COMS E6998 UNIX Reflection

Avery Blanchard,[1, *] Angela Peng,[1, †] and Shuo Liu[1, ‡]
[1] *Columbia University*

With the publication of Dennis M. Ritchie and Ken Thompson's 1974 paper, The UNIX Time-Sharing System,[1] the field of computer science changed. To quote the selection committee that awarded Ritchie and Thompson the Turing award in 1983, "The genius of the Unix system is its framework, which enables programmers to stand on the work of others." [2] UNIX provided an elegant solution that led to enormous developments in computer hardware, software, networking, and security. [3] The UNIX Time-Sharing System paper is an undeniable classic with immense impact. In this paper, we explore the work presented by Ritchie and Thompson over fifty years ago and illustrate how UNIX shaped the field of computer science.

I. INTRODUCTION

In 1974, Dennis Ritchie and Ken Thompson published the paper "The UNIX Time-Sharing System". [1] The ideas presented in the paper are ubiquitous in modern systems and programming philosophies. In the paper, Ritchie and Thompson describe the central ideas of UNIX – the file system and the shell. The ideas presented by Dennis Ritchie and Ken Thompson through the development of UNIX are ubiquitous in modern systems and programming philosophies. Following our presentation of UNIX as a classic work in computer science, we write this paper as a reflection on UNIX's core ideas and lasting impact.

II. ABOUT THE AUTHORS

Dennis Ritchie and Ken Thompson published UNIX Time-Sharing System[1] in 1974. The two worked together on numerous projects throughout their careers at Bell Labs. In 1983, Ritchie and Thompson won the Turing Award 1983 for their work on UNIX.[2] Together, they also were awarded the Hamming Medal in the 1990s[2], the National Medal of Technology[4], and the Japan Prize for Information and Communication in May 2011[2]. Ritchie and Thompson are prolific within computer science. In this section, we explore their background and accolades.

A. Dennis M. Ritchie

Dennis Ritchie was born in New York in September 1941.[2] He received his bachelor's degree in physics and graduate degrees in applied mathematics from Harvard University in 1963 and 1967.[2] During his time at Harvard, Ritchie proclaimed: "My undergraduate experience convinced me that I was not smart enough to be a physicist and that computers were quite neat. My graduate school experience convinced me that I was not smart enough to be an expert in the theory of algorithms and also that I liked procedural languages better than functional ones."[5] During his graduate studies, Dennis Ritchie began working part-time at Bell Labs on the MULTICS project in 1967. And in 1968 he defended his dissertation, "Computational Complexity and Program Structure"[6], advised by Patrick C. Fischer.

Dennis Ritchie joined Bell Labs full-time after completing his graduate studies. [2] During his time there, Ritchie worked on numerous projects, such as MULTICS, B, BCPL, C, UNIX, and ALTRAN. [2] Ritchie co-authored "The C Programming Language Book" with Brian Kernighan – which remains a central part of modern computer science education.

Dennis Ritchie died on October 12, 2011, at 70 years old. [5] As historian Paul Ceruzzi stated on his death, "His name was not a household name at all, but... if you had a microscope and could look in a computer, you'd see his work everywhere inside." [4]

B. Ken Thompson

Ken Thompson was born in Louisiana on February 4th, 1943. [7] Thompson received his bachelor's and master's degrees in computer science and electrical engineering from the University of California, Berkeley in 1965 and 1966. [7] In 1966, Thompson started working at Bell Labs. Thompson is known for his work on MULTICS, UNIX, B, a chess machine named Belle, UTF-8, Plan 9, Inferno, grep, endgame tablebase (a database of chess endgame positions), and the Go programming language.[7]

* agb2178@columbia.edu
† ap4042@columbia.edu
‡ sl4921@columbia.edu

FIG. 1. Ken Thompson and Dennis Ritchie (seated) working at the PDP-11

III. BACKGROUND

In this section we will briefly define the role of an operating system and explore a brief history of operating systems.

A. Operating System Basics

An Operating System provides an interface between hardware and software. The purpose of the system is to control a machine's hardware and logical resources. [8] The development of modern operating systems would shift the way computers were used.

