Final Project

Shaun Hoadley

CPT304: Operating Systems Theory and Design

Joshua Reichard

July 5, 2020

***Week 1 Interactive Assignment:***

**Categorize, describe, and give examples of the major functions of operating systems.**

The major functions of operating systems, according to Silberschatz, Galvin, and Gagne, include a user interface, program execution, I/O operations, file system management, communication management, error detection, resource allocation, accounting, and protection and security (2014, 2.1).

The UI, or user interface, is exactly what the name implies. The UI is what the user sees an interacts with and comes in several forms. There is the command-line interface (CLI), where commands are typed by the user in a text-based environment, such as MS-DOS and (still frequently used with) Linux. There is a batch interface, where the commands are pre-loaded in a file and executed by running the file. In the days of MS-DOS, you would typically load a video game by running a batch file that would start the game, set up the sound card configuration, and set up the graphics display configuration, so the game would function properly. Finally, there is the GUI, or graphical user interface. GUIs provide visual interaction with a system and are considered by most as being the most user-friendly. Some examples of GUIs include Microsoft Windows, MAC-OS, and many Linux distros (distributions) such as Fedora or Ubuntu.

Program Execution is the next major function of the OS. A computer must be able to load, run, and terminate a program. The next function of an operating system is I/O, or input/output, operations. When a running program needs to access a file or other device, it communicates the request to the OS, which in turn provides the required access.

Now we come to file-system manipulation. The OS handles the organization of files, creating new files, deleting files, and searching files by a variety of different methods.

Many OSs today also control who is able to access certain files, directories, and drives.

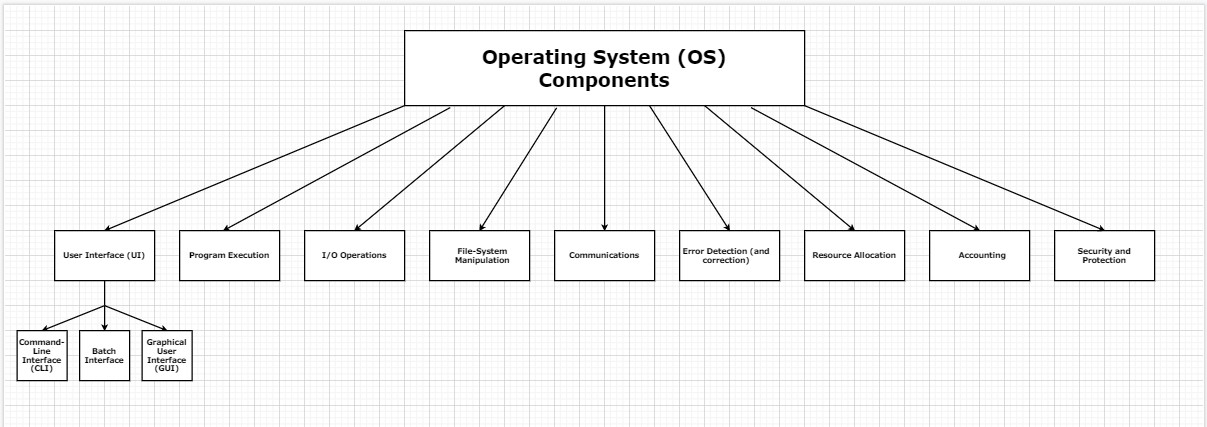
The OS handles communications for programs and hardware, either on the same system or between systems that are networked together.

Error detection (and correction) is another vital function of the OS. The OS attempts to keep everything moving efficiently, but sometimes the only recourse it has is to halt the operation.

Computer systems have a finite number of resources available for use. The OS is responsible for allocating those resources based on priority and reallocating them when the current operation is completed.

Accounting, not in the financial sense, is also handled by the OS. It keeps track of all the resources, which users/devices access those resources, and provide statistics on the resources.

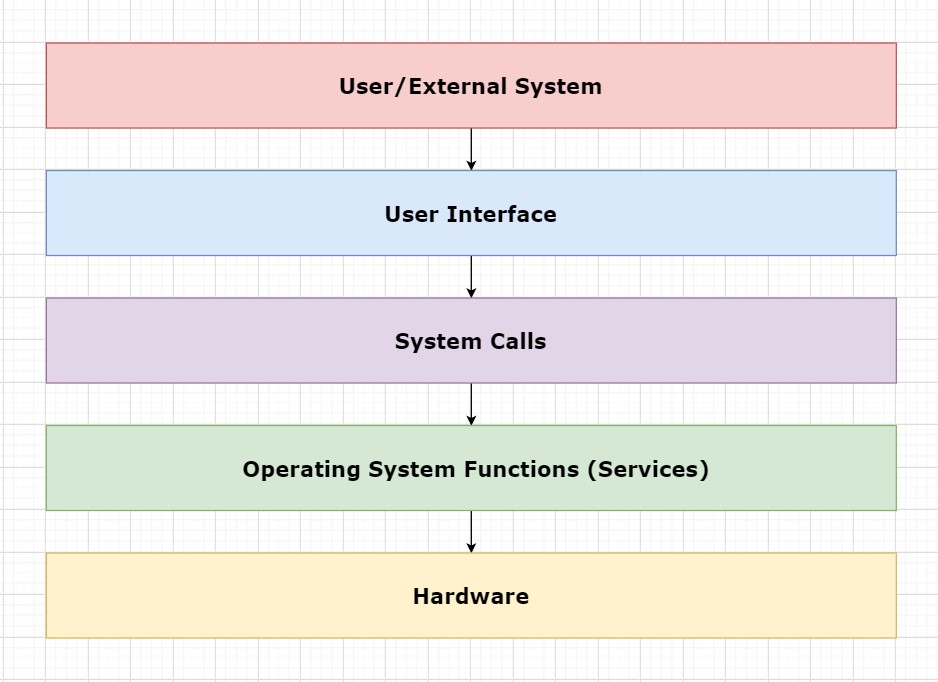
Lastly, the OS is responsible for security and protection. This is done in many ways, including the setting of permissions for users, authentication to ensure a user is valid (particularly if being remotely accessed), and encrypting data so it can only be accessed with the appropriate key.



**Figure 1: Operating System Components**

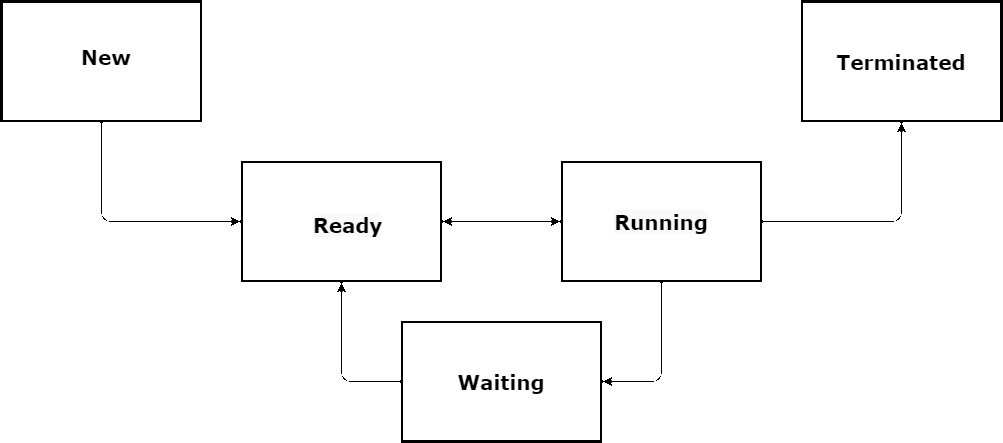
**Illustrate the hierarchy of subsystems, components, or subcomponents of operating systems and explain how they interact with one another.**

The hierarchy of subsystems, components, and subcomponents goes from user, either a person or another system outside of the computer, user interface, system calls, OS services (functions), and hardware.



