Final Paper

Alicia Piavis

CPT 304 Operating Systems Theory and Design

Sassan Sheedvash

10/13/2018

Week 1 Interactive Assignment: Categorize, describe, and give examples of the major functions of operating systems.

According to *Operating System Concepts Essentials* (2014), the major functions of operating systems include a user interface, program execution, I/O operations, file system management, communication management, error detection, resource allocation, accounting, and protection and security. These functionalities are described as follows. The user interface (UI) is what the user sees and interacts with to direct processes on the computer. The UI can be command line, batch interface, or a graphical user interface (GUI) (Silberschatz, Galvin, & Gagne, 2014). Most consumer devices have GUI's, which can be aesthetically pleasing and interactive.

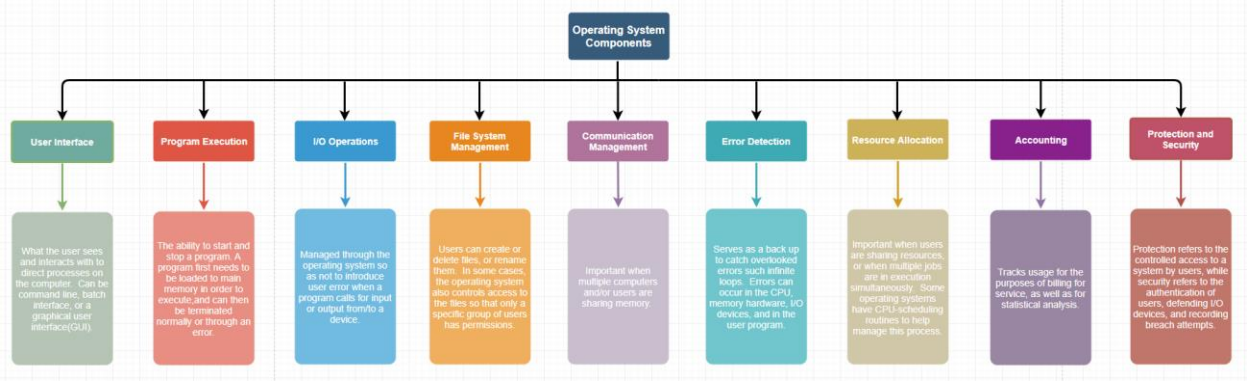
Program execution refers to the ability to start and stop a program. A program first needs to be loaded to main memory in order to execute, and can then be terminated normally or through an error. I/O operations are managed through the operating system so as not to introduce user error when a program calls for input or output from/to a device. In addition, the operating system provides file management capabilities to the user(s) so that they can create or delete files, or rename them. In some cases, the operating system also controls access to the files so that only a specific group of users has permissions (Silberschatz, Galvin, & Gagne, 2014).

The operating system also manages communication between processes. A process is the unit of work in an operating system, and describes a program or job that is currently in execution. Managing communication is important when multiple computers and/or users are sharing memory. In addition, if it weren't for error detection, the performance of a computer could be compromised by a continuous loop. While strategic programming can prevent infinite loops, the operating system serves as a back up to catch overlooked errors such as these. Errors

can occur in the CPU, memory hardware, I/O devices, and in the user program (Silberschatz, Galvin, & Gagne, 2014).

Additionally, the operating system manages resource allocation. This is especially important when users are sharing resources, or when multiple jobs are in execution simultaneously. Some operating systems have CPU-scheduling routines to help manage this process. In addition to resource allocation, accounting is another feature controlled by the operating system. Accounting tracks usage for the purposes of billing for service, as well as for statistical analysis. Lastly, operating systems serve as a barrier in terms of protection and security. While these two terms may seem interchangeable, they actually describe two different concepts. Protection refers to the controlled access to a system by users, while security refers to the authentication of users, defending I/O devices, and recording breach attempts (Silberschatz, Galvin, & Gagne, 2014).