B. A Brief History of Operating Systems

In the 1960s, Bell Labs introduced the multiplexed information and computing service (MULTICS) as a time-sharing operating system with MIT and General Electric. [9] MULTICS's design included a supervisor program that managed hardware resources (using symmetric multiprocessing multiprogramming and paging), a segmented memory addressing system supported by the hardware, a tree-structured file system, device support, 100s of command programs, 100s of user callable library routines, operational and support tools, and documentation. [9] MULTICS combined ideas from other operating systems with the inclusion of some innovations. [9] It aimed to change the way people used and worked with computers. [9] Bell Labs withdrew from the project in 1969. [10]

In the 1970s, Bell Labs developed UNIX as an alternative to MULTICS. This effort was led by Dennis Ritchie and Ken Thompson. Apple's disk operating system, Apple DOS, was also released in 1988 for the Apple II. [11] The operating system was quickly discontinued in 1983. cite[11]

In the 1980s, the Microsoft DOS and NeXTStep operating systems were introduced. Microsoft bought 86-DOS from its creator Tim Patterson for seventy-five thousand dollars in 1986. [12] 86-DOS was renamed as MSDOS and within the year Microsoft licensed this software to over seventy companies. [12] NeXTStep is a UNIX-based operating system that was released in the late 1980s. [13]

In the following decade, Linus Torvalds released LINUX. LINUX is a free and open-source operating system based on a descendant of UNIX as a master's student at the University of Helsinki. [14] Microsoft also released Windows 3.0 to replace MSDOS. [12]

In the 2000s, Apple's UNIX-based operating system, MacOS, was released.

Over the decades since UNIX, there were numerous operating systems released. Many of these systems were based on the innovative core ideas presented by Ritchie and Thompson in 1974.

IV. UNIX ORIGINS

After the Bell Labs decision in 1969 to leave the MULTICS project, Dennis Ritchie and Ken Thompson began looking for alternatives. During this time, Ritchie and Thompson submitted unsuccessful several proposals.[10] However, in 1969, Ritchie and Thompson designed the UNIX file system.[10]

A. UNIX and the C Programming Language

The C programming language was designed in the early 1970s in parallel with UNIX. MULTICS was written in the B programming language. UNIX and C were designed in parallel to address the problems observed when using B. [15] In the implementation of B there are no types, everything was a word. [15] Other problems included the single datatype of the hardware machine word, the lack of floating point arithmetic, and the way pointers were handled.[15] In B, each pointer generated a run-time scale conversion from the pointer to the byte address expected by the hardware.[15] UNIX was the most central factor in the success of the C language and C allowed for UNIX's hardware independence and probability.[3]

B. UNIX and Hardware

As UNIX was written in C rather than assembly, it was a hardware-independent operating system. This allowed for UNIX's portability and accessibility, which ultimately led to its prominence. [3] UNIX was an OS for inexpensive hardware.

UNIX originated from Bell Labs' need to develop an alternative to MULTICS. Ritchie and Thompson aimed to innovate upon the goals of MULTICS. The C programming language was developed to address the shortcomings of B. C enabled UNIX's hardware independence. The origins of UNIX vastly impacted computer science. The rest of the paper will describe the central ideas of the 1974 paper authored by Ritchie and Thompson and their impact on computer science.

V. UNIX FILE SYSTEM

A. Features

The Unix file system boasts numerous beneficial features and advantages, distinguishing it from other operating systems during that era. For example, Table. I provides a comparison that shows the major difference between the file system of Unix and Windows OS.

Feature	Unix	Windows
File naming	Case-sensitive	Case-insensitive
Directory structure	Hierarchical	Tree
File permissions	Granular	More granular
Path separators	/	\
Links	Symbolic links	Symbolic links&joint points
Drivers	Normal	Wider-range

TABLE I. Comparison of Unix and Windows file systems

File Types

The Unix file system incorporates 3 fundamental file types:

1. *Ordinary files*, the most common file type found on Linux systems, hold text, data, or program instructions. They encompass a wide variety of formats, like text files, image files, compressed files, and binary files.
2. *Special files* also exist within the Unix file system:
   (a) *Device files*, known as "blocks," provide buffered access to system hardware components, while "character files" offer unbuffered serial access to the same components. Character files transfer data one character at a time, whereas block files can transmit large data chunks simultaneously.
   (b) A *symbolic link* serves as a pointer to another file or directory within the system, which can be either a regular file or a directory.
   (c) *Pipes* are file-based channels that enable inter-process communication by connecting the output of one process to the input of another. Conversely, *sockets* facilitate data and information exchange between processes operating on different machines across a network.
3. *Directories* represent rooted tree structures in the file system that house ordinary files and special files.