**Figure 2: Operating System Hierarchy**

***Week 2 Interactive Assignment:***

****

**Figure 3: Process States**

**Describe process, process state, and process control block.**

According to Silberschatz, Galvin, and Gagne (2014), a process is a program that is being executed. While a program itself is static, “process” is the word used to describe a program whose instructions, along with possible interaction from the user, have been fetched from memory with the intention of execution. Processes can be categorized into different stages at any particular moment in time. This classification of the stage is known as the process state. A new process created has the state of that process as “new.” Example, a user clicks on the icon for the application Microsoft Word, the state of the first process initiated is “new”. When the instructions for the program are being executed, the process state is “running.” In this example, this might occur when the program is opened and the application asks the user what they would like to do, create a new document, open an existing document, etc. While the application waits for user input, the state of the process is “waiting.” The process state is “ready” when the process (creating a new Word document) is waiting to be assigned to a processor. After execution of the program, when the user has created their Word document, saved it, and closed the Word application, the state of the process is “terminated.” In this state, execution has finished and all the resources that were being used during execution are freed up (Silberschatz, Galvin, & Gagne, G., 2014).

A process control block (PCB), or task control block, is a data structure that stores the temporary data relevant to the state of a process. Each process has its own process control block, and all PCBs are maintained in a process table by the operating system. In multiprocessing, the process control block is extremely important, since the execution of processes often switch back and forth so that the CPU is never idle, leading to increased performance. If we go back to the Word application example, when Word initially asks the user what they would like to do, the process needs to wait for user input. The process would, at this point, get added to the I/O queue, freeing up the CPU for other processes that are running simultaneously. The CPU needs a way to “remember” where the process was in its state of execution before it got added to the I/O queue, so that it can resume where it left off once user input is received, the OS relies on the PCB for this function. The PCB stores information including the process ID, process state, program counter, register information, scheduling information, memory related information, accounting information, and I/O information (resource utilization and files opened), all to restore the process back to its correct state when it resumes execution on the CPU (Shukla, 2017).

**Compare single and multi-threaded motivations and models.**

According to Silberschatz, Galvin, and Gagne (2014), a thread is “a flow of control within a process.” Multi-threaded models are more efficient and yield higher performance than single-threaded processes. In a single-threaded model, a single process requires all of the resources necessary for execution (source code, data, files, memory, registers, a program counter, and a “stack which contains the execution history” (Tutorials Point, 2018). The advantage of a multi-threaded approach is that parallelism is employed using multiple threads that run simultaneously and share resources. Whereas a single-thread is heavy on the system (uses a lot of resources), a multi-threaded approach overall uses fewer resources, since the threads share code, data, and files. In addition, a single-threaded model requires interactions with the OS in order to employ switches, whereas OS interactions are not required in a multi-threaded model. Lastly, a single-threaded model can cause stalls when a process gets blocked, however, in a multi-threaded model, if a thread gets blocked, other threads for the same task can continue to run (Tutorials Point, 2018).

A multi-threaded approach may seem more advantageous than a single-threaded approach, there are challenges to multi-threaded models. There are two different levels of threads. User-level threads are controlled by the user and are not seen by the kernel. Additionally, they are faster to create. Kernel-level threads are created and managed by the operating system and are slower to create (Tutorials Point, 2018). It is important to know about the user and kernel levels to grasp the interactions between the levels during multi-threading.

There are three different models for multi-threading: Many to Many, Many to One, One to One. With the Many to Many model, multiple user-level threads multiplex to many kernel-level threads. If a thread gets blocked, another can be scheduled so that the process does not come to a halt. In this model, threads run in parallel, providing the best accuracy on concurrency. The next model is Many to One. With this model, many user threads multiplex with a single kernel thread. Therefore, if a thread gets blocked, the whole system must wait until it becomes unblocked. Finally, the multi-threaded model One to One is the model in which there is one user thread for every kernel thread. Multiple threads can run simultaneously on a multiprocessor, and if a thread gets blocked, the other threads can still run. However, in order to create a user thread, there must be an available kernel thread (Tutorials Point, 2018).

**Describe the critical-section problem and explain a software solution that resolves this problem.**

The critical-section problem refers to the fact that processes running in parallel cannot be allowed to access shared resources simultaneously. Most processes contain a segment of code that instructs the process to change a variable, update a table, or change a file (this set of instructions is considered the critical-section of code). For example, if two processes are running, and both need to access the same file, the software needs to be written in a way that these two processes are not executing their critical-sections at the same time. Otherwise, one process may be editing a file that the other process is trying to simultaneously access, causing unpredictable behavior in a program. The critical-section problem can be solved using software that exhibits a mutual exclusion, not allowing multiple processes to execute their critical-sections simultaneously, progress (a thread that wants to execute its critical section can do so when no other thread is executing its own), and bounded waiting. There is a limit to how many times threads can enter their critical-sections before a request to enter by another thread is granted (Silberschatz, Galvin, & Gagne, 2014). The following is an example of a software solution to the critical-section problem provided by our textbook. This example portrays what is known as Peterson’s Solution:

do {

flag[i] = true;

turn = j;

while (flag[j] && turn == j);

critical section

flag[i] = false;

remainder section

} while (true);

This solution alternates execution between processes’ critical sections and remainder sections.

The variable turn represents which process has the right to enter its critical section. The readiness of a process to enter its critical-section is indicated by the flag array (Silberschatz, Galvin, & Gagne, 2014).

****

**Figure 4: Process Execution**

***Week 3 Interactive Assignment:***

**Outline the objectives and functions of memory management in operating systems.**

In computers, effective memory management is extremely important, considering the operating system shares memory space with user processes. To increase performance and optimize CPU utilization, multi-programming, the ability to execute multiple programs simultaneously, is facilitated in many modern computers through concurrent and/or parallel processing (Silberschatz, Galvin, & Gagne, 2014). To achieve successful multi-programming, a memory management system is required in a computer. The objective of the memory management system is to handle allocation, protection, relocation, sharing, logical organization, and physical

organization (Hand, 2010).

