Functions & Services of an OS

CISC 3320 — CUNY Brooklyn College
Lecture Notes

Functions & Services of an OS

In this chapter, we will:

1. List and analyze various services that an operating system provides.
2. Discuss data structures that we'll use throughout the semester.
3. Introduce and bring examples of system calls.
4. Learn what structures operating systems could have.
5. Find out how to debug and analyze the performance of the operating system.
6. Describe the process of booting a computer and how the OS is involved in it.

Functions & Services of an OS

Services provided by any operating system are:

1. **Abstraction of Hardware**: Every hardware device requires a specific lower-level code to execute on it, as each device has its own design particularities.
   The operating system eliminates the need for the user or programmer to learn the specifics of each hardware component by introducing 'standardized' application and user interfaces. Examples:
   - RAM (or any memory storage, in general) comes in different sizes. When allocating memory for programs, the Operating System manages the available memory and 'hides' the details from the user (e.g., at which memory location a certain number is stored) to make the user's life easier.
   - A programmer who wrote low-level code (e.g., Assembly code) for one set of hardware may find that this code wouldn't run on other hardware. To remedy this, the OS supports common application programming interfaces (APIs), which include functions that the programmer can use in their code such that it runs on various devices.

Functions & Services of an OS

2. **User Operating-System Interface**: Allows the user to communicate with the operating system, such as by issuing command (instructions) to the OS or by obtaining information from the OS.
   We divide user interface types into 3 general categories:
   i. **Command-line interface (CLI)**: When a computer user logs into the computer, a program called **command interpreter** (also referred to as a **shell**) launches.
      - The command interpreter awaits the user to enter commands, and then runs the programs that perform these commands.
      - The interpreter might be a part of the kernel itself or, more commonly today, a separate program.
      - Some operating systems, like Mac and Linux, might have several shells installed on it, and users can choose what shell to launch by default.
      - The **command-line interface** is a wrapper program through which a user interacts with the shell. On Windows, this interface is called **Command Prompt**, and on Linux or Mac, it is called the **Terminal**.

Functions & Services of an OS

The video at https://www.youtube.com/watch?v=Us3G-nJYru0 shows the execution of some basic commands on a Ubuntu Linux terminal. The command interpreter that is being used in the video is most likely **Bash** (**B**ourne **A**gain **Sh**ell):

https://www.youtube.com/watch?v=Us3G-nJYru0

Functions & Services of an OS

2. ii. **Graphical user interface (GUI)**: Allows a user to interact with visual objects on a device's monitor via gestures, such as clicking, swiping, dragging & dropping, etc.
    - Fosters the use of terms like icons, folders, buttons, ribbons, windows, etc.
    - Each icon or button represents a file, folder, executable (binary file), link, or disk.
    - Clicking, or using other gestures, on an icon or button would either access the underlying item or perform an action upon it.
    - GUI is more convenient for use on touch devices than a command-line interface.

   iii. **Batch systems**: Represents a method in which a collection of tasks, called **jobs**, are executed in groups (recall the batch OS type.)
    - A user would interact with such a system by either supplying punch cards or describing what jobs to perform using a job-control language.
    - Batch systems are used today on specific-purpose machines, like bank or payroll systems.

Functions & Services of an OS

3. **Executing Programs**: The operating system is responsible for running programs on the device.

   A compiled program is a file stored on disk that contains execution instructions. To make the program run, the operating system must load the instructions into RAM, and control the execution of each instruction in the given order.

   When a program is loaded into memory, we call it a **process**. The process's data (the program's instructions variables, arrays, objects,) are stored in an array-like data structure called the **process table**.

   Once all the instructions have been executed, the OS needs to terminate the program, which is done by removing the program's data from memory. If a runtime error occurred during execution that prevents the program from continuing running, and no measures that the operating system takes resolve the error, the OS will terminate the program in an abnormal way.

Functions & Services of an OS

4. **Managing Resources**: A device has a limited amount of resources that are critical for the execution of programs, including a limited amount of CPUs, limited memory size, limited secondary storage space, and a limited number of peripheral devices (like printers or speakers.) Since a running program might request more resources than available (e.g., the allocation of memory for a huge array,) or several programs that are running simultaneously may compete for resources, the OS must control the availability and allocation of resources among programs. The "CPU Scheduling" and "Memory Management" topics elaborate on this OS role.
   Examples:
   - When a program requests to allocate a piece of memory greater than the available, the OS will either deny the request, thereby terminating the program, or may use virtual memory (i.e., disk space instead of RAM space), if such a functionality is supported.
   - When the number of programs that are running concurrently is greater than the number of CPUs, the operating system will 'juggle' the programs. That is, each program will get a few milliseconds to run on an available CPU before being replaced by another program, a cycle that will repeat until the programs complete their execution.

