functions we want to do.

The OS has a user and a kernel layer so not all processes and users can do everything. This provides protection for each user, program, and to the OS itself.

The OS must also manage memory. It ensures that programs can’t access parts of other programs unless it checks it out first. These are called traps and are handled by the trap handler.

The OS handles I/O, I/O is typically very slow so it has ways of either working on other stuff while its waiting and is given an interrupt once the data is ready. It can load large files to memory so it doesn’t have to handle a lot of interrupts.

The OS has a timer so it can ensure no process gets all the resources. It changes between them every so often. Some processes might have parts that really need to not be interrupted so interruptions can be disabled as needed.

The OS manages virtual memory. This is when programs are stored on disk space and only moved to memory as needed, giving the illusion that the memory is infinite.