Detachability

One of the key attributes of the Unix file system is its detachability. Although the root of the file system is consistently stored on a specific device, the entire file system hierarchy is not required to reside on that same device. For example, the "mount" command can reroute references from a standard file to the root directory of the file system on a detachable volume.

Several benefits arise from having a detachable file system. First, it allows for the storage of more data using portable and easily transportable media. Removable media is often faster and more convenient to transfer than internal hard drives, as it eliminates the need for cables or connections between devices. Furthermore, detachable file systems generally do not necessitate specialized software or drivers, making them ideal for sharing files across multiple devices.

Reliability

An important aspect of the Unix file system is its reliability. Each user on the system receives a unique user ID, which is assigned to any file they create. Unix allocates 7 protection bits to each new file, with 6 of these bits independently controlling read, write, and execute permissions for both the file's owner and all other users. The last one functions as an indicator; when enabled, it temporarily changes the creator ID to the current user. This user ID modification is effective only during the execution of the program that necessitates it. The set-user-ID feature enables privileged programs to prevent unauthorized access by non-superusers.

The MOO accounting problem, inspired by the MOO game, illustrates that the Unix file system protects files by distinguishing between superusers and ordinary users. In this game, users can only modify their score files by playing the game or executing the relevant program, thereby preventing ordinary users from tampering with specific files. Furthermore, the set-user-ID mechanism provides flexibility by allowing users to execute carefully crafted commands that require privileged system access. A prime example of this is the widely used 'sudo' command.

Inodes

Inodes are a fundamental aspect of the Unix file system, offering a distinct method to oversee and access files. As a critical data structure, they store vital information about files, including size, permissions, ownership, and data block locations on the storage device. Through inodes, Unix facilitates efficient insertion and removal of storage units, as well as versatile file management.

Upon creating a file in Unix, an inode is assigned to store its metadata. The inode id, a unique identifier, enables the file system to locate and access the file's data blocks on the storage device. This indirect file referencing through inodes, rather than directly by their names, presents numerous benefits. It supports hard links, which allow multiple filenames to reference the same inode and share identical data, and streamlines renaming or relocating files within the file system.

Regarding the insertion and removal of storage units, Unix's inode-focused approach offers significant flexibility. To insert a new storage unit, the file system simply allocates new inodes and data blocks on the added device. In contrast, to remove a storage unit, the file system must ensure that the associated inodes and data blocks are transferred to another storage unit if needed. This method enables Unix to effectively manage storage resources while preserving the file system's overall integrity.

B. I/O Calls and Locking

In Unix, the file system does not enforce any visible user locks or limitations on the number of users accessing a file for reading or writing. Unix's creators argue that traditional locking mechanisms are inadequate for preventing interference among users accessing the same file. While large, single-file databases managed by independent processes are rare, standard locks fail to prevent confusion when multiple users edit a file using an editor that creates a file copy.

Various Unix variants offer distinct file-locking mechanisms, with fcntl being the most widespread. Using fcntl, diverse lock types can be applied to specific sections or the entire file. Multiple processes may hold shared locks, while only one process can possess an exclusive lock, which is not able to exist with a shared lock together. In order to obtain a shared lock, one process must be frozen until no processes maintain an exclusive lock. On the contrary, the process must also wait until there are no processes that hold either lock type to secure an exclusive lock.

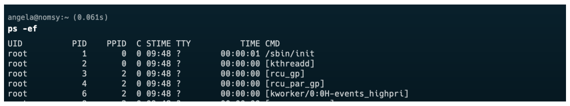
Locks generated by flock persist across forks, making them advantageous in forking servers. Consequently, multiple processes can maintain an exclusive lock on the same file if they share a filial connection, and the exclusive lock was originally established in one process before being replicated by a fork.

