**Chapter 3: A dialogue on Virtualization**

Virtualization in an OS is like having only one peach but many eaters, so we give each eater a virtual peach. They seem to have a peach each person, but in reality, they don’t.

Similarly, a system only has one CPU. **Virtualization takes that single CPU and make it looks like many virtual CPUs** to the applications running on the system so that they think they all have their own CPU to use

**Chapter 4: The process**

**The process** is one of the most fundamental **abstractions** that the OS provides to users.

The program itself is a lifeless thing that lives on the disk, have a bunch of instructions and wait to spring into action. The OS runs the program and transform it into something useful. The challenge is that we often want more than one program to be run at once, as one never need to be concerned with whether a CPU is available. The OS solves this by **virtualizing the CPU**.

By running one process, then stopping it and running another, and so forth, the OS can promote the illusion that many virtual CPUs exist when in fact there is only one physical CPU (or a few). This technique is known as **time sharing**. This allows us to run as many concurrent processes as we like, but the potential cost is performance. **Time sharing** is a basic technique used by an OS to share a resource by allowing the resource to be used for a little while by one entity and then a little while by another, etc. Its counterpart is **space sharing** as a resource is divided among those who wish to use it.

To implement virtualization of the CPU, we need both low-level machinery (**mechanisms**) that implement a needed piece of functionality and high-level intelligence (**policies**) that makes **decisions** within the OS.

**4.1 The abstraction: A Process**

**The process** is simply a running program. At any instant of time, we can summarize a process by taking an inventory of the different pieces of the system it accesses or affects during the course of its execution.

**Machine state**: what a program can read or update when it is running? What parts of the machine are important to the execution of this program?

Components of machine state that comprises a process are memory and registers (program counter, stack point and frame pointer). Programs often access persistent storage devices too.

**4.2: Process API**

APIs that are available on any modern operating system:

1. Create: the ability of OS to create new processes.
2. Destroy: the ability to destroy process forcefully.
3. Wait: it is useful wait for a process to stop running.
4. Miscellaneous Control: Other controls. For example, OS provides some kind of method to suspend a process (stop a process for a while) and the resume it.
5. Status: get some status information about a process, such as how long it has run or what state it is in.

**4.3: Process Creation: A Little More Detail**

How programs are transformed into processes? How does OS get a program up and running? How does process creation actually work?

Diagram

Description automatically generated

1. OS loads its code and any static data (initialized variables) into memory, into the address space of the process. This is because programs initially reside on **disk** or SSDs in some kind of **executable format**. Therefore, the loading requires the OS to read the bytes from disk and place them in memory somewhere. In early or simple operating system, the loading process is done **eagerly** (all at once before running the program), while modern OSes perform the process **lazily** (only load pieces of code or data as they are needed during program execution).
2. After the loading steps, some memory must be allocated for the program’s **run-time stack**. The OS allocates this memory and gives it to the process. OS also initialize the stack with arguments, i.e. parameters to the main() function (argc and argv). Note: C programs use the stack for local variables, function parameters, and return addresses
3. OS may also allocate some memory for the program’s heap. In C programs, the heap is used for explicitly requested dynamically-allocated data; programs request such space by calling malloc() and free it explicitly by calling free(). The heap is needed for data structures. Malloc() is used to request more memory. OS may be involved and allocate more memory to the process to help satisfy such call
4. OS will also do some initialization tasks, particularly related to I/O

* OS now set the stage for program execution. Its last task is to start the program running at the entry point (main()). By jumping to the main() routine, the OS transfers control of the CPU to the newly-created process and the program’s execution begins.

**4.4 Process States**

Diagram

Description automatically generated

3 states of a process:

1. Running: in this state, a process is running on a processor. This means it is executing instructions.
2. Ready: a process is ready to run, but for some reason, the OS has chosen not to run it at this given moment.
3. Blocked: In this state, a process has performed some kind of operation that makes it not read to run until some other event takes place. For example: when a process initiates and I/O request to a disk, it becomes blocked and some other processes can use the processor.

Example of two processes without I/O:

Table

Description automatically generated

Example of two processes with I/O:

Text, table

Description automatically generated

OS must make decisions (by OS scheduler), even in the above two examples.

**4.5 Data Structures:**

**The OS is a program** and it has some key data structures that track various relevant pieces of information. For example, to track the state of each process, the OS is likely to keep some kind of **process list (task list)** for all processes that are ready and some additional information to keep track which process is running. The OS must also keep track of blocked process. Each entry is found in what is sometimes called a **process control block (PCB),** which is really just a structure that contains information about a specific process.

Xv6 kernel:

Text

Description automatically generated with medium confidence

The register context (struct context) will hold the content of a stopped process’s register. When a process is stopped, its register will be saved to this memory location and the OS can resume running the process. This technique is called context switch.

It also has a number of states that a process can be in (other than just ready, running and blocked). Sometimes, we have **initial** state that the process is in when created. A process could be placed in a **final** state where it has exited but not yet been cleaned up (sometimes called **zombie** state). The final state can be useful as it allows other processes to examine the return code of the process and see if the just-finished process executed successfully. When finished, the parent will make one final call (e.g., wait()) to wait for the completion of the child, and to also indicate to the OS that it can clean up any relevant data structures that referred to the now-extinct process