Functions & Services of an OS

5. **Handling i/o requests**: As we discussed in the previous topic, an operating system handles i/o requests. That is, when an i/o device needs to transfer data (to or from the computer,) the operating system needs to take an appropriate action for the transfer to take place, such as by executing a particular i/o device driver. We will learn more about i/o in the next topic, "I/O and Interrupts".
6. **Manipulating the file system**: Files are stored on disk, but are associated with folders (a.k.a., directories). The operating system's role is to maintain the linkage between a file and its directory, between directories and their parent directories, and between the names of files and their corresponding storage locations on disk. More on this in the "File Management" topic of the course.
7. **Process Communication**: In many cases, two programs need to exchange data, such as when one calls the other. Various process communication techniques that programs use are managed by the operating system, such as signaling, pipes, and memory sharing. This topic will be covered in more detail in the "Processes" topic.

Functions & Services of an OS

8. **Handling errors**: When an error occurs during the execution of a program, or when a hardware error is detected, the OS is expected to handle the error, i.e., to find a suitable solution that will allow proper continuation of the execution of the device's tasks. The operating system might employ various methods that include error prevention, detection, and recovery algorithms and backup or redundancy solutions, in which data is stored in more than one copy/location. Examples of error prevention and detection algorithms are discussed in the "Deadlock" topic of the course.
9. **Security & Protection**: As the end of the previous topic explained, it is the responsibility of the operating system to prevent the overwriting or corruption of data from either intentional or unintentional actions. We'll dive into more details in the "Introduction to Protection and Security" topic.
10. **Accounting**: To operate smoothly, the OS keeps a record of the current system performance and the availability of resources. This information could help the developers of the operating system learn how to make improvements to its performance.

Functions & Services of an OS

Operating System functions and services. Miriam Briskman, CC BY-ND 4.0.

Note: click the GIF amination above to open and play it in a browser window.

Functions & Services of an OS

Any questions so far?

Data Structures

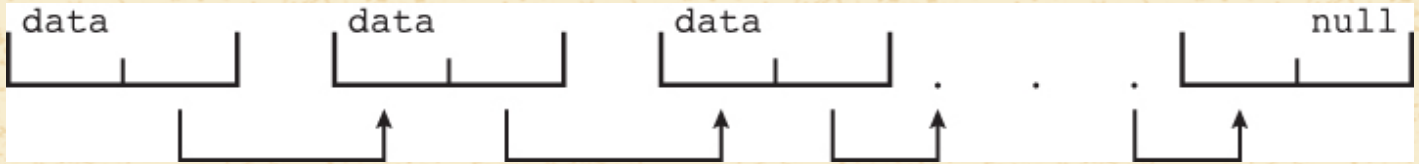
As we touch on various aspects of the operating system, we will learn that the OS uses different data structures to accomplish tasks and solve problems. As a heads up, we briefly review some of these data structures here.

1. **List**: A data structure built up from **nodes**, each of which contains a **data** section and at least one **address** (or **link**) section. The data section stores some element (number, character, string, or even another object), and an address section contains the location in memory of the next (or previous) node.

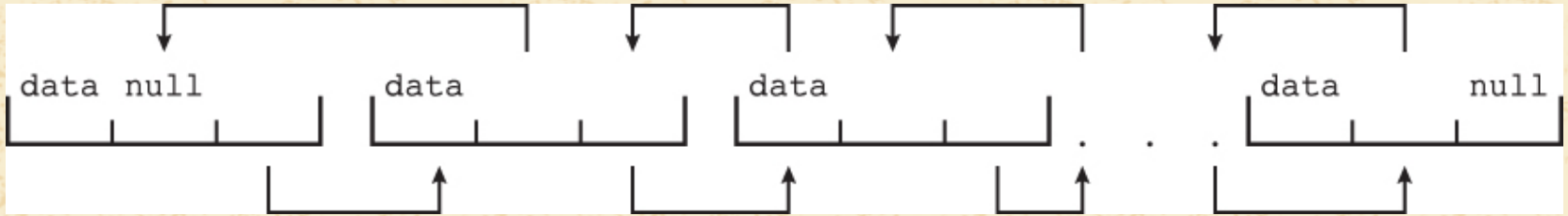
   There exist different types of lists, including **singly linked lists** (each node points only to the next node), **doubly linked lists** (each node points both at the previous and the next node,) and **circularly linked lists** (the last node points back at the first node.)

   Some operations, such as accessing the *n*th element of a list, may require you to traverse the entire list (rather than accessing the particular element directly, as you would do when using an array.)

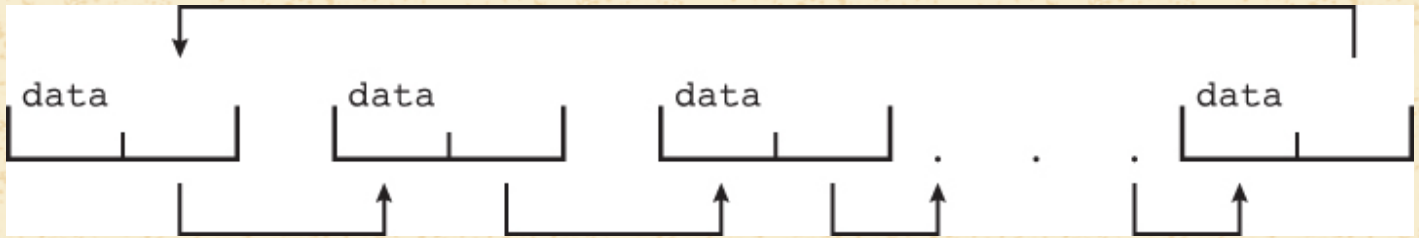
Data Structures



Singly linked list. Taken from Bell, John T. "Operating-System Structures." University of Illinois, Chicago.