Nonetheless, failures may still arise when reading and writing occur sequentially. If the read pointer is too near the file's end, such that reading a specific number of characters surpasses the end, only enough bytes will be transferred to reach the file's end. Moreover, typewriter-like devices will never return more than a single line of input.

VI. UNIX PROCESS AND SHELL

A. Process

A process is an instance of a running program that has been loaded into memory and is being executed by the computer's CPU. Every process is assigned a unique process ID (PID) that identifies it within the system. UNIX treats each process as an independent entity and each process has its own address space, which is used for storing important information about the code, the variables used in the program, and other important data needed to run a process smoothly. As you can see from the images



```
angela@nomsy:~ (0.061s)
ps -ef
UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  0 09:48 ?        00:00:01 /sbin/init
root           2       0  0 09:48 ?        00:00:00 [kthreadd]
root           3       2  0 09:48 ?        00:00:00 [rcu_gp]
root           4       2  0 09:48 ?        00:00:00 [rcu_par_gp]
root           6       2  0 09:48 ?        00:00:00 [kworker/0:0H-events_highpri]
```

FIG. 2. Running the ps -ef command in the terminal shows info about all the processes running on the computer

the first column shows the UID of the process, this indicates which user the process belongs to. In this case, the root is running the process thus it shows "root" under the column. The UID can also be the current user or processing running on behalf of other applications. The next column is the PID, this as indicated before is unique to every single process running on the computer, and the first-ever process that started will have a PID of 1, and is called the init process. This is the parent of all the future processes, we will talk more about this inheritance relationship in the fork section. The PPID is the PID of the child's parent ID and as you can see the process with PID 2, has a PPID of 1, this means that its parent has a PID of 1. The next column that is important to understand is the CMD column, this indicates the command prompt that started the corresponding process on the computer. There are a lot more attributes associated with a process, but these are the most important and fundamental ones.

B. Images

An image refers to the current state of a computer program or application, which includes information like the program's code, data, and settings. When a program is executed, it becomes a process that runs on the computer, using system resources like memory and CPU time to perform tasks. The image is the starting point for the process, and it contains all the necessary information for the program to run correctly. For example, what time is it when the process is running? What are the processes that are running at this moment? What is CPU usage? etc. The image encapsulates all this essential information for a process to run smoothly. When a program is executed, the image is loaded into memory and becomes a running process. From that point on, the process can modify the image's state as needed, but the original image remains unchanged. In some cases, multiple processes may share the same image, allowing them to run more efficiently and conserve system resources.

C. The fork command

Forking is one of the most important features of UNIX. It is the way for a new process to be born! As we have seen in the previous process section, the process has "memory" of who their parents are, and this is because the fork system call tells the child process who the parent process is (via the PID), and informs the parent process what is the child process created from the system call. More formally, a fork is a system call that creates a new process by duplicating the calling process. When a fork is executed by a process, it splits into two independently executing processes. The two processes have independent copies of the original core image, which includes the code and data of the original process, and share any open files. The new processes differ only in that one is considered the parent process, and the other is the child process. The parent process is the process called the fork system call, while the child process is the new process that is created by the fork. Control returns directly from the fork in the parent process, while control is passed to a specific location (usually the beginning of the program) in the child process. The PID returned by the fork call is the identification of the other process, which is used to distinguish between the parent and child processes. Here is a simple program that demonstrates how the control flow works with the fork() system call

```
#include <stdio.h>
/* for printf() and fprintf() */
#include <stdlib.h>
/* for atoi() and exit() */
#include <string.h>
/* for memset() and strcmp() */
#include <unistd.h>
/* for close() and pipe() */

int main(int argc, char **argv)
{
    fork();
    printf("fork1!\n");
    fork();
    printf("fork2!\n");
    fork();
    printf("fork3!\n");
     return 0;
}
```

And here is the output of the program

```
./hello
fork1!
fork2!
fork1!
fork3!
fork2!
fork2!
fork3!
fork3!
fork2!
fork3!
fork3!
fork3!
fork3!
fork3!
```

At the beginning of the program, there is a call to the 'fork()' system call. This creates a new process, which is a copy of the original process. From this point on, there are two processes running in parallel, both executing the same code. The first 'printf()' statement outputs the string "fork1!". Since both processes execute this statement, the string is output twice.