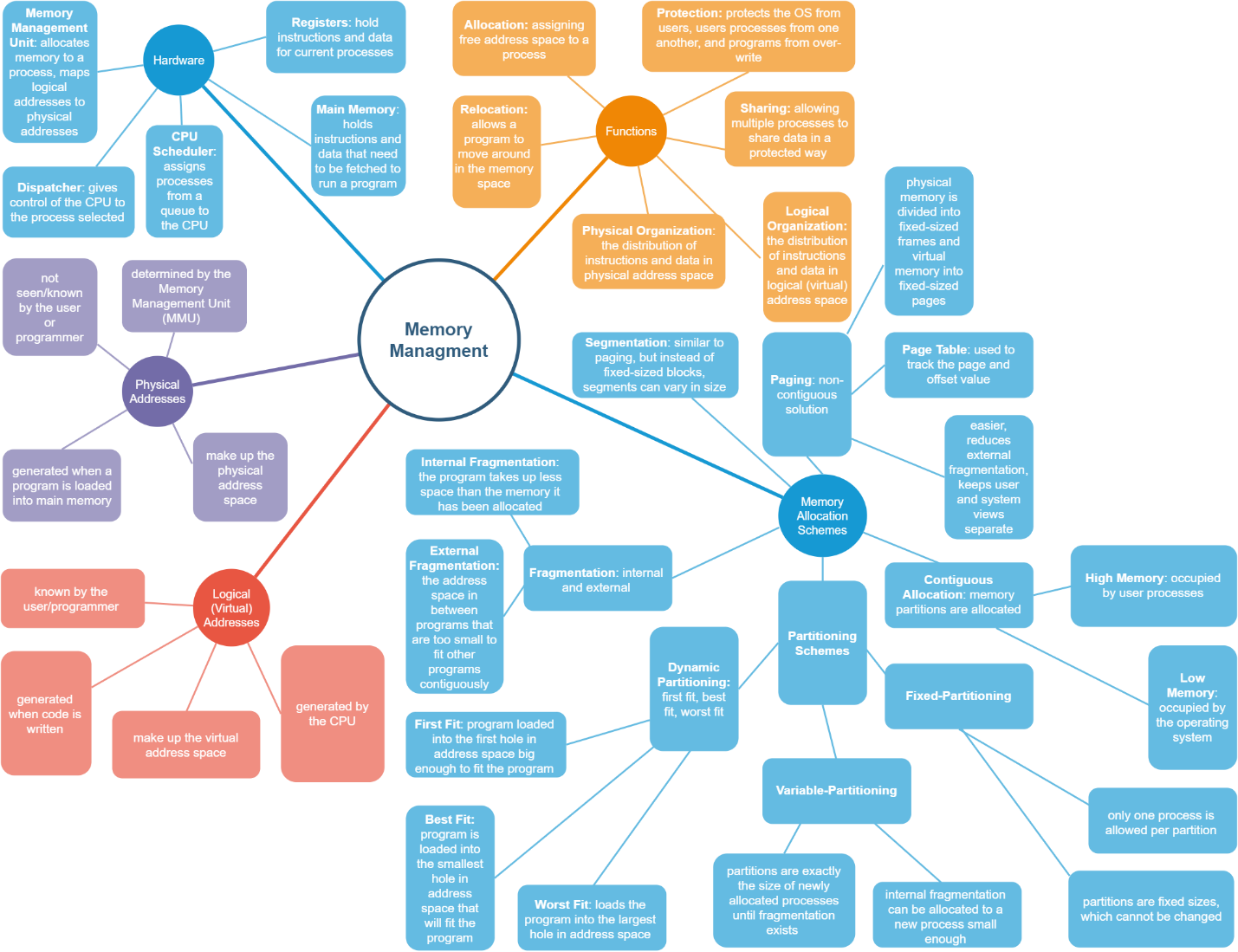
The memory management unit (MMU) must be able to map logical addresses to physical addresses (explained further later). Allocation refers to assigning free memory space to a process. Protection, refers to the operating system being protected from users, user processes are protected from one another, and a program is protected from memory overlaps and over-writing (Staff, n.d.). The functionality of a memory management system that allows a program to be moved around within a memory space to create more free blocks of available memory is relocation. Sharing is the ability for different processes to share data in a protected way. Logical and physical organization is how logical and physical address spaces are mapped to one another, and how processes that occupy a large logical address space can be executed with a smaller physical address space.

**Compare and contrast the physical address space with the virtual address space as they relate to different memory mapping techniques in operating systems.**

One reason for having both logical (virtual) and physical address spaces is, when the code is written, the programmer is unaware of where in memory the program will be loaded (Hand, 2010). Thus, logical addresses generated by the CPU when a program begins execution need to be mapped by the MMU into physical address spaces. This process is called address-binding, and can occur during compiling, loading, or execution. If completed during compiling, absolute code is generated and the code may need to be recompiled before running. If done during loading, the physical address was not known at compile time, so the compiler generated relocatable code and final binding does not occur until load time. If completed during execution, the process was likely moved during execution, so binding does not occur until run-time (Silberschatz, Galvin, & Gagne, 2014). The MMU utilizes base and limit registers while mapping the physical address that correspond to a logical (virtual) address. Base and limit registers define the logical address space and determine the range of addresses for a process. Once the base logical address is offset to calculate the corresponding physical address, the limit register is used to calculate the maximum physical address (Silberschatz, Galvin, & Gagne, 2014).

Memory can be allocated in several ways through the MMU. One model for allocation is contiguous allocation. In this model, memory allocation has two partitions, low memory and high memory. The OS resides in low memory, while user processes occupy high memory. Memory can be separated into fixed-size partitions made of contiguous physical address space, and each partition contains only one process. When a partition frees up, a new process is loaded into the address space (Hand, 2010). Extra space inside a fixed partition (internal fragmentation) cannot be used by a new process. Another model of partitioning is variable partitioning. With variable partitioning, the memory space is one large hole, in which, operations are allocated partitions exactly equal to the size of the process. Once multiple processes are added to the space, holes begin to form in the remaining space as processes finish execution and get removed from the address space, and new processes are added (Silberschatz, Galvin, & Gagne, 2014). Dynamic partitioning is then utilized to allocate processes to the remaining physical address space.

With dynamic partitioning, three common algorithms are used in determining where new processes gets loaded into the address space, first fit, best fit, and worst fit. First fit loads a program into an address space as soon as a hole (or vacancy) large enough to fit the process has been found. Best fit is a method that searches until it finds the smallest hole that will fit a program, and then loads the program. Worst fit is a search algorithm that finds the largest memory hole, and then load the program there. While first fit and best fit are better than worst fit in terms of decreasing time and storage utilization, contiguous allocation can still cause problems such as fragmentation (Silberschatz, Galvin, & Gagne, 2014). Fragmentation is the small fragments of memory in between the contiguous address spaces of other processes that become wasted when they are not large enough to load a new process contiguously (Hand, 2010). With a fixed-partitioning memory allocation scheme, the space created by internal fragmentation within a partition gets wasted, because each partition can contain only one process. In a variable-partitioned environment, internal fragmentation can sometimes be reduced because it can be re-allocated to a new process if the process is small enough. In any case, internal fragmentation can be resolved through compaction (reorganizing memory to create larger holes from the smaller fragments), paging, or segmentation (Silberschatz, Galvin, & Gagne, 2014). Paging and segmentation are memory allocation models that aid in resolving the issue of fragmentation due to utilizing non-contiguous memory addresses. Rather than searching for one large block big enough to fit the whole program, the physical memory is divided into smaller fixed-sized blocks called frames, and logical memory is divided into fixed sized blocks called pages (Hand, 2010). A page tables are used to track the pages in virtual memory and the offset values that correspond to a frame in physical memory. This solution makes memory allocation easier, reducing external fragmentation, and keeping the user and system views of memory usage separate. However, there is added overhead for context switching when a page table entry is not presented for the referenced page, and the page table takes up space in main memory, so there can still be internal fragmentation (when a frame is larger than the address space required for a process) (Hand, 2010). Segmentation is like paging except segments of a process can be varying sizes, rather than being a fixed size.



