Chapter 3 – The Programming Interface:

# 3.1 Process Management

A shell is a job control system. MacOS, windows and Unix all have shell. With a shell, you can write down a sequence of steps, as a sequence of programs to run to do each step. It’s basically a way of manage the operating system by writing a set of code to the shell.

**Windows Process Management:**

In windows, there is a routine called CreateProcess to, obviously, create a process. We call the process creator the *parent* and the process being created the *child*.

What steps does the CreateProcess take to create a process?

* Create and initialize the process control block
* Create and initialize a new address space
* Load the program into the address space
* Copy arguments into memory in the address space
* Initialize the hardware context to start execution at “start”
* Inform the scheduler that a new process is ready to run

On important thing to mention is that there are quite a few aspects of the process that the parent might like to control; such as its privileges etc.

**Unix process management:**

The UNIX process management is slightly different than the windows one. UNIX splits createProcess into two steps, called fork and evec.

*UNIX Fork* creates a complete copy of the parent process. There is just one slight exception. We need to establish which is the parent and which is the child. Since the child runs exactly the same code as the parent we can trust the child to set up the context for the new program correctly.

Once the context (privileges, priorities, IO etc) is set the child process calls UNIX exec. UNIX exec is a system call changing the program being run by the current process. UNIX exec then brings the new executable image into memory and it starts running. Then we have swapped the process we have created with the fork, and starts running that.

UNIX fork takes no arguments, and returns an integer. On the other hand, UNIX evec takes two arguments (the name of the program to run and an array of arguments to pass to the program). The difference between the windows createProcess and the fork/exec is that the UNIX exec only takes those two parameters, while the windows takes 10.

The steps for implementing UNIX fork in the kernel are:

* Create and initialize the process control block (PCB) in the kernel
* Create a new address space
* Initialize the address space with a copy of the entire contents of the address space of the parent
* Inherit the execution context of the parent (e.g., any open files)
* Inform the scheduler that the new process is ready to run

The UNIX Fork returns twice; once to the parent and once to the child. To the parent, UNIX returns the process ID (PID) of the child, to the child, it returns zero indicating success.

So the fork() returns two integers; the process ID of the child (which is zero) and the process ID of the parent.

**UNIX Evec and wait:**

The UNIX system call evec completes the steps needed to start running a new program. The child process typically calls the evec once it has returned from UNIX fork and configured the execution environment for the new process.

UNIX evec does the following steps:

* Load the program into the current address space
* Copy arguments into memory in the address space
* Initialize the hardware context to start execution at “start”.

The UNIX exec does not create a new process.

UNIX has a system call, naturally enough called wait, that pauses the parent until the child finishes, crashes, or is terminated. The wait system call is parameterized with the process id of the child.

**3.2 Input/output:**

All computers have the possibility to connect to many different I/O devices. How should the OS deal with all of those? We could specialize the application programming interface for each device. However, early computer systems took the approach of specializing the interface to the device, but it had a significant downside: every time a new type of hardware device is invented the system call interface had to be upgraded to handle that device.

UNIX had several primary innovations; one of them was to regularize all device input and output behind a single common interface. It uses this same interface for reading and writing files and for interprocess communication.

The basic ideas in the UNIX I/O interface are:

* Uniformity: All device I/O, file operations and interprocess communication use the same set of system calls, open, close, read and write.
* Open before use. Before an application does I/O, it must use the system call open. This gives the OS the chance to check access permissions and set up any internal bookkeeping.
  + Open returns an file descriptor which can be used in later calls to read, write and close.
* Byte-oriented. All devices are accesed with byte arrays.
* Kernel-buffered reads. Stream data, is stored in a kernel buffer and returned to the application on request.
* Kernel-buffered writes. Outgoing data is stores in a kernel buffer.
* Explicit close. When the applications is done with the device it calls close.