Then there is another 'fork()' call. This creates two new processes: one child process of the original process, and one child process of the first child process. The original process and the first child process continue to execute the code after the 'fork()' call, while the second child process starts executing at the same point in the code as the first child process did.

The second ‘printf()‘ statement outputs the string ”fork2!”. Since there are now four processes running in parallel, all of them execute this statement, and the string is output four times.

Next, there is a third ‘fork()‘ call. This creates eight new processes: four child processes of the two existing child processes, and four grandchild processes of the original process. The original process and the first child process continue to execute the code after the ‘fork()‘ call, while the second child process and the four new child processes start executing at the same point in the code as the first child process did.

The third ‘printf()‘ statement outputs the string ”fork3!”. Since there are now eight processes running in parallel, all of them execute this statement, and the string is output eight times.

The output of the program is determined by the number of processes that are created by the ‘fork()‘ calls. Each ‘fork()‘ call doubles the number of processes, so after three ‘fork()‘ calls there are eight processes running in parallel. Each process executes all the ‘printf()‘ statements in the code, so the output consists of multiple copies of each string. The exact order in which the strings are output depends on the timing of the processes, which is determined by the operating system scheduler. Fork is one of the most important system calls in UNIX, it allows the creation of many processes and the myriad functionality of the operating system.

D. pipes

Pipes is another type of system call in UNIX. This system call allows processes to communicate with one another. Processes can `read` or `write` to another process via pipes. There are several types of pipes in UNIX. One directional pipe and two-way pipe.

one directional pipe As the name suggests these pip can only pass information from process to file, or from file to process 1. The symbol denoting pipes in the command line is `>` and `<`. The `>` pipe translates to human language as ”send the output from the process on the left of the symbol and feed it into the file on the right on the symbol”. A simple example would be `echo "hello world" > hello.txt`. The bash code basically writes ”hello world” into the `hello.txt` file, and in the context of pipe, we are basically ”piping” the output of the echo command into the `hello.txt` file.

The other type of one-directional pipe is `<`. This is used as a command input redirection operator, which redirects the input of a command to come from a file instead of the standard input (usually the keyboard). When the shell encounters the ¡ symbol, it opens the specified file for reading and sets it as the standard input for the command to be executed. This allows the command to read its input from the file instead of from the keyboard. For example, consider `sort < input.txt` In this case, the sort command sorts the contents of the file input.txt, which is used as input via the ¡ operator. The output of the sort command is sent to the standard output, which is usually the terminal screen. Note that the ¡ operator creates a one-way communication channel from the file to the command, and it is the opposite of the `>` operator, which redirects the output of a command to a file.

bidirectional pipe

As the name indicates, this kind of pipe creates an undirected pipe between the two processes. The output from the first command is sent as input to the second process. The difference between the — and then `>` pipes is that the vertical pipe is used between processes, while the `>` pipes are used between files and processes. An example would be ’ls -l — sort -rnk 5’ This command will list the files in the current directory and then sort the output in reverse order based on file size. In this case, the pipe connects the ls process and the sorting process, piping the output of ls into the input of the sort.

The `pipe()` command can also be used in programming. The call `filep = pipe( )` opens a file descriptor and creates a pipe that allows processes to communicate with each other. The file descriptor that contains this channel is passed from the parent to the child via `fork()`, and thus would allow the parent process and the child process to communicate with each other and relay information even after splitting into different processes. However, these dependencies on inheritance means that a pipe can not be used an any two random processes, instead, there has to be some relationship between them. The modern Linux system, on the other hand, solves this issue by having a named pipe using a command named `mkfifo`. The pip that’s created from this command allows any two processes that have access to the name of this pipe to communicate with each other. We will not go into detail as this is not what the paper entailed.

E. the execute command

The ‘execute‘ command is a system call in Unix that allows a program to load and execute another program. The ‘execute‘ command takes the filename of the program to execute as its first argument, followed by any command-line arguments that should be passed to the program.