**Figure 5: Memory Management**

***Week 4 Interactive Assignment:***

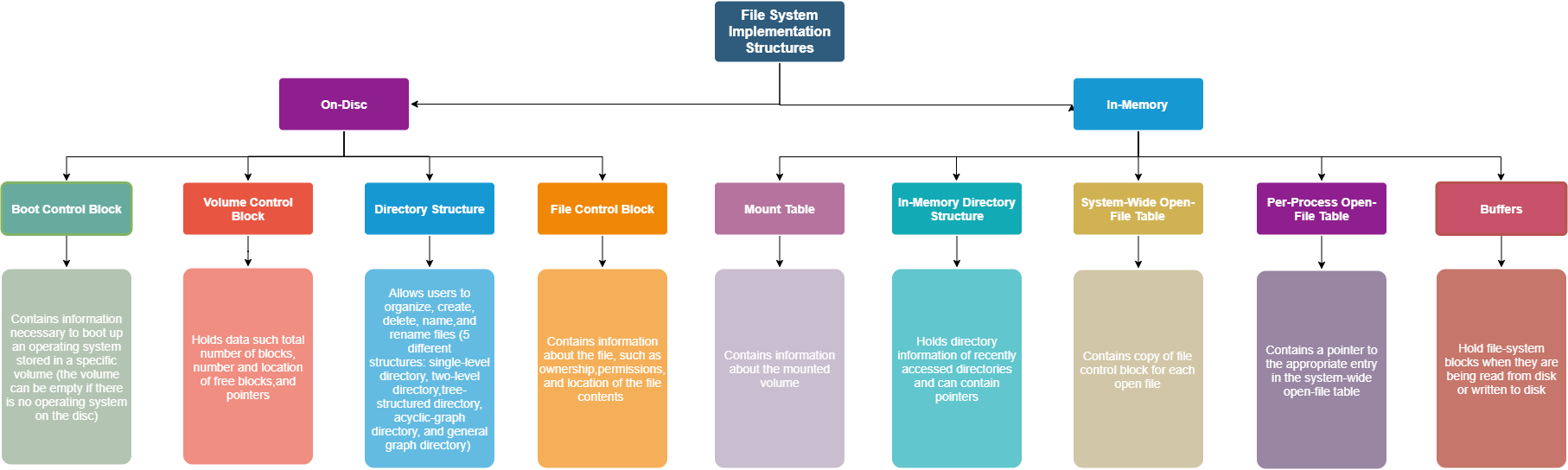
**Outline the objectives and functions of file systems management and the supported operations, including their reliability and performance.**

A file system is a logical storage unit that holds a collection of related information. A file system is in secondary storage, as on a disk, and serves as an interface between the user and the disk where the file system is stored. Two major objectives of the file system are keeping track of where file location, and optimization of the storage and retrieval of data (Columbia University, n.d.). According to Silberschatz, Galvin, and Gagne (2014), “The file system consists of two distinct parts: a collection of files, each storing related data, and a directory structure, which organizes and provides information about all the files in the system.”

A file is the smallest allocation of logical secondary storage. Files can store various types of data including numeric, character, and binary. The creator of the file determines the contents of a file, such as text files, source files, and executable files. In addition to data, files may also contain entire programs. A file system manages file attributes including name, identifier (a unique number used to reference files), type (directing the OS to applications or systems that support that file type), location (a pointer to a file location on secondary storage), size (amount of data stored in the file), protection (determines permissions for reading, writing, and executing files), time, date, and user identification (for usage monitoring) (Columbia University, n.d.).

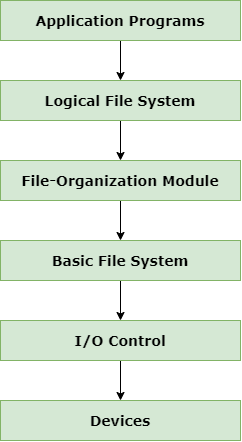
File systems also manage file operations. These operations include creating a new file, writing to a file (or pointer location), reading a file (at pointer location), repositioning within a file (seeking a specific value within the file), deleting a file, truncating a file (erasing the contents of a file, but keeping its attributes), opening a file (searching the directory structure and then pulling it into main memory), closing a file (moving file content from main memory back onto the disk) (Geeks, n.d.). To manage these file operations, the file system controls I/O transfers between the memory and disk in units of blocks (Silberschatz, Galvin, & Gagne, 2014). Specific structures in-memory and on-disk allow these transfers to occur.

On-disk structures that contribute to the implementation of the file system through the OS include: a boot control block (containing the information necessary to boot up an OS stored there), a volume control block (holding data such as the total number of blocks, the number and location of free blocks, and pointers), a directory structure, and a per-file file control block. The file control block contains information about the file, such as ownership, permissions, and location of the file contents (Silberschatz, Galvin, & Gagne, 2014). Regarding the purpose of in-memory structures that aid implementation of the file system, Silberschatz, Galvin, & Gagne (2014) state that, “The in-memory information is used for both file-system management and performance improvement via caching. The data are loaded at mount time, updated during file-system operations, and discarded at dismount. Several types of structures may be included.” These in-memory structures are the mount table (containing mounted volume information), an in-memory directory structure cache (holding directory information of recently accessed directories and may contain pointers), a system-wide open-file table (containing a copy of the file control block for each open file), a per-process open-file table (containing a pointer to the appropriate entry in the system-wide open-file table), and buffers (holding file-system blocks while being read from or written to disk) (Silberschatz, Galvin, & Gagne, 2014).



**Figure 6: File System Implementation Structures**

One requirement of operating systems is the ability to support various types of file systems. One way of accommodating various types would be to write directory and file routines for each type; however, according to Operating System Concepts Essentials (2nd ed.) (2014), most operating systems “use object-oriented techniques to simplify, organize, and modularize the implementation.” File system implementation is broken down into three major layers: the file-system interface (based on the open, read, write, and close calls and on file descriptors), the virtual file system layer (separating file-system-generic operations from their implementation and providing a mechanism for uniquely representing a file throughout a network), and the layer implementing the local or remote-file-system protocol. Silberschatz, Galvin, and Gagne (2014) describe the layered design of a file system in Figure 2 below, in which “Each level in the design uses the features of lower levels to create new features for use by higher levels.” (Silberschatz, Galvin, & Gagne, 2014).