Link to concept map: Operating System Components



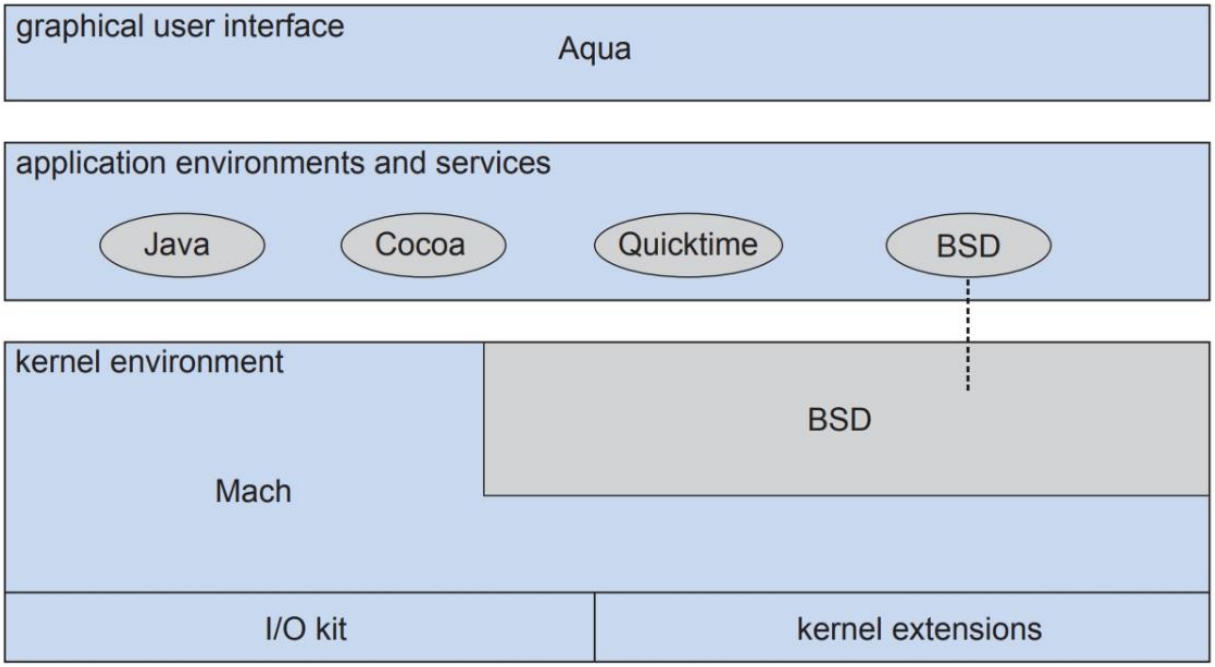
Illustrate the hierarchy of subsystems, components, or subcomponents of operating systems and explain how they interact with one another.

Operating systems can vary in design, depending on the approach implemented for the organization of the components previously described. The most rudimentary design is called

monolithic, and refers to the fact that all of the functionalities of the operating system are bundled into one space, rather than being separated into multiple levels, like a hierarchy. Modern operating systems tend to organize the components into smaller subsystems or subcomponents called modules. This organization into levels prevents vulnerability in the system. For example, "application programs are able to access the basic I/O routines to write directly to the display and disk drives" (Silberschatz, Galvin, & Gagne, 2014), which can cause errors or overwrite important data.

Many operating systems today entail a hybrid design, meaning that they draw from multiple approaches of component organization. The benefit of dividing the operating system into levels, and designing certain components to be modular allows for added functionality and modifiability of the system. For example, the Apple Mac OS X uses a layered system approach. The top layer is the GUI. Then there is a layer for the applications and services. Examples of applications include Java and Quicktime. Below the application and service layers is the kernel, and an additional layer includes the I/O kit and kernel extensions (Silberschatz, Galvin, & Gagne, 2014). The layered approach of the Apple Mac OS X system supports the addition of modules, which can increase functionality. Figure 2.16 from *Operating System Concepts Essentials* (2014) does a great job of visually portraying the subsystems of the Apple Mac OS X operating system.

Figure 2.16: Operating System Concepts Essentials (2014)



Week 2 Interactive Assignment: Describe process, process state, and process control block.

