**CMPSCI 577 Operating Systems Design and Implementation** **Spring 2020**

# Lecture 3: 01/28

*Lecturer:* ***Prashant Shenoy*** *Scribe:* ***Vinita Hissaria***

## 3.1 Summary

In this class we briefly covered the origins of Unix, Linux and Minix. We went over why the microkernel architecture makes sense, specifically because of the problems caused in an OS due to device drivers. We also go over some features of Minix, which is the focus of this class.

## 3.2 Brief History

In 1964, researchers at Bell Labs, MIT and GE created a time-sharing system, MULTICS, which was meant to be an improvement over previous OSes. However, it turned out to be too complex and was a software engineering failure. However, this led to further development in operating systems.

After MULTICS, the developers implemented a basic, stripped down version of MULTICS on a PDP-7, called UNICS. This OS was then re-implemented on newer machines in C and called Unix. It was open source, and was taught in universities, where students could make changes to the source code. However, AT&T licensed this version, which meant that it could not be taught anymore. In order to remedy this, free BSD was created, which could be modified and distributed. Most versions of Unix are modifications of BSD.

### 3.2.1 Minix

Tannenbaum created a smaller, compact version of Unix, called Minix(Mini Unix), for PCs. It ran on very less memory and was designed primarily for educational purposes. The initial version of Minix was unstable and would crash every time it was booted, which was later attributed to a fault in the 8088 processor which would send an interrupt on being overheated and caused the process to shut down. Once this interrupt was ignored, Minix could be used. This led to the development of Linux.

### 3.2.2 Linux

Linus Torvalds understood the Minix code and wrote the Linux OS, as a hobby. However now Linux is widely used everywhere. Linux is a direct descendent of Minix, and still contains some code from Minix.

## 3.3 Why Microkernels?

### 3.3.1 Why do OSes crash?

Most operating systems use a monolithic design, in which the different modules are not separate. So any bug causes the OS to crash. The bigger the code base, higher the probability of bugs. The OS code is made bigger by device driver code. It was found that most of the crashes were caused by buggy driver code, since they probably did not follow the same coding standards as OS developers.

### 3.3.2 Reliability of different OS architectures

Monolithic architecture is least reliable, since device drivers are part of privileged code, and any bug in a driver causes the whole OS process to crash. Comparatively, microkernels are more more reliable, since device drivers are outside the kernel and a bug only causes that driver process to crash, and not the entire OS. The reliability of a hybrid kernel is based on whether the buggy driver is in the kernel or not.

### 3.3.3 Takeaway

Thus, there is merit to the idea of microkernels because moving the device driver code outside the kernel reduces the OS crashes and increases reliability of the OS.

## 3.4 Improvements in Monolithic Architecture

In order to not make drastic design changes to the existing monolithic architecture, developers tried to make small changes to improve monolithic architecture and make it more stable. These efforts were aimed at providing reliability without discarding existing code.

### 3.4.1 Nooks

Nooks were wrappers for device drivers introduced in Unix. The device drivers still remained in the kernel, but the wrapper examined all requests,function calls going in and out of the device driver. Any problematic operation was not allowed, and even the driver did not crash. It was a means of exception handling without affecting the rest of the OS. In addition, when the driver code was running, it made the rest of the kernel read-only such that the driver code could not change anything in the kernel.

Nooks avoided major code re-design. However, this approach only limited the harm that could be caused by the driver, did not completely eliminate it.

### 3.4.2 Virtualization

In this approach, the OS was run on one virtual machine and the device driver on another. So, in case the driver had a bug the VM would fail, the OS would still run on the other VM without interruptions.

### 3.4.3 L4 Microkernel

This was an attempt to change Linux to microkernel architecture. The Linux OS was run as a process on top of an L4 microkernel(acts as hypervisor), with some of its functionality separated from it and run as separate processes. The complete separation of the OS into different parts could not be done because it required too much restructuring and it ended up being a hybrid approach. The drivers still reside in the Linux kernel in this approach If there is the bug, the OS process shuts down and can be restarted, instead of the whole kernel crashing.

## 3.5 Singularity

This is another approach at creating reliable operating systems. The presence of bugs in OSes was attributed to writing kernels in languages like C which were not type safe. So, the solution was to use a new language called sing# which would not offer as much flexibility and power as C.

Also, all user processes and the OS would be part of the same virtual address space. Since the compiler had proper ownership and type checks, no process would be allowed unnecessary privilege and access the resources of another process. Hence, very simple or no context switching was required. However, this also meant that modules such as drivers could not just be plugged in. They had to be compiled, verified and could then be used.

Singularity followed a microkernel approach where each driver ran as a separate process, just in the same address space.

## 3.6 Microkernel Architecture - Minix

In this architecture, the kernel has few important functionalities in it, and all the others run as servers outside it. The kernel takes care of IPC, memory management, scheduling and interrupts. This leads to reduced performance because the different processes cannot use function calls, but have to communicate through IPCs which are more expensive. However, reliability is increased.

Minix is a multiserver OS. It has a microkernel, with three components - SYS, IPC and Clock. Clock is used for timer functions and SYS communicates with the processes outside the kernel. Some processes are implemented as servers, like the file server and reincarnation server.

### 3.6.1 IPC in Minix

In Minix, IPC is synchronous. Also, all processes have different privileges, which are enforced by the kernel. Almost all processes run outside the kernel, but system processes have higher privilege and can make calls to the kernel, whereas user processes make calls to the system processes.