****

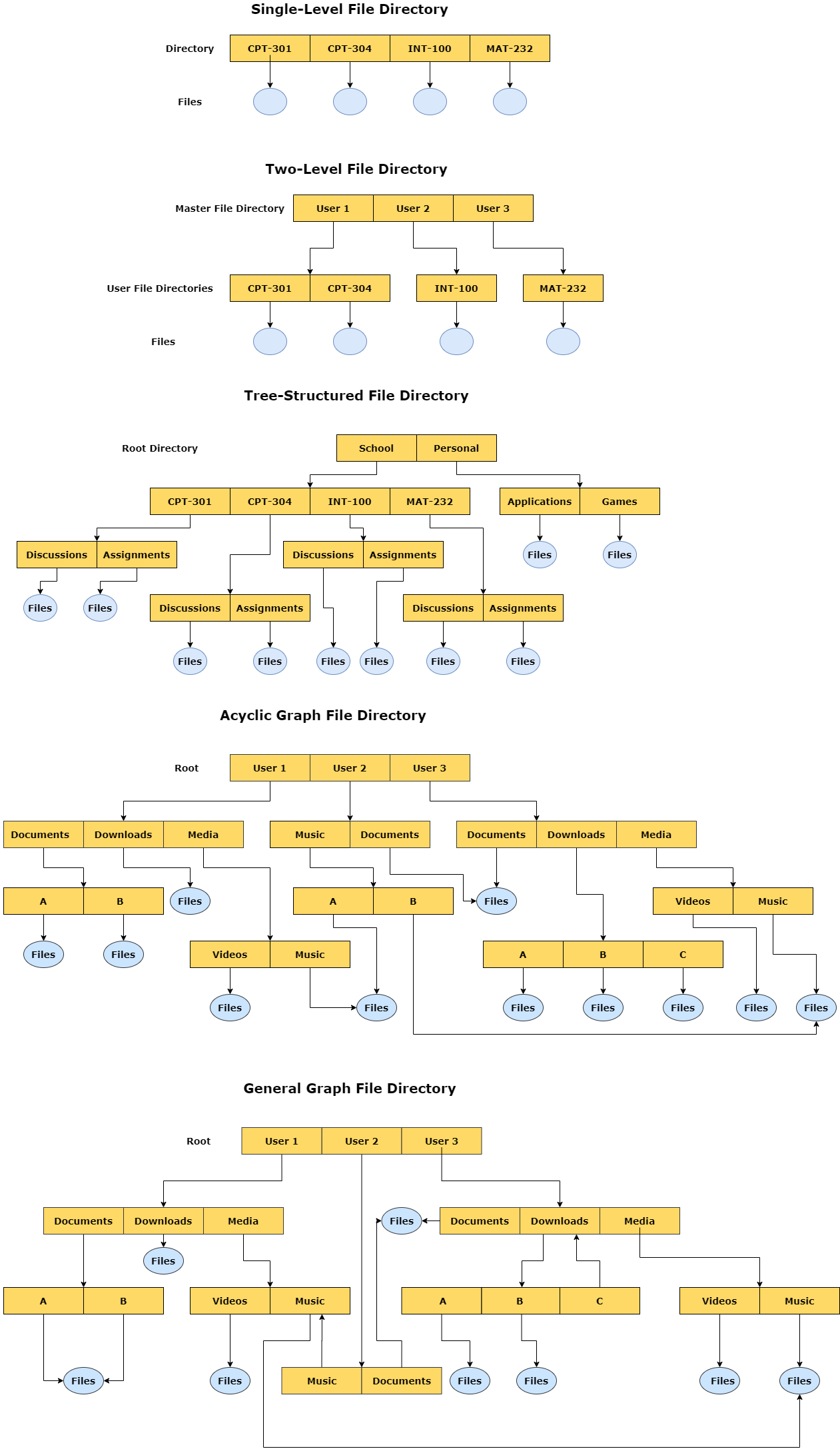
**Figure 7: Layered File System**

**Contrast different directory structures and create unique diagrams to illustrate an example of each directory structure.**

A file system consists of two parts: the collection of files and a directory structure. A file directory is an abstraction from the hardware, which allows users to organize, create, delete, name, and rename files. Users can also exploit the file directory in order to list out the contents of a directory, search for a file, and traverse the files system. There are five common schemes for structuring directories: single-level directory, two-level directory, tree-structured directory, acyclic-graph directory, and general graph directory (Silberschatz, Galvin, & Gagne, 2014). A single-level directory contains all the files in one large directory. This is the simplest scheme but becomes a problem with large numbers of files or multiple users. Since each file needs to have a unique name, and multiple users might name their files similarly, it becomes difficult to remember and locate files.

Providing a partial solution to the challenges of a single-level directory, the two-level directory contains user file directories within the master, so that each user maintains their own file directory. This scheme avoids the issue of multiple users trying to utilize the same file name; however, a disadvantage of this approach is that it isolates users from one another, preventing the sharing of files. The third directory scheme is a tree-structured directory. This approach is an extension of the single- and two-level directories in that users can create as many levels to the tree as desired through the creation of subdirectories. File paths are often used to trace the location of a file within these branching directories. According to Silberschatz, Galvin, and Gagne (2014), “A tree is the most common directory structure. The tree has a root directory, and every file in the system has a unique path name.”

The fourth file directory scheme is an acyclic-graph directory. This approach promotes collaboration and allows users to share a subdirectory in a file system by allowing the subdirectory to exist in two places at once (in both users’ independent directories). In this model, changes made to a shared file by one user will be immediately apparent to the other user when they view the shared file from their own shared subdirectory. In addition, a new file created by one user will be visible in all shared subdirectories. One of the challenges of the acyclic-graph directory is that distinct file names can refer to the same file (a file has multiple absolute path names). Another challenge lies in deletion, since pointers may remain, which then point to a non-existent file. The final directory scheme is the general graph directory. Unlike the acyclic-graph directory, this model allows cycles. While the acyclic-graph directory algorithm for traversing the graph for file references is simple, the general graph directory search algorithm is more complex in that it is designed to avoid searching the same component of the directory twice. This scheme is more flexible, but disadvantages include high cost and the need for garbage collection to manage deleted files (Geeks, n.d.).