When the ‘execute‘ command is executed, the operating system reads the specified program file into memory and sets up a new process to run it. The new process inherits the open files, current working directory, and other system resources from the process that executed the ‘execute‘ command. Once the new process is set up, it begins executing the code in the specified program file.

Here is an example of using the ‘execute‘ command in C:

```
#include <unistd.h>

int main(int argc, char **argv) {
    char *args[] = {"ls", "-l", NULL};
    execvp("ls", args);
    return 0;
}
```

This program executes the ‘ls‘ command with the ‘-l‘ option. The ‘execvp‘ function is used to execute the ‘ls‘ command with the specified arguments. If ‘execvp‘ is successful, the ‘ls‘ command replaces the current process, and its output is displayed in the terminal. If ‘execvp‘ fails, the program returns 0. Note that the execvp here is one of the variations of the execute command that UNIX describes.

F. Process Synchronization and Multitasking

The wait command

When a process calls wait, it suspends its operation until one of its children has finished running. The wait function will return the PID of the child who finished running, or return an error if the process calling `wait()` does not have any child. In modern systems, the `wait()` system call for the child process is called `waitpid()`. Where in this case, the parent process can specify which child process it wants to wait for.

The exit command

The exit command, as suggested by its name, terminates the running process and makes it "exit" the CPU. More specifically the process's running image is destroyed, its opened file descriptors are closed and its parent is noticed through the wait command call. If the parent has already terminated when the child call `exit`, then the signal will be sent to its grandparents, or great grandparents and so on. The process with PID 1 will exist only when the operating system shuts down, so there will always be some process running. However, the modern Linux system takes a different approach. When a child process terminates after the parent process exit, called an orphan process, the init process will take over as a parent and call `wait()` to clean up the child process.

Putting exit and wait together

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main() {
    pid_t pid;

    pid = fork();

    if (pid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    } else if (pid == 0) {
        /* Child process */
        printf("Child process\n");
        exit(EXIT_SUCCESS);
    } else {
        /* Parent process */
        printf("Parent process\n");
        wait(NULL);
/* Wait for the child process to complete */
        printf("Child process completed\n");
        exit(EXIT_SUCCESS);
    }
}
```

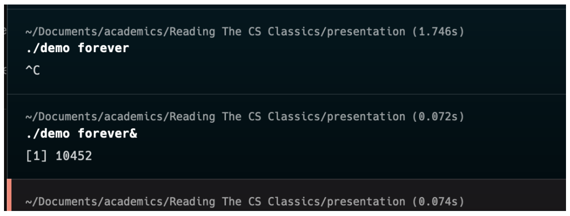
In this example, the parent process creates a child process using fork(). The child process prints a message and then exits using exit(). The parent process waits for the child process to complete using wait() and then prints a message indicating that the child process has been completed. Finally, both processes exit using exit(). As you can see the wait and exit call works together to ensure that all program terminates smoothly and all memories and resources are recycled when this happens.

Multitasking

Usually, when a process is running, the programmer can not do anything else on the shell except terminate the process. However, a special symbol `&` attached to the end of running a process will tell UNIX "send the process to run in the background, and return control to the user". For example

G. Shell

In 1974 when there was not much nice Graphical User interface to click around, people used the shell to interact with the UNIX operating system. The



```
~/Documents/academics/Reading The CS Classics/presentation (1.746s)
./demo forever
^C

~/Documents/academics/Reading The CS Classics/presentation (0.072s)
./demo forever&
[1] 10452

