CMPSCI 630 Systems **Fall 2019**

Lecture 11

Lecturer: Emery Berger *Scribe: Bryce Bodley-Gomes, Bochen Xu, Wenjun Huang*

11.1 Eunices

Multics introduced security rings. Ring 0 = supervisor mode; ring 3 = user mode.

Software can have bugs. Having hardware enforce security was a good idea, as there tend to be less bugs in hardware compared to software implementations. But new attacks like Spectre show that hardware security is not a solved problem.

Multics was a huge failure. The actual motivation behind Unix was to make portable video games.

Unix was expensive, so people at Berkeley wrote a Unix clone (BSD) without looking at the source code. This in turn led to other operating systems. None of the OSs directly based their code off of their predecessors but they definitely adopted similar feature design.

Unix's primary adaption from Multics is the file system. On Unix-like systems, everything is a file (i.e. the "one level of indirection" philosophy), including devices (e.g. hard drives). Even processes are represented as files (e.g. Linux's /proc, which came from Plan 9). Files can be block-based (IO done in blocks) or character-based (IO done in bytes).

Unix also introduced mountable file system (without letters that DOS reintroduced, which caused portability issues). Mountable file system is again following the "one level of indirection" philosophy: you can refer to remote file systems using regular paths (e.g. /mounted/file1).

On Unix-like systems, it is possible to redirect output to files/programs in the shell. Since all devices are files, this allows you to do thing like printing program outputs directly in the shell.

Shell pipeline has built-in parallelism ("streaming"): as the first program starts generating output, a second program is often able to begin processing this output before the first program finishes execution.

The Unix philosophy is to have many small utilities and combine them using pipelines. But this is a bad idea in practice, as each program invocation creates a new process, and the overhead is high.

11.2 Languages are Cults

This overhead issue led to Perl, which conceptually packages many Unix utilities into one language. But Perl has weird syntax, making it "write-only", so Python came along and solved that problem.

Ruby didn't solve any problem. But Rails, being the first decent web framework, gave people a reason to use Ruby (i.e. Ruby has a killer app).

People weren't happy with Miranda^{TM}, so Haskell came along. It is a pure lazy functional language. Here, pure means functions can't have any side effect. It's hard to code in Haskell, so there aren't many large programs written in it.

Lazy evaluation: do work only when it's needed. So a self-referencing data structure is possible: ones = 1 :: ones, and take(ones, 3) = [1, 1, 1]. However, lazy evaluation makes programs slow, as each time a value is needed, we have to check if it's been evaluated (i.e. if(evaluated(x)) proceed else evaluate(x); proceed).

Currying: passing some arguments to a function creates a new function with those arguments filled in. E.g. f(x, y, z)(1, 2) = f(z) [x = 1, y = 2]. List comprehension (as seen in Python) and currying were adopted by many other languages (originally they were Haskell features).

11.3 Microsoft v.s. Macrohard

In a monolithic kernel, all functionality implemented by the OS is written in one "blob." That blob includes all the device drivers as well. This makes the trusted computing base (TCB) very large. Also, device drivers have the same privilege as the kernel. Thus, a bad device driver can bring down the whole system. This design violates the "principle of least privilege," which essentially states that you should give each process the minimal amount of access it needs to complete its job.

Microkernel strips out unnecessary components from the kernel (e.g. device drivers) and separate them into individual user-space modules. The kernel is therefore minimal, and device drivers generally can't bring down the whole system. But these modules incur extra overhead, as (1) they need to communicate with the kernel (2) they need to switch into kernel mode occasionally. As such, microkernel OSs tend to run slow.