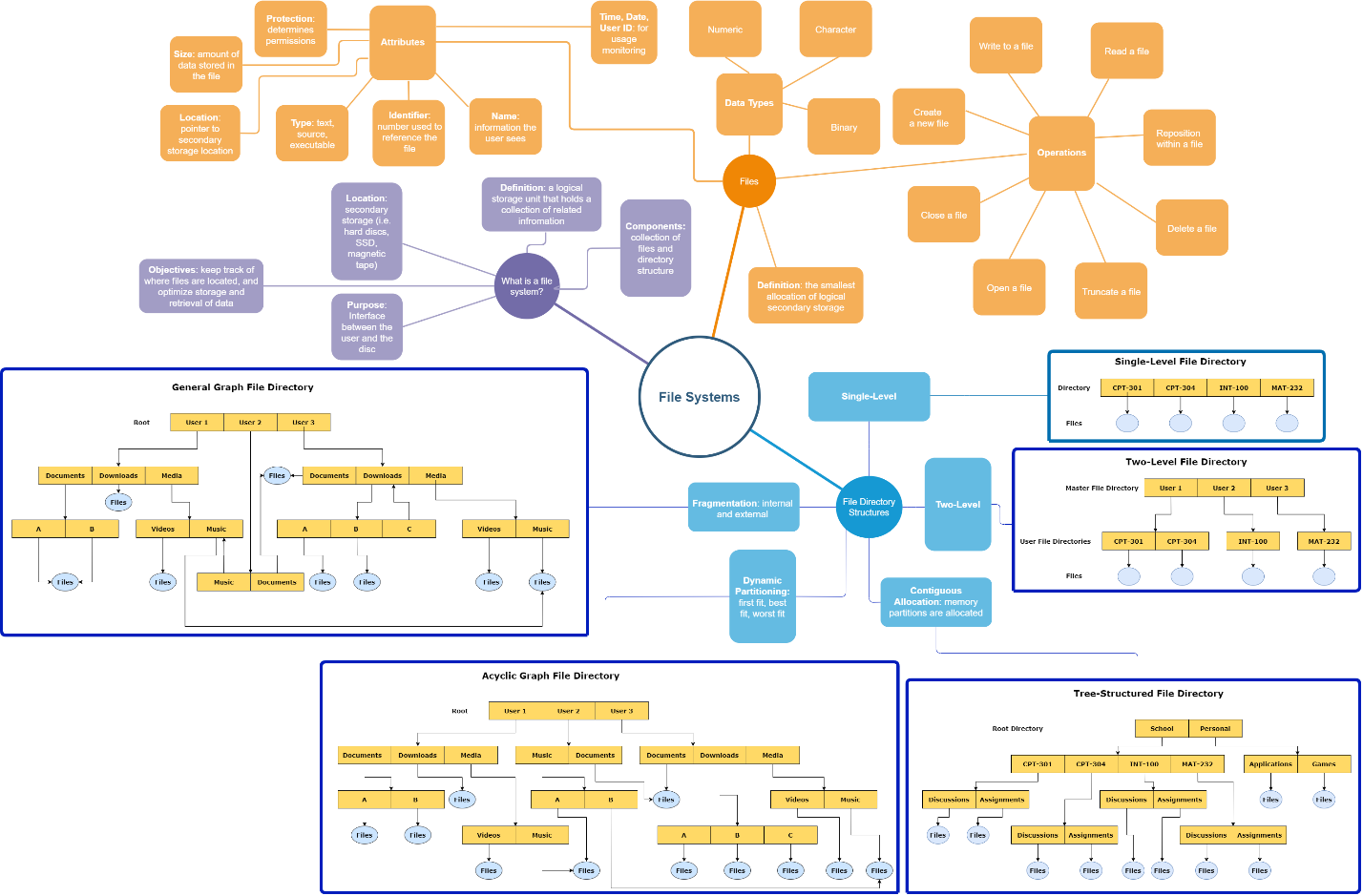


**Figure 8: Directory Structures**

**Describe different types of input/output devices, distinguishing between the hardware and software layers and summarizing the integration across I/O and memory components.**

The operating system provides an abstraction for the two main functions of a computer, I/O and processing. Regarding I/O, the operating systems manages I/O operations and devices. Since I/O devices vary greatly, the methods to control them form an I/O subsystem of their own in the kernel. Even though the range of I/O devices has grown over time, the software/hardware interface across systems has become more standardized. While it seems difficult to integrate new devices with an older system, device drivers help resolve this issue by providing a standardized interface between devices and the I/O subsystem, and system calls provide an interface between applications and the operating system (Silberschatz, Galvin, & Gagne, 2014).

Most I/O devices either serve as storage (CD-ROM’s), modes of transmission (Bluetooth), or user interface devices (mouse, keyboard, etc.). Other I/O devices are specialized. The hardware components of I/O devices consist of buses, device controllers, I/O ports, registers, and the devices themselves. Buses are a group of wires that send messages via electrical voltages according to a specified protocol. Device controllers are electronics that control a port, bus, or device. A port is the connection point between a device and computer. The registers contain data and control signals that allow the processor and controllers to communicate with one another. This is where the hardware/software interface come into play. Specialized I/O instructions initiate buses that select the desired device and move bits in or out of device registers (Silberschatz, Galvin, & Gagne, 2014). Device controllers can employ memory-mapped I/O in which “device-control registers are mapped into the address space of the processor” (Silberschatz, Galvin, & Gagne, 2014). Then I/O requests are executed by the CPU. Computers communicate with I/O devices through signals exchanged either in the air or through cables.



**Figure 9: File Systems Concept Map**

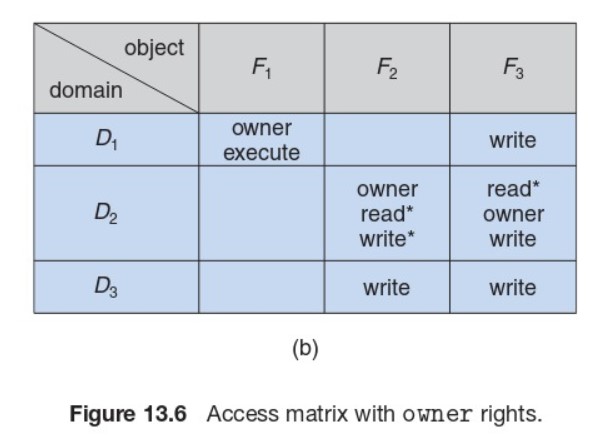
***Week 5 Interactive Assignment:***

**Outline the goals and principles of a domain- and language-based protection in a modern computer system, and describe how an access matrix is used to protect specific resources a process can access.**

Computers need methods to avoid both accidental, as well as malicious, attempts to violate access restrictions. Additionally, strict policies are needed for access to the resources in a computer system. Providing the mechanism for deploying both of these kinds of protection is the operating system. In other words, the operating system solves the protection problem—that each object (such as a file) is accessed correctly, and only by processes (or users) with specific access rights to that object (Silberschatz, Galvin, & Gagne, 2014). One way that the operating system deploys this protection is through the principle of least privilege. This principle states that users should receive the least amount of privileges possible that will allow them to complete the necessary tasks. An example of this type of protection would be creating separate accounts for users, each consisting of customized privileges relative to the user’s role and responsibilities. A second principle on which system protection is based is the “need to know” principle. The basis of this concept is that a process should only be allowed to access resources that it currently requires (Silberschatz, Galvin, & Gagne, 2014).

System protection may further be subdivided into domain-based and language-based. Language-based protection refers to the code written in a programming language specifying policies for the allocation of resources. Language-based protection is deployed through software and is used to initiate system calls corresponding to protection policies to employ resources for an operation (Silberschatz, Galvin, & Gagne, 2014). Domain-based protection works hand-in-hand with language-based protection. Domains consist of objects, such as memory segments, printers, or files, as well as a set of operations that can be performed on those objects. Domains can either be static (unchanging over the life of a process or system) or dynamic (changed as needed). Additionally, domain switching allows a process to switch from one domain to another and allowing for the escalation of privileges as needed (Silberschatz, Galvin, & Gagne, 2014). These characteristics of domains allow flexibility in terms of protection and access.

