**Review of OS Structures**

There are 5 main parts of the computer. The **processor**, **memory**, **I/O devices**, **file system**, and **distributed systems**. These fill the 5 main categories of functionality for the OS: **Process management, Memory management, interrupt handling, File management, and Networking**.

System Calls are the **interface** we use to use the services made **available by the OS**. Usually written in **C/C++** and sometimes assembly. Basically, when we write a program, even simple ones, there are lots of actions that need to be made behind the scenes by the kernel. A system call is used when we need to do something that cannot be done at the user level. **Its something we ask the kernel to do for us**.

The types of system calls are: **process control, file manipulation, device manipulation, information maintenance, communication, and protection**.

There are four **architectures for OS kernels**. **Monolithic** (all in one level), **layered** (layers onto of each other and can only talk one layer up and down), **microkernels** (make the kernel as small as possible, make a lot system and user programs), and **modules** (main kernel is small but has access to smaller, independent kernels with specific tasks).

**Processes**

**What is in a Process**

**Processes** are programs that are **currently executing**. While programs are static, processes are **dynamic**. Multiple processes can run a single program, but each is a **distinct** process within. Anytime you do anything on the computer, you are starting a process.

Processes are executing steps one at a time, these steps are the code for running the process and they are called static data. Processes also include **dynamic data (the heap)** and the **heap pointer**. Processes have a **program counter** pointing to the next step in the process. They have an **execution stack** for the programs call chain (**the stack**) and the **stack pointer**. Processes have access to **CPU registers**. **OS resources** like files that are open for the process. Processes have a **process execution state** as well **(ready, running, etc.)**

**Process Execution State**

The process execution state tells us **what the process is currently doing**. There are a couple states it could have: **New** (OS is setting ups the process state), **Running** (executing instructions on the CPU), **Ready** (ready to run but waiting on CPU), **Waiting** (waiting for an event to complete like an I/O completion)**,** **Terminated** (the OS is destroying this process).

Something to note, a process cannot go from waiting to running. All processes that are in the **waiting state must go to the ready state before they can go to running**. This is so the CPU never has to wait and can be sure that when it moves to continue executing that process, the process won’t cause any delays.

**Process Control Block**

The Process Control Block or **PCB** is the **main data structure** the OS uses to keep track of all its processes. It tracks the execution state and location of the process. When a process is created, the OS allocates a new PCB and places it on a state queue. Once the process is terminated, its PCB is deallocated.

The PCB contains but is not limited to: process state, process number (how its identified), program counter, stack pointer, general purpose registers, memory management info, username of the owner, list of open files, queue pointers for state queues, scheduling information, I/O status.

The PCB holds all information we could need about the process itself.

**Process State Queues**

The OS manages **queues for each state** that a process can be in. For example, the Waiting state has its own queue called the wait queue and the **OS uses the PCBs** of each process to manage them. When the state of the process is changed, it is moved to the queue of the next, corresponding state. Each **I/O device has it’s own** dedicated queue in wait.

When the PCBs are in a queue, they have the address for the PCB in front of and behind it in the queue. The queue itself just has a **head and tail pointer** for the address of the first and last PCBs.

When we switch states, this is called a **context switch**. Context switches are very expensive operations. We have to save all of the values of the registers the process is using to the PCB and then load all the values from the next PCB before we can start running it. There is a balance between switching a lot and spreading out the CPUs time and wasting time saving and loading all the information from the PCB.

**How do we create processes**

Every process is **created by another process**. This means that every process has a **parent** process that it is a **child** of. This means processes can be visualized as a tree as they parent processes create children that can also create children. Parent processes **allocate resources and privileges** to its children. A parent can either **wait** for a child to complete or **continue in parallel**.

When you click google chrome on your computer, the GUI is actually what creates the process for opening chrome. The shell window can also create this process if you were to use the terminal to open the program. Obviously, there is some process that is process 0 and doesn’t have a parent but only one.

In Unix, we use the **Fork** command to create a new process. When we do this, we create an **exact copy** of the parent process. The only difference between the parent and child is the **ID numbers of the parent and child**. Everything else is the exact same.

When you call fork(), the new process starts **after the fork call**. This is why it doesn’t get stuck going infinitely. Processes are created **2^n times** where n is the number of fork() that is called. This means if we call fork() 3 times, we will run the program 8 times (2^3 = 8).

Processes created with fork() are **run concurrently**. That is why we typically issue a wait command, this makes the parent wait for the child to finish first. You’ll need a wait call for every fork you use to make the parent, for sure, finish last.

If the parent is terminated before the child is done, than the child becomes an **orphaned process.** It still needs a parent so it is assigned to the root. Typically PID 1. Parent processes wait until their child process is done before continuing but it is still possible to destroy a parent before the child is done.

The **fork()** command returns the **child ID number for the created process** for the parent and the **child sees a 0 returned**.

When we use the fork command, we usually are using the exec() command for the child. This replaces the child with whatever program you give it to run. The child ceases to exist and the program you told it to execute takes its place. It is called **execlp(**program name**)** now and it **replaces the current process with the named program**. When you run execlp(), the process is immediately killed after so even if you have code that comes after that, it **will not execute** for the child.

We have to be very careful when using fork because if we make them in a while loop, we can infinitely make new processes.

To stop the parent from running at the same time as the child, we can use **waitpid(**id number**)** which waits on a process with the given id number to finish first. This is why fork() return the child id, so we have it and can make the parent wait.

**How to terminate a process**

We can use the **kill(**id number**)** to kill any child process to the one that is executing this command. Thats how we can kill it from another process. We can also use the **exit()** command to end a process within that processes. There are limits on which processes you can kill, like you can’t kill the root.

**Cooperating Proceses**

Any two processes are either **independent** or **cooperating**. This can improve efficiency if we wanted to divide up work between to processes. We would need to share data between these processes though and we can do that with either **message passing** or **shared memory**.

**Message passing**

We have one process that uses the **send**(data, process 2) and the other uses **receive**(data, process 1). We can use waits to make each process wait to either send or receive until the other is ready.

**Shared Memory**

Shared memory works like you think it would. We reserve a block of memory that both processes have access to. This is when synchronicity becomes really important, we don’t want to be reading and writing data at the same time or reading the data at the wrong time. This is typically done using **mmap().**