Doubly linked list. Taken from Bell, John T. "Operating-System Structures." University of Illinois, Chicago.



Circularly linked list. Taken from Bell, John T. "Operating-System Structures." University of Illinois, Chicago.

Data Structures

2. **Queue**: A data structure that behaves in a sequential manner that is similar to a list, but abides by rules that are more strict than those of a list. Specifically, an element can be inserted into the queue only at the tail (end) of the queue and removed from the queue's head (start). A common term for this behavior is "First In, First Out" (FIFO).

   Queues are used by the OS, for example, when deciding what programs should run next on a CPU.

3. **Stack**: Another sequential data structure. The difference between the stack and the queue is that the stack supports "Last In, First Out" (LIFO). That is, to access or remove an element from the stack, you must first remove all the elements lying on top of it.

   We'll return to stacks when we cover function-calling and recursion since their data are stored in a stack.

Both of these data structures are usually implemented with lists.

Data Structures

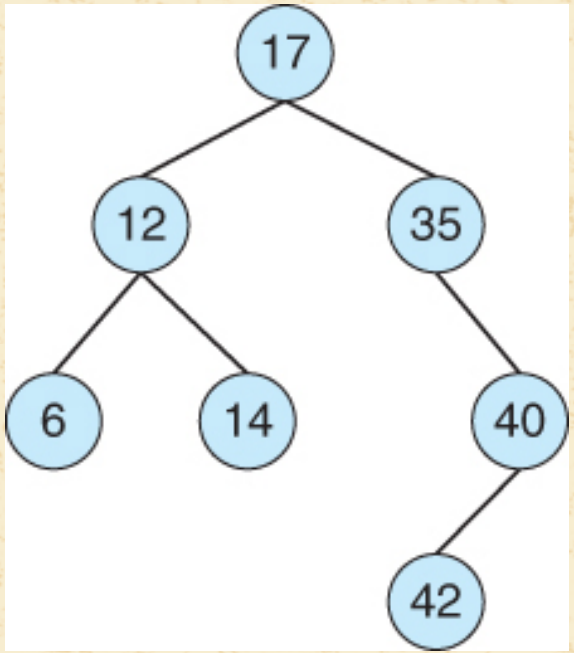
4. **Tree**: A data structure whose nodes link back to their parent nodes and might also link to children nodes.

   Because each node can be either a child or a parent of some node, the tree establishes a hierarchical relation between the nodes. The node in the tree that doesn't have a parent is called the **root**.

   Different types of trees include: **binary trees** (in which each node can have up to 2 children), **general trees** (in which there is no limitation on the number of children a node can have), and **binary search trees** (in which the left child has a value that is less than or equal to the value of the right child.)

We will encounter trees when talking about the file system (which contains files inside folders inside folders, etc.) and CPU-scheduling algorithms.

Data Structures



A binary search tree. Taken from Bell, John T. "Operating-System Structures." University of Illinois, Chicago.

Any questions so far?

Data Structures

5. **Hash Map**: A data structure that relates (or 'maps') a key with a value, each of which could be a number, string, object, etc.

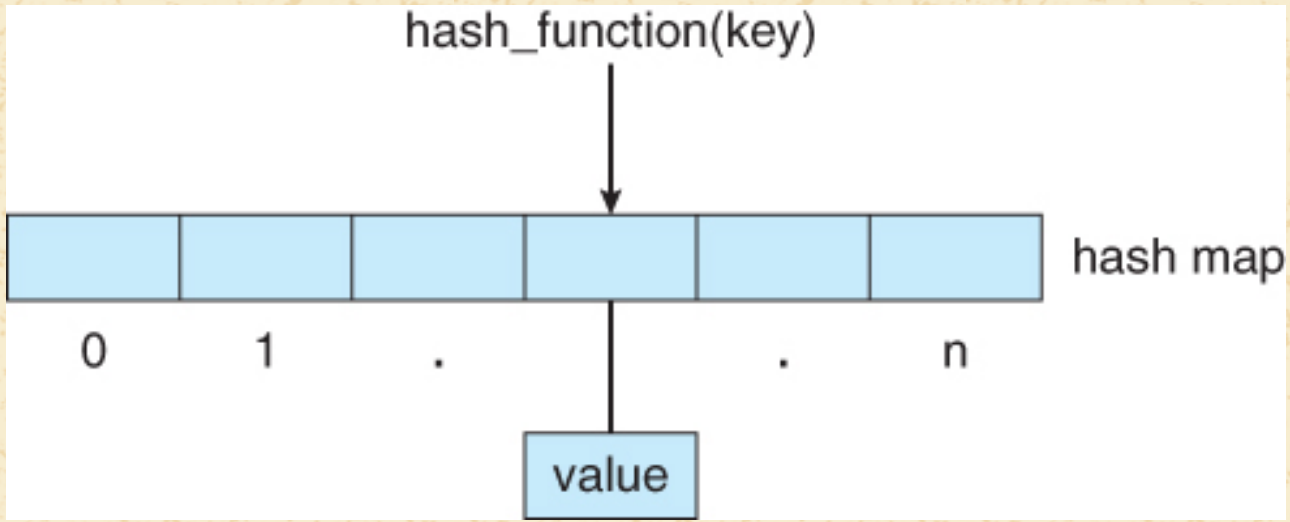
   A key is mapped to a value by using a **hash function**, which applies a mathematical calculation to the key and returns a numerical value. This value is then used as an index to a hash table, where the key's associated value is stored.