The ability to perform a specific operation on a given object is known as an access right. In the example below (Figure 13.6), the user associated with domain D1 is the owner of file F1 (an object). Therefore, they possess the access right to execute the file. The user associated with domain D2 is the owner of file F2 and can both read and write to the file. While the user associated with domain D3 can also write to file F2, they have more limited access rights, since they are not the owner of the file. Access rights are determined by the owner, and can include the ability to read, write, append, insert, execute, delete, lock, modify rights, set owner, create group, and add a member to the file (UNC, n.d.). These access rights are controlled through what is known as an access matrix. An access matrix is the mechanism for establishing and referencing access rights for objects in relation to domains. Domains can be users, processes, or procedures. In an access matrix, the access rights of objects are arranged into columns, while domains are arranged into rows (Silberschatz, Galvin, & Gagne, 2014). Figure 13.6: Operating System Concepts Essentials (2nd ed.) (2014).

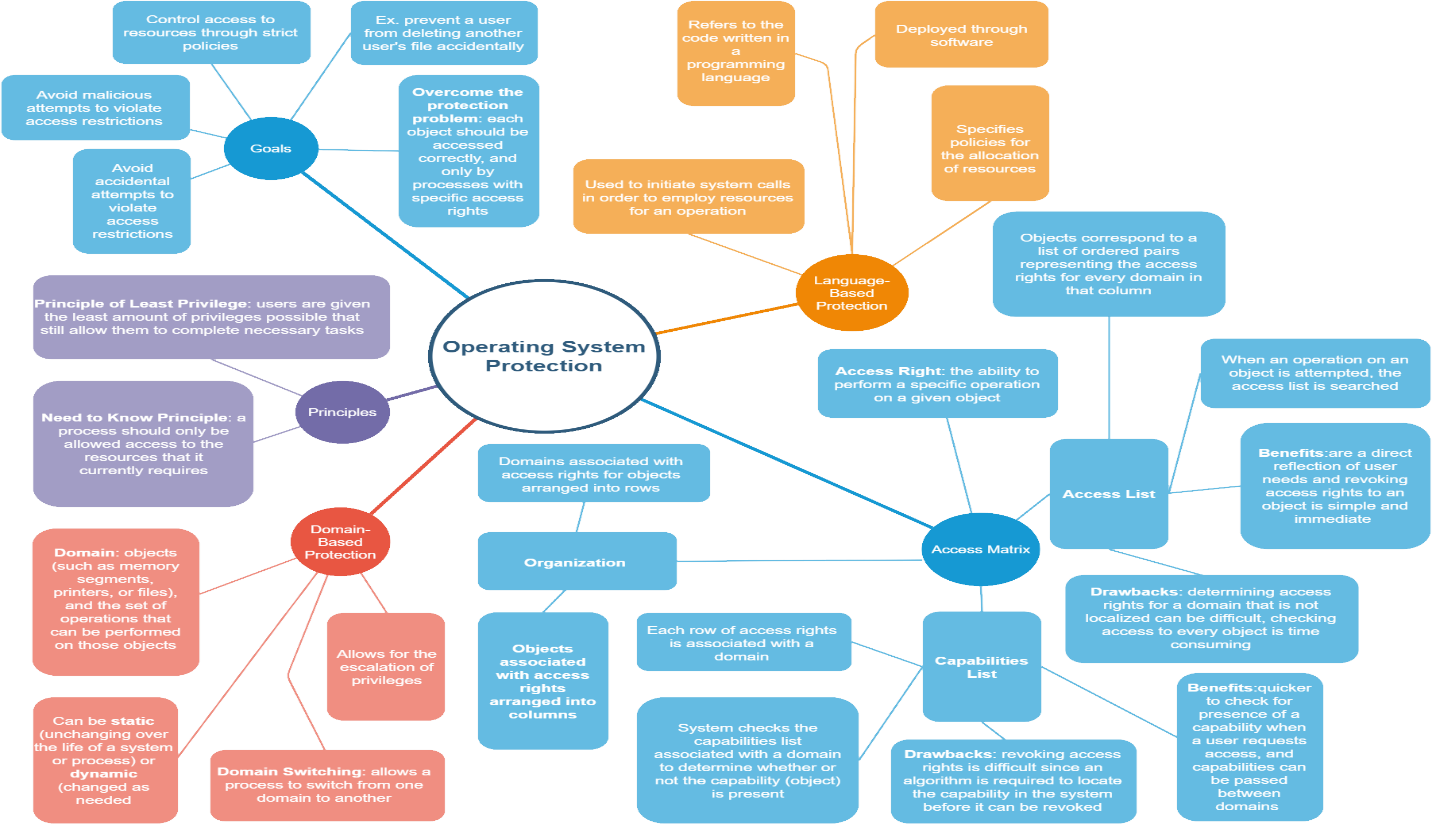


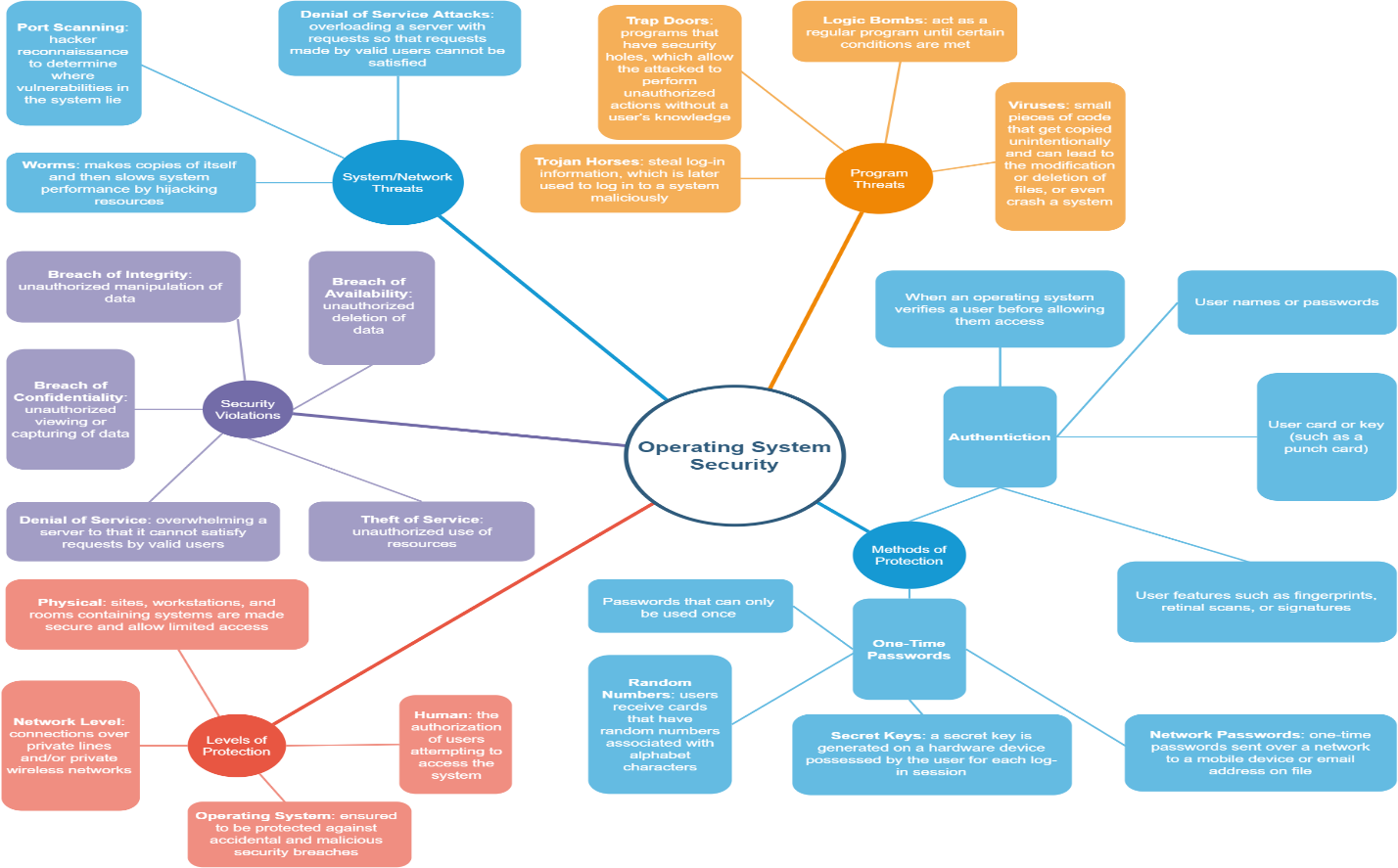
Access matrices can be implemented in two different ways, either through access lists or through capability lists. Implementing an access matrix through access lists refers to a model in which objects (files) correspond to a list of ordered pairs that represent the access rights for every domain falling under the column of that object. When an operation on an object is attempted (i.e., writing to a file), the access list for that file is searched to determine whether or not it contains an access right for the domain. If it does, the operation executes. If there is no access right for the domain found in the access list, then the default set is checked. If it is still not found, the operation is denied (Silberschatz, Galvin, & Gagne, 2014). The other method of access matrix implementation utilizes capability lists. In this model, each row of access rights is associated with a domain. Therefore, if a user wants to write to a file, the system checks the capability list associated with a domain to determine whether the capability (the object) is present. If the domain possesses the capability, the operation can execute. Capability lists are treated as protected objects, and are not directly accessible by the domain (Silberschatz, Galvin, & Gagne, 2014).

**Describe how security is used to protect programs, systems, and networks from threats.**

While protection refers to the internal control of access to data and resources, security aims to guard a system from potential external threats. There are five main categories of security violations that operating systems are designed to protect against. These categories are breach of confidentiality (unauthorized viewing or capturing of data), breach of integrity (unauthorized manipulation of data), breach of availability (unauthorized deletion of data), theft of service (unauthorized use of resources), and denial of service (overwhelming a server so that it cannot satisfy requests by valid users) (Silberschatz, Galvin, & Gagne, 2014). While security threats are organized into these five broad categories, operating systems are often designed to protect against specific threats such as those designed to attack via programs or those which create vulnerabilities in the system or network. Some examples of program threats include Trojan horses (steal log-in information, later used to log in to a system maliciously), trap doors (programs that have security holes, which allow the attacker to perform unauthorized actions without a user’s knowledge), logic bombs (act as regular programs until certain conditions are met), and viruses (small pieces of code that get copied unintentionally and can lead to the modification or deletion of files, or even crash a system). Some examples of system threats include worms (copies itself repeatedly, slowing system performance by hijacking resources), port scanning (hacker reconnaissance to determine where vulnerabilities in the system lie), and denial of service attacks (overloading a server with requests so that requests by valid users cannot be satisfied) (Tutorials Point, n.d.).