~/Documents/academics/Reading The CS Classics/presentation (0.074s)
```

FIG. 3. Adding & at the end of running a program will send it to the background

user types in a command and then provide input to communicate with the Operating system what to do next. However, if a command can not be found the shell program will prefix a `/bin` which has all the commands that users generally employ in their daily interaction, like `ls,cat`... etc. The shell is also responsible for dictating what are the data that goes into a running program, an what are the data that comes out as the process is running. As described earlier in the pipe section we see the > and <. These are actually the shell program calling `pipe()` undercover. These shell pipes control the flow of data between a process and a file descriptor.

Shell initialization

Unix initialization ends with the creation of a single process, called the init process as mentioned earlier. The keyboard will have a file descriptor of 1, while the printer will have a file descriptor of 0. Back then it was the printer, but now it is the prompt on the shell screen, also known as the standard output. After all, these are done, the shell then hangs and waits for user input. If after receiving a login signal followed by the correct username and password, the init process wakes up changes to the user's default current directory, sets the process's user ID to that of the person loin in, and performs an `execute` on the shell. At this point the shell is ready to receive user input and the login process is completed.

Other processes as Shell

In addition to the normal full-text shell we see as we open the "command line" app, another program can serve as a shell as well. A shell is basically a front-facing program that allows the user to interact with the Operating system, thus other shell includes a text editor, a chess game, and even 3D tic-tac-toe.

VII. IMPACT OF UNIX

The impact of UNIX on the field of computer science has been vast. Over fifty years after the development of UNIX, the elegant core ideas presented by Ritchie and Thompson in 1974[1] continue to be prominent in modern operating systems.

The file system developed by Ritchie and Thompson for UNIX has had a large impact across the field of computer science. The system view of UNIX had a unique perspective on files and system structure. This continues to be prominent in modern operating systems. The design of the file systems discussed earlier in this paper represents the core of UNIX's architecture. Ritchie and Thompson's innovative and elegant file abstractions continue to have an immense impact on system design in modern technology.

Since the development of UNIX, there have many numerous UNIX-based operating systems released. The descendants of UNIX and their prominence across various uses from scientific to personal machines represent a continuation of UNIX and illustrate the impact of Ritchie and Thompson's work. The direct descendants of UNIX include MacOS, Solaris, and the BSDs. There are also numerous popular Linux distributions including Debian, Fedora, Ubuntu, and more. Linux is used throughout computing, from mobile devices to desktops. [14]

As C and UNIX were developed in parallel, the prominence of C as a programming language is a lasting impact of UNIX. [3] C allowed for UNIX's hardware-independent design and UNIX solidified C as the foremost systems programming language.

UNIX expanded the domain of academic research. The hardware-independent design allowed for larger accessibility. This accessibility provided the base of systems-level software that can be used on a variety of machines, both expensive and inexpensive. The accessible system of UNIX impacted academic research. This base of systems software that could be used across universities and institutions would allow for results to be tested and verified by others with more ease. It also allowed for research to have an accessible base of systems software to build more software and experiments on top of it. The functionality of UNIX expanded the domain of academic research, allowing for a newfound ease of testing and verification of results.

UNIX also impacted modern computer programming. The Turing Award committee discussed this impact when awarding Ritchie and Thompson the award in 1983. "The genius of the Unix system is its framework, which enables programmers to stand on the work of others." [2] UNIX provided a framework that allowed programmers and researchers to build upon others' work. This represented a great shift in computer science that is evident today. UNIX revolutionized both academics and industry through its accessibility and innovative framework.

The impact of UNIX is immense and evident. Ritchie and Thompson changed the field of computer science in 1974. The legacy of UNIX is evident throughout the modern technological landscape. From academia to industry, UNIX influenced numerous innovations. In this paper, we presented UNIX as a classic paper in computer science. Through exploring its core ideas and its impact on numerous facets of the field, we feel UNIX's worthiness as a classic work is clear. After over fifty years since Ritchie and Thompson presented UNIX, it would be hard to find a piece of computer science that has not been impacted by it.

VIII. REFLECTIONS ON THE PRESENTATION

During our presentation on UNIX, we received multiple questions. In this section, we will explore these questions in further detail.