   Because a hash function returns 'almost' unique values, there is a slight chance that two different keys will produce the same hash function result. As such, the hash map might resort to storing the values of the two keys in a list. That is, the hash table is an array of lists.

   The advantage of hash maps is that, when we search for a value based on a given key, we can find it in constant time because the underlying data structure is an array.

Data Structures

5. We will see how the operating system uses this advantage when we discuss, for example, hashed page tables as part of the "Memory Management" topic and directory hash tables in "File Management".

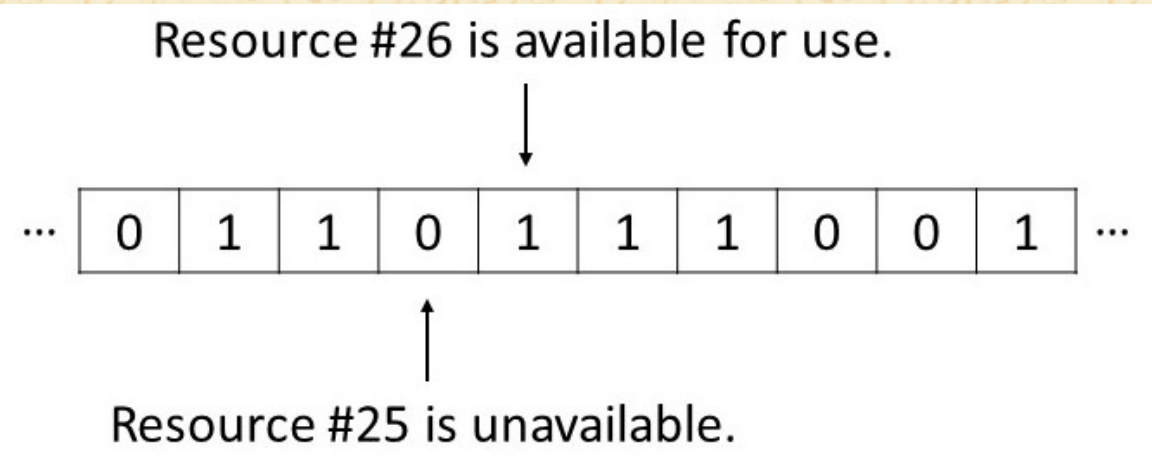


A hash map. Taken from Bell, John T. "Operating-System Structures." University of Illinois, Chicago.

Data Structures

6. **Bitmap**: This is an array of 0s and 1s (also called a **bit vector**.) Each array slot is associated with an entity, like a resource, device, or storage block, and indicates whether the entity is available (1) or unavailable (0) for use.

   Bitmaps will be mentioned in various topics in our course, such as "Memory Management" and "File Management".



A bitmap of resources. Miriam Briskman, CC BY-ND 4.0

System Calls

Let's return to OS services.

How do programs and applications request services from the operating system? They use system calls.

System calls (or **syscalls**) are low-level functions (usually written in C, C++, or even Assembly) that a program can call to request the OS to perform a certain service, such as allocating memory for a variable, creating a file, or send a message to another program. Linux distributions use about 300 system calls, while Windows uses about 2000(!)

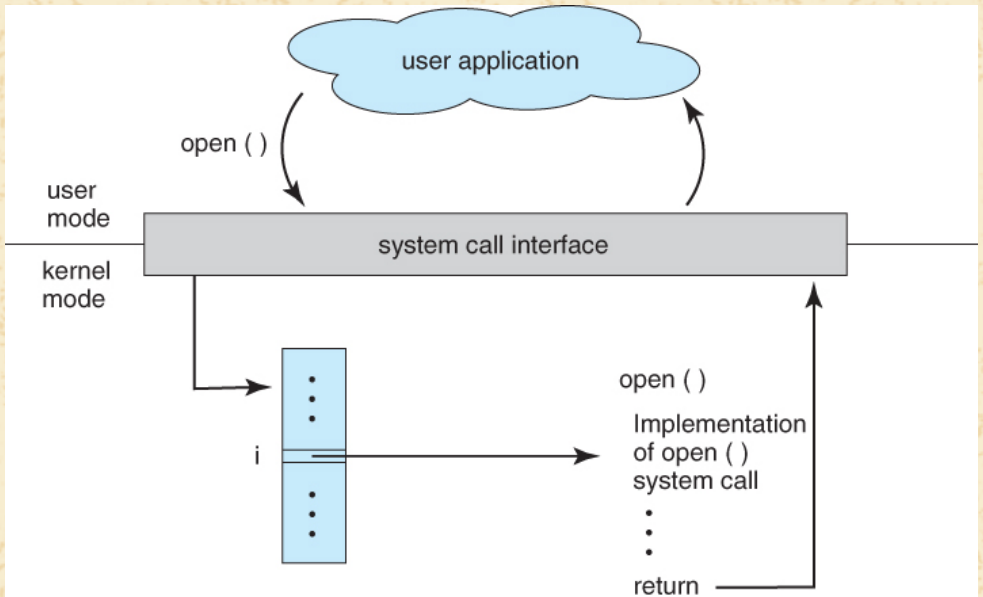
About 90% of all the syscalls used on Linux are shared by all Linux distributions (including macOS,) but there are exceptions.