To protect against these threats, systems are protected at four different levels: physical (sites, workstations, and rooms containing systems are made secure and allow limited access), human (the authorization of users attempting access to the system), operating system (protected against accidental and malicious security breaches), and at the network level (connections over private lines or private wireless networks) (Silberschatz, Galvin, & Gagne, 2014). In addition, systems often employ authentication processes and one-time passwords to increase security. Authentication refers to the process in which an operating system verifies a user before allowing them access to the system. Methods of authentication can include usernames or passwords, a user card or key (such as a physical punch card), or even user features such as fingerprints, a retinal scan, or a signature. One-time passwords refer to passwords that can only be used once. They can be implemented through random numbers (users receive cards that have random numbers associated with alphabet characters), secret keys (a secret key is generated on a hardware device possessed by the user for each log-in session), and network passwords (one-time passwords sent over a network to a mobile device or email address on file) (Tutorials Point, n.d.). My bank implements one-time passwords when I want to log in to my account on a new device by sending a code to my cell phone.





**Figure 10: Protection and Security Concept Maps**

***Resources:***

Columbia University. (n.d.) Unit 4: Application and System Software. Retrieved from http://www.columbia.edu/itc/hs/medical/dbmi\_hit\_cert/ONC\_Comp\_4/comp4\_unit4- 3\_lecture\_handout.pdf

Franklin, C., & Coustan, D. (2000). Process Control Block in Operating System. Retrieved from https://www.includehelp.com/operating-systems/processcontrol-block-in-operating-system.aspx

Geeks for Geeks. (n.d.). File Systems: Operating Systems. Retrieved from https://www.geeksforgeeks.org/file-system-operating-systems/

Hand, S. (2010). Operating Systems. Retrieved from https://www.cl.cam.ac.uk/teaching/1011/OpSystems/os1a-slides.pdf.

Shukla, A. (2017). Process Control Block in Operating System. Retrieved from https://www.includehelp.com/operating-systems/process-control-block-inoperating-system.aspx

Silberschatz, A., Galvin, P. B., & Gagne, G. (2014). Operating system concepts essentials (2nd ed.). Retrieved from https://redshelf.com/

Introduction to Operating Systems. Retrieved from http://staff.um.edu.mt/csta1/courses/lectures/csm202/os10.html

The Critical Section Problem. Retrieved from http://crystal.uta.edu/~ylei/cse6324/data/critical-section.pdf

TutorialsPoint. (n.d.). Operating System – Memory Management. Retrieved from https://www.tutorialspoint.com/operating\_system/ os\_memory\_management.htm

Tutorials Point. (2018). Operating System - Multi-Threading. Retrieved https://www.tutorialspoint.com/operating\_system/os\_multi\_threading.htm

Tutorials Point. (n.d.). Operating System- Security. Retrieved from https://www.tutorialspoint.com/operating\_system/os\_security.htm

University of North Carolina, Chapel Hill, Department of Computer Science. (n.d.). COMP 730/242 Class Notes: Section 8: Protection. Retrieved from http://www.cs.unc.edu/~dewan/242/s07/notes/prot.PDF