CMPSCI 630 Systems **Spring 2023**

Lecture 9

Lecturer: Emery Berger *Scribe: Sanath Upadhya*

9.1 History of UNIX

A brief history and impact of UNIX

- One of the most successful OS created
- UNIX inspired many of the current OS - foe example OSX, MINIX, Linux
- It was designed and developed by AT & T Bell Labs (even C and C++ was developed by the same entity)
- Initially developed so that the developers could play video games and thus, UNIX was written so that it could be portable and run on different H/W platforms
- After UNIX became successful, AT & T imposed a licensing fee. University of California Berkeley implemented the Application Binary Interface (ABI) to negate this and developed BSD having most of the functionality of UNIX.
- This ABI of BSD was then used by OSX, Windows etc.
- They also developed a kernel and its conceptual digram is as shown in the figure below

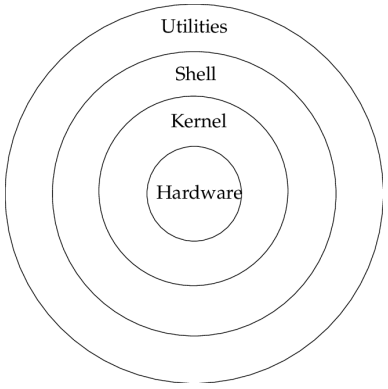

Figure 1: Conceptual diagram of kernel [C1]

- Kernel has access to everything and this runs in privileged mode
- The user programs all run outside the kernel and the basic diagram of the user space and the kernel space is as shown in the figure below

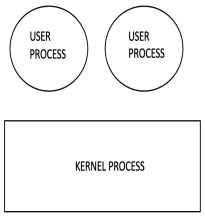

Figure 2: User space and kernel space

9.2 MULTICS

- This OS was designed and developed by MIT, General Electric and Bell labs
- Was written in a high level programming language PL/I
- Was designed to run on multi core processors and was designed such that any processors could be removed while the OS was still running
- As a result, it turned the OS problem into a distributed systems problem and thus was very complex
- A lot of the features that was seen in UNIX was borrowed from MULTICS, and thus, UNIX was a limited version of MULTICS (UNIX was developed by former MULTICS programmers)

9.3 Features borrowed from MULTICS

- Protected access and the privileges was borrowed from MULTICS
- Symbolic links
- Hierarchical file system
- Writing an OS in a high level language
- Security rings, i.e. privileged access (see conceptual diagram of kernel), with kernel being given the ring 0, ring 1 to the device drivers and ring 3 to the user applications. This privilege ring is baked into H/W
- Command line or shell was also borrowed from MULTICS

9.4 Device drivers

- Device drivers usually reside in the kernel. These are the programs that directly interact with the H/W.
- They offer some level of abstraction to the processes above that use these device drivers

- A device driver has to be written for every H/W and for every OS that supports this H/W (thus, an explosion of device drivers seen)
- To minimize this, the USB was created - one device driver that can handle multiple H/W devices. USBs have a DOS file system when you initially plug in
- Since the device drivers reside in kernel, a lot of security issues might be exposed if the device driver does not behave properly
- If a device driver overwrites memory in the kernel, we see kernel panic (and in Windows OS results in Windows Blue Screed of Death)
- To prevent this, MSFT had to do a lot of static checking of device driver code so that no untoward incidents happen
- They also decided to sign the device drivers, saying which are safe and which are not
- The fundamental question that arose out of this was whether the device drivers should reside in kernel or should be a part of the user space
- This lead to the development of micro-kernels, wherein only a small amount of functionality was kept in the kernel and everything else was moved out to the user space
- The corollary situation was the monolithic kernel, where all the functionality resided in the kernel itself
- Micro-kernel
  - This was first instantiated by Any Tannebaum in MINIX (which is nothing but UNIX with a micro-kernel)
  - Deployed on the H/W i386 (early version of x86 of Intel Thinking was, even if the device driver crashes, the overall system remains safe and one would just have to restart the device driver in such a scenario
  - However, the idea of micro-kernel did not take off due to performance issues
  - We see that, every time there is transfer of control from user space to kernel space (i.e. context switching), there is a lot of copy operations performed (to save the state, get data from hard drive etc.). This is obviously much slower than a monolithic kernel, where this would just be a function call
  - There is also the problem of caches being "warm" (i.e. all relevant data might already be in the cache), and when we do context switching, this all has to be flushed and has to be copied again, resulting in a performance hit
- Thus, we see that monolithic kernels are the ones that are preferred in today's operating system
- Context switching is also used when different applications share the same CPU resource. This is also called as time slicing of programs
- Every program is allowed a certain quanta (one quanta is around 5ms, which is reasonably high), so that every program does some reasonable progress whenever it gets the CPU
- However, the idea of micro-kernel is still valid and this comes to TCB (trusted computing base)
- This means that find a small base where the trust resides (smaller the base, easier it is to analyse and detect security risks)
- Intel also has something like this called the SGX (secure enclaves) and they also have TEEC (trusted execution environments)

9.5 UNIX file system

- UNIX took the concept of file abstraction (where everything is a file
- We also see that standard I/O are treated as files and ¡ and ¿ were introduced in UNIX
- Write to a file is a ¿ foo.out
- Read from a file is a ¡ in
- They also introduced the mounted file system which enabled the remote file system that we see today (like NFS)
- There is only one level of indirection - we can move the mounted file system anywhere to any directory in our local FS
- They also introduced the concept of i-nodes and direct/indirect addressing of files (as shown in figure below)

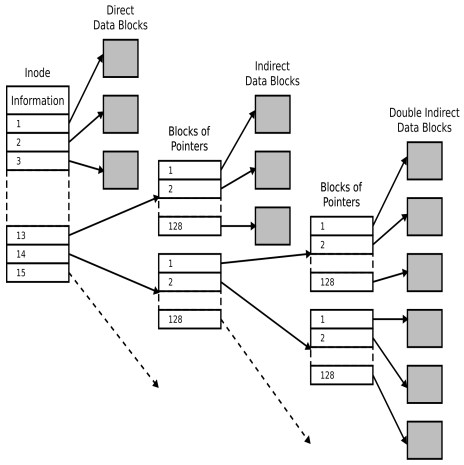

Figure 3: Direct and indirect addressing [C2]

- Thus, there is no limit on the size that a file can be in this FS (as long as you have space in hard drive)

9.6 Miscellaneous discussions

- Current language suitable for OS
  - Of the current languages, RUST can be used for writing an OS
  - GO might not be suitable as this has garbage collection, which might not be good for writing an OS
  - There is a concept called pinning, wherein objects are to be statically located in the memory, and if there is garbage collection, then everything must go through this (due to pinning and unpinning), performance is hugely impacted

  - The pinning/unpinning is also the reason why Python is better suited for machine learning algorithms. Since all ML APIs are written in C, C++, programs written in Java have to go through JNI (Java Native Interface) and this does a lot of pinning/unpinning resulting in performance decrease

- Philosophy of UNIX - everything just does one thing, thus the concept of piping in CLI took place
- This lead to the development of min languages like awk, regular expressions like grep, sed etc.
- These lead to the development of Perl (which had awk, sed, grep), which in turn lead to the development of Python