Because of those exceptions, programmers prefer to use API functions, which are more portable than system calls.

- That is, programs containing API function calls will compile and run on every Linux distribution.

System Calls

Any API function, internally, invokes a system call. The OS program that connects API function calls to system calls is the **system call interface**. When a function is called, the interface looks up the corresponding number of the system call based on which function was called.



Invoking the `open()` system call. Taken from Bell, John T. "Operating-System Structures." University of Illinois, Chicago.

System Calls

The diagram on the previous slide shows two environments: the user space and the kernel space.

The **kernel space** is the section of memory (RAM) in which the kernel performs privileged actions and services (recall the discussion about privileges.)

User space is any other memory section not belonging to kernel space.

Since the invoking of system calls is considered privileged, mostly since the abuse of system calls can damage the operating systems or corrupt data, system calls can only be performed on kernel space.

While a system call is executing, its parameters (usually, no more than 2-3) are stored on the CPU's registers.

If a parameter is larger than what a register can hold, the register will store a pointer to a memory location where the parameter's data is stored.

System Calls

We can divide system calls into the following 6 general categories:

- **Processes**: create, load, terminate, abort, get attributes, wait for a process to finish, and allocate & free memory.
- **Files**: create, open, close, delete, read from, write to, move, get attributes of, and set attributes of files.
- **Devices**: request, release, read from, write to, move, attach, detach, get attributes of, and set attributes of devices.
- **Information Maintenance**: get & set time and date, get & set system data, and get & set process, file, or device attributes.
- **Communication**: create & delete a communication connection, send & receive messages, transfer status info, and attach or detach remote devices.
- **Protection**: get & set file access permissions (read, write, and execute) and get & set file owners.

Next, let's view some examples of system calls on Linux/Mac and Windows operating systems. The full list of Linux system calls is at https://man7.org/linux/man-pages/man2/syscalls.2.html.

System Calls

	Windows	Unix

Process Control	`CreateProcess()`
	`ExitProcess()`
	`WaitForSingleObject()`
		`fork()`
		`exit()`
		`wait()`

File Manipulation	`CreateFile()`
	`ReadFile()`
	`WriteFile()`
	`CloseHandle()`
		`open()`
		`read()`
		`write()`
		`close()`

Device Manipulation	`SetConsoleMode()`
	`ReadConsole()`
	`WriteConsole()`
		`ioctl()`
		`read()`
		`write()`

Information Maintenance	`GetCurrentProcessID()`
	`SetTimer()`
	`Sleep()`
		`getpid()`
		`alarm()`
		`sleep()`

Communication	`CreatePipe()`
	`CreateFileMapping()`
	`MapViewOfFile()`
		`pipe()`
		`shmget()`
		`mmap()`

Protection	`SetFileSecurity()`
	`InitializeSecurityDescriptor()`
	`SetSecurityDescriptorGroup()`
		`chmod()`
		`umask()`
		`chown()`

Names of some system calls on Windows and Linux/Mac. Adapted from Introduction to Operating System.

System Calls

Any questions so far?

Operating System Structures

Throughout their existence, operating systems have been evolving not only in their power and speed, but also in structural complexityand the variety of components:

1. **Monolithic Structure**: At their dawn, OSs comprised a single binary file – the kernel – that executed all the OS functions. *Mono* means 'one' in Greek, and *lithos* means 'rock'.

   Despite such a structure being called 'simple', it is quite difficult to build a monolithic kernel that performs all sorts of OS services.

   Nonetheless, the advantage of such OSs is that they execute fast (since no time is spent on communication between the kernel and other OS programs – since only the kernel is present with the OS.