A. Systems Software Research is Relevant

At the conclusion of our presentation, we posed the question: "Is Systems Software Research irrelevant?" This was in response to Robert Pike's 2000 paper, "Systems Software Research is Irrelevant". The paper presented a pessimistic view of the state of software systems research. [16] Pike blames this on several things such as Microsoft, startups, Unix, and Linux. For UNIX he claims that UNIX was a victim of its own success: portability led to ubiquity. [16] This is why we decided to bring Pike's paper into this presentation.

Systems software research is a broad area that encompasses fields like operating systems, distributed systems, and more. As for the current state of systems software research, we argue for its relevance. Current systems research coming out of academia involves containerization, virtualization, confidential computing, and more. Numerous fields rely on secure and reliable software systems to support their work. The field of systems software research aims to provide these.

B. Modern Operating Systems

When we presented UNIX as a classic work of computer science, we were questioned about the current state of operating systems. While operating systems have evolved and innovated since UNIX's introduction, descendants of UNIX still have the original ideas from the 1974 paper at their core. The shell and the file system presented by Ritchie and Thompson in 1974 had an immense impact on modern operating systems. There have been numerous innovations since UNIX's introduction that have lent to the reliability and security of modern operating systems. While the number of new operating systems being introduced is small, the main operating systems continue to cultivate innovation in the field.

Another question we received was if there were any projects that use operating systems principles. Modern browsers use similar ideas as operating systems for isolation and sandboxing to protect privacy and security. For example, a concurrent browser innovation is the development of WebAssembly (WASM). WASM is an assembly-like low-level language that supports the running of multiple programming languages as a compact and fast binary. [17] This language can be run in modern browsers. [17] WASM uses principles of design a low-level system and applies them to modern browser design. There are numerous innovative projects that have been influenced by UNIX's core design.

In presenting Dennis M. Ritchie and Ken Thompson's 1974 paper, The UNIX Time-Sharing System[1], and its impact on computer science, we demonstrated its immense influence and status as a classic work in the field.

IX. CONCLUSION

With the publication of Dennis M. Ritchie and Ken Thompson's 1974 paper, The UNIX Time-Sharing System[1], the field of computer science changed. The work done by Ritchie and Thompson on UNIX has had a profound impact on computer science and has shaped our perspective on systems and programming. The central ideas of UNIX and the system's elegance[2] led to its widespread adoption. In our presentation and this paper, we have demonstrated why UNIX is a classic work of Computer Science.

[1] D. M. Ritchie and K. Thompson, "The Unix Time-Sharing System," *Communications of the ACM*, vol. 17, no. 7, p. 365–375, 1974.

[2] "Dennis M. Ritchie," 1983, dennis M. Ritchie - A.M. Turing Award Laureate. [Online]. Available: https://amturing.acm.org/award_winners/ritchie_1506389.cfm

[3] "The Good, the Bad, and The Ugly: The UNIX Legacy," 2001. [Online]. Available: http://doc.cat-v.org/bell_labs/good_bad_ugly/slides.pdf

[4] "Dennis Ritchie," 1998, national Science and Technology Medals Foundation. [Online]. Available: https://nationalmedals.org/laureate/dennis-ritchie

[5] "Dennis M. Ritchie," 2016, princeton University. [Online]. Available: https://www.cs.princeton.edu/~bwk/dmr.html

[6] D. Ritchie, "Program structure and computational complexity draft," 1967, computer History Museum. [Online]. Available: https://www.computerhistory.org/collections/catalog/102790971

[7] "Kenneth Lane Thompson," 1983, kenneth Lane Thompson - A.M. Turing Award Laureate. [Online]. Available: https://amturing.acm.org/award_winners/thompson_4588371.cfm

[8] "Coms4118: Introduction to unix." [Online]. Available: http://www.cs.columbia.edu/~jae/4118-LAST/L01-unix-intro.html

[9] "MULTICS History," 1983, multics History. [Online]. Available: https://multicians.org/history.html

[10] "The Evolution of the Unix Time-sharing System*," 1979. [Online]. Available: https://www.bell-labs.com/usr/dmr/www/hist.html

[11] "Apple II DOS source code," 11 2019, cHM. [Online]. Available: https://computerhistory.org/blog/apple-ii-dos-source-code/

[12] W. Foundation, "Ms-dos," 4 2023, wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/MS-DOS

[13] ——, "NeXTSTEP," 3 2023, wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/NeXTSTEP

[14] "The Complete History of Linux," 2023. [Online]. Available: https://history-computer.com/the-complete-history-of-linux-everything-you-need-to-know/

[15] "The development of the C language," 1993, chistory. [Online]. Available: https://www.bell-labs.com/usr/dmr/www/chist.html

[16] "Systems Software Research is Irrelevant," 2000. [Online]. Available: https://doc.cat-v.org/bell_labs/utah2000/utah2000.pdf

[17] "Webassembly." [Online]. Available: https://developer.mozilla.org/en-US/docs/WebAssembly