Operating System Structures

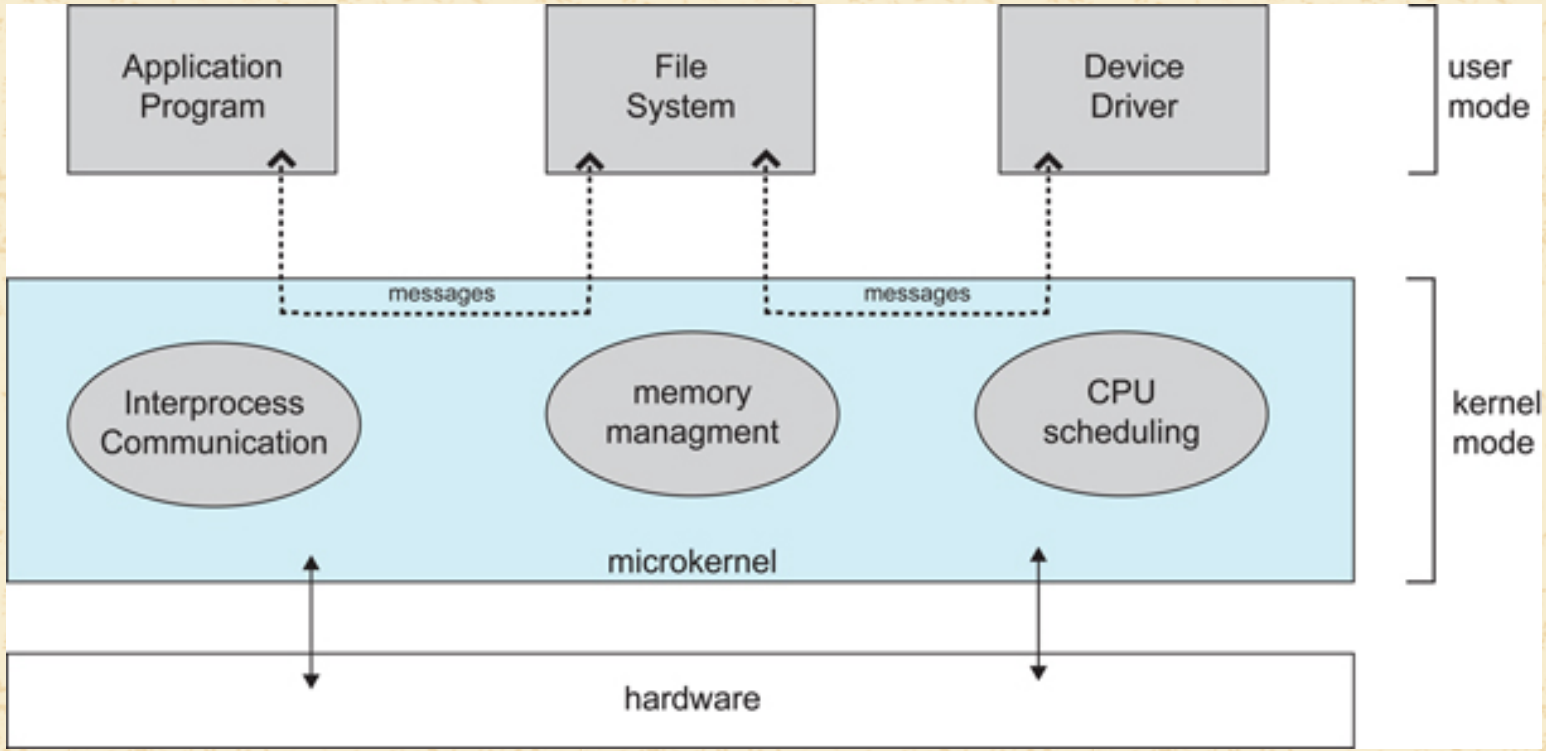
2. **Layered Structure**: A more complex OS structure implements layers. Each layer uses the services of the layers below it, with the lowest layer (layer 0) interacting directly with the hardware.

   Layered operating systems are easier to debug, fix, and therefore implement: each level can be debugged independently of the layers below it.

   On the other hand, since the layers must communicate with one another to transfer data and perform services, such communication will incur extra time (also called **overhead**) besides the actual execution of the function or service.

3. **Microkernel Structure**: A microkernel is one that is being removed all non-essential services, which instead are implemented as system applications. This makes the kernel smaller and robust, so, as the OS evolves in the future, no changes are needed in the kernel. However, the frequent system calls create a significant overhead.

Operating System Structures



A microkernel and its communication with the user space. Taken from Bell, John T. "Operating-System Structures." University of Illinois, Chicago..

Operating System Structures

4. **Moduled Structure**: A moduled OS is similar to a layered operating system in terms of the distribution of duties among the layers. However, the advantage of the moduled OS is that each module is capable of contacting any other module, without going through several layers of modules.

   This construction of this kind of OS uses object-oriented approaches.

   The kernel in this type of architecture is similar in size to a microkernel.

5. **Hybrid Structure**: Today, most operating systems use a mixture of the approaches mentioned so far. For example, Windows is monolithic + microkernel, while Linux is monolithic + modular.

Operating System Structures

Let's watch a summary of these OS structure types, with nice illustrations!

https://www.youtube.com/watch?v=-HQ0FNrHMsE

Operating System Structures

Any questions?

Operating System Debugging

Debugging is the process of removing software or hardware **bugs** (= errors).

Debugging is a difficult activity and, according to this study, occupies about 35% to 50% of a programmer's time.

A wise description of debugging was nailed by Brian Kernighan:

> "Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it." (*Kernighan's Law*)

Performance issues are considered OS bugs, so it is necessary to discover what causes bad performance and prevent that from happening.

Operating System Debugging

Failure Analysis is the activity of discovering what caused a program to fail.

Two approaches could be used by the operating system to record the failure-related data:

1. The OS could write any errors that occur during the execution of the program to a **log file**, which can later be reviewed by the computer administrator or the user who ran the program. This data could hint at what causes the program to fail.
2. The OS could copy data in memory that the program used to a file at the moment when the program crashes. This process is called a **core dump**. Since a core dump represents the state at which the program failed, a programmer could later use this data to discover what caused the failure.

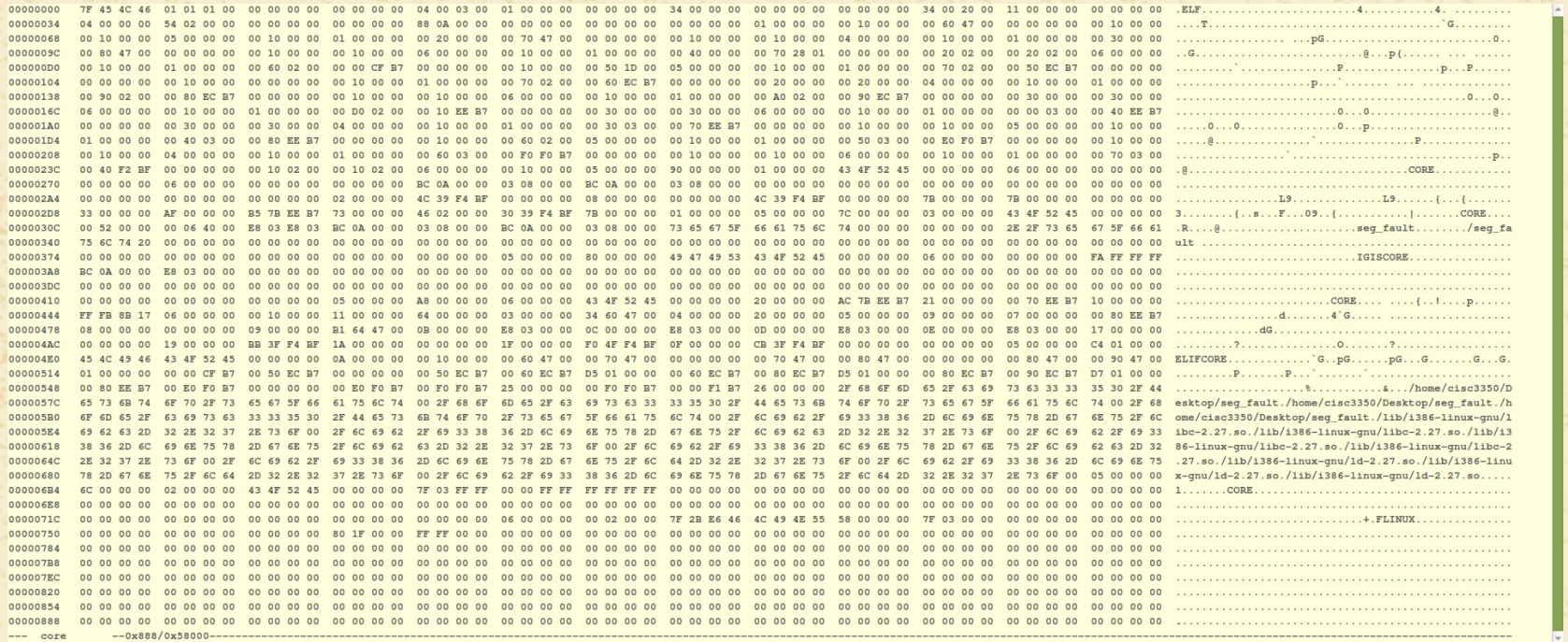
However, when operating system code needs to undergo debugging, the methods above might be inappropriate for the next reasons.

Operating System Debugging

1. The recording of errors during a program's execution requires running error-tracing code while the program executes, which, if done on OS code, will make the operating system run very slow due to this extra error-tracing code that runs in the background.
2. The OS makes frequent contact with hardware, so it is difficult to trace code-related (software) errors.
3. For regular program debugging, the programmer could use user-level debugging tools, such as debuggers or performance utilities, which cannot be applied to OS code (since, if the OS fails, it won't provide these services.)
4. If an error occurs with the file system, which is used in storing files, saving a core dump will be impossible if the file system fails.

As such, a solution to the issues above would be to save a core dump at the time when operating system code fails to an especially-dedicated section of the disk that the file system doesn't control. This way, even if the file system fails, the data of the core dump could be safely saved and later accessed by the system programmers for debugging purposes.

Operating System Debugging



The view of a core dump file of a C user program named `seg_fault` that crashed. Miriam Briskman, CC BY-ND 4.0.

Operating System Debugging

```
Sep 14 19:36:05 cisc3350-student systemd-logind[515]: Watching system buttons on /dev/input/event0 (Power Button)
Sep 14 19:36:05 cisc3350-student systemd-logind[515]: Watching system buttons on /dev/input/event1 (Sleep Button)
Sep 14 19:36:05 cisc3350-student systemd-logind[515]: Watching system buttons on /dev/input/event2 (AT Translated Set 2 keyboard)
Sep 14 19:41:21 cisc3350-student sudo: pam_unix(sudo:auth): authentication failure; logname= uid=1000 euid=0 tty=/dev/pts/0 ruser=cisc33
50 rhost=  user=cisc3350
Sep 14 19:41:28 cisc3350-student sudo: cisc3350 : TTY=pts/0 ; PWD=/home/cisc3350/Desktop ; USER=root ; COMMAND=/usr/bin/apt install base
z
Sep 14 19:41:28 cisc3350-student sudo: pam_unix(sudo:session): session opened for user root by (uid=0)
Sep 14 19:41:43 cisc3350-student sudo: pam_unix(sudo:session): session closed for user root
Sep 14 19:52:07 cisc3350-student sudo: cisc3350 : TTY=pts/0 ; PWD=/home/cisc3350/Desktop ; USER=root ; COMMAND=/usr/bin/apt install hexe
dit
Sep 14 19:52:07 cisc3350-student sudo: pam_unix(sudo:session): session opened for user root by (uid=0)
Sep 14 19:52:18 cisc3350-student sudo: pam_unix(sudo:session): session closed for user root
Sep 14 19:56:58 cisc3350-student systemd-logind[503]: New seat seat0.
Sep 14 19:56:58 cisc3350-student systemd-logind[503]: Watching system buttons on /dev/input/event0 (Power Button)
Sep 14 19:56:58 cisc3350-student systemd-logind[503]: Watching system buttons on /dev/input/event1 (Sleep Button)
Sep 14 19:56:58 cisc3350-student systemd-logind[503]: Watching system buttons on /dev/input/event2 (AT Translated Set 2 keyboard)
Sep 14 19:57:00 cisc3350-student useradd[699]: failed adding user 'vboxadd', data deleted
Sep 14 19:57:00 cisc3350-student useradd[704]: failed adding user 'vboxadd', data deleted
Sep 14 19:57:13 cisc3350-student lightdm: pam_unix(lightdm-autologin:session): session opened for user cisc3350 by (uid=0)
Sep 14 19:57:14 cisc3350-student systemd-logind[503]: New session c1 of user cisc3350.
Sep 14 19:57:14 cisc3350-student systemd: pam_unix(systemd-user:session): session opened for user cisc3350 by (uid=0)
@@@
                                                                                                                1,1           Top
```

The view of a log file of a Linux authentication system. Errors are shown in red. Miriam Briskman, CC BY-ND 4.0.

Operating System Debugging

To improve an operating system's performance, it is necessary to identify **bottlenecks**, which are situations in which a computer's performance is limited. This procedure is often called **Debottlenecking** or **performance tuning**. As such, the operating system must provide tools for monitoring running programs and general system behavior.

One way of tracking system behavior is by writing system activities into log files, as we've already seen.

A second way is for the operating system to provide system utilities that help monitor system activity.

- On Linux/Unix, one could issue commands that track system activities, such as `top`, which reports real-time statistics for current processes, or `ps`, which reports information for a single process or selection of processes.
- On Windows, one could launch the **Task Manager** application, which shows process statistics, as well as CPU, Memory, and Disk real-time performance.

Operating System Debugging

Here is an illustration of how the `top` command runs on a Linux computer:

https://www.youtube.com/watch?v=idT301bNPIA

Operating System Debugging

Here is an illustration of how the Windows Task Manager utility operates:

https://www.youtube.com/watch?v=lp27u1_7e7s

Operating System Booting

Booting is the process of loading the operating system into memory and, thereby, launching the OS. It occurs in several stages:

1. When the power button is pressed, a hardware interrupt that takes place causes the copying of an address to one of the CPUs. This address points to the 1st instruction of the **bootstrap program** (or **boot loader**) program on the read-only memory (ROM) section of memory.
2. Since the data of the boot loader software is already inside main memory, the boot loader starts running.
3. While running, the program checks that all the essential hardware devices are connected and available for use.
4. If all the necessary hardware devices pass the check, the program continues to find the storage location of the OS kernel program on disk.
5. The bootstrap program copies the instructions of the OS Kernel into memory, and the OS begins running.
6. The OS mounts the root file system. That is, the OS finds the location on the disk of the root directory and establishes access to it.

Operating System Booting

Many devices load a small boot program, called **BIOS** (Basic Input/Output System), which, in turn, loads a more complex boot loader program that then continues to load the OS into memory.

BIOS operates by accessing secondary storage (disk) section-by-section to execute the 2nd stage boot loader.

In newer devices, the BIOS program is replaced by **UEFI** (Unified Extensible Firmware Interface), which has several advantages over BIOS, including faster execution and support for larger storage devices.

UEFI works by storing the operating system's initialization info on disk rather than in memory and requires no 2nd stage boot loader to run (which is why it is faster than BIOS.)

A great article on the differences between BIOS and UEFI is at https://www.maketecheasier.com/differences-between-uefi-and-bios.

Operating System Booting

Any questions?

Functions & Services of an OS: Sources

Bell, John T. "Operating-System Structures." *University of Illinois, Chicago*, Accessed 11 July 2022. URL: https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/2_Structures.html

Fay-Wolfe, Victor. "Operating Systems." *University of Rhode Island*, Accessed 11 July 2022. URL: https://homepage.cs.uri.edu/faculty/wolfe/book/Readings/Reading07.htm