According to *Operating System Concepts Essentials (2nd ed.)* (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 have been fetched from memory with the intention of execution, along with possible interaction from the user. A process can be categorized into different stages at any particular moment in time. The classification of the stage is known as the process state. When a new process is created, the state of that process is "new." For example, when a user clicks on the icon for the Microsoft Word Application, the state of the first process initiated is "new". When the instructions for the program are being executed, the process state is "running." In the Word application 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.). As the application waits for user input, the state of the process is

"waiting." When the process (creating a new Word document) is waiting to be assigned to a processor, the process state is "ready." After execution of the program (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 of the resources that were being used during execution are freed up (Silberschatz, Galvin, & Gagne, G., 2014).

A process control block (PCB) is a data structure that stores temporary data relevant to the state of a process. Each process has its own process control block, and all process control blocks are maintained in a process table by the operating system (Franklin & Coustan, 2000). The process control block is extremely important in multiprocessing, since the execution of processes often switch back and forth so that the CPU is never idle (which leads to increased performance). Going 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. Therefore, the process would get added to the I/O queue at that point, freeing up the CPU for other processes that are running simultaneously. However, 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 operating system relies on the process control block 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.

A thread is "a flow of control within a process." (Silberschatz, Galvin, & Gagne, 2014). Multi-threaded models can be 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 through the use of 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 less resources, since the threads share code, data, and files. In addition, a single-threaded model requires interactions with the operating system in order to employ switches, whereas operating system interactions are not required in a multi-threaded model. Lastly, a single-threaded model can cause stalls when a process gets blocked. On the other hand, if a thread gets blocked, other threads for the same task can continue to run (Tutorials Point, 2018).

While a multi-threaded approach can seem much more advantageous than a single-threaded approach, there are challenges to running 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. In addition, they are faster to create. Kernel-level threads are created and managed by the operating system and are slower to create (Tutorials Point, 2018). Knowledge of the user and kernel levels is important in grasping 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. In 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 can run in parallel, and according to *Operating System - Multi-Threading* (2018), "this model provides the best accuracy on concurrency." Another model is Many to One. In this model, many user threads multiplex with a single kernel thread. Therefore, if a thread gets blocked, the whole system has to wait until it becomes unblocked. The last multi-threaded model is One to One, 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 at some point or another 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. This can cause unpredictable behavior in a program. The critical-section problem can be solved with software that exhibits mutual exclusion (does not allow 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 *Operating system concepts essentials (2nd ed.)* (2014). This example portrays what is known as Peterson's Solution:

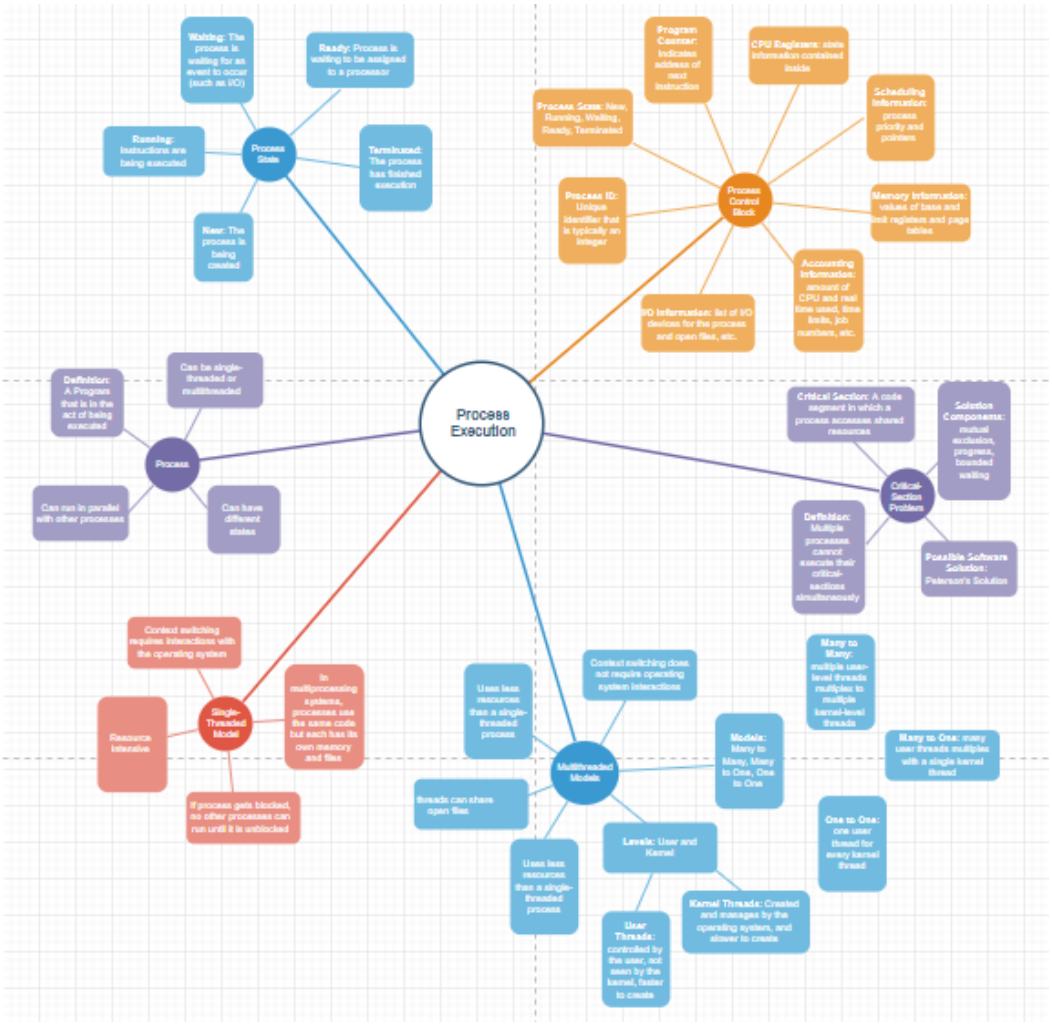
```
do {
        flag[i] = true;
        turn = j;
        while (flag[j] && turn == j);

                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).

Link to concept map: [Process Execution]

Process Execution Concept Map



Week 3 Interactive Assignment: Outline the objectives and functions of memory management in operating systems.

Effective memory management in computers is extremely important, since the operating system shares memory space with user processes. In order to increase performance and optimize utilization of the CPU, many modern computers facilitate multi-programming, the ability to execute multiple programs simultaneously, through concurrent and/or parallel processing

(Silberschatz, Galvin, & Gagne, 2014). In order to achieve successful multi-programming, a computer needs a memory management system. The objective of the memory management system is to handle relocation, allocation, protection, sharing, logical organization, and physical organization (Hand, 2010).

Relocation refers to 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. In addition, the memory management unit (MMU) must be able to map logical addresses to physical addresses (later explained in further detail). Allocation refers to assigning free memory space to a process. Protection refers to the fact that the operating system should be protected from users, user processes are protected from one another, and a program is protected from memory overlaps and over-writing (Staff, n.d.). Sharing refers to the ability of different processes to share data in a protected way. Logical and physical organization refers to how logical and physical address spaces are mapped to one another, as well as 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.

Part of the reason for having both logical (virtual) and physical address spaces is that when the code is written, the programmer does not know where in memory the program will be loaded (Hand, 2010). Therefore, logical address spaces generated by the CPU when a program begins execution need to be mapped by the MMU into physical address spaces. This process is known as address-binding, and can occur during compile time, load time, or execution time. If completed at compile time, absolute code is generated and the code may need to be recompiled

before running. If completed at load time, the physical address was not known at compile time, so the compiler generates relocatable code and final binding does not occur until load time. If completed during execution time, this means that the process was likely moved during execution, so binding does not occur until run-time (Silberschatz, Galvin, & Gagne, 2014). In order to map the physical address that corresponds to a logical (virtual) address, the MMU utilizes base and limit registers. 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 a number of ways through the memory management unit. One model of allocation is contiguous allocation. In this model, memory allocation has two partitions: low memory and high memory. The operating system is in low memory. 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 is free, 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. In this model, the memory space is one large hole, and 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). At this point, dynamic partitioning is utilized to allocate processes to the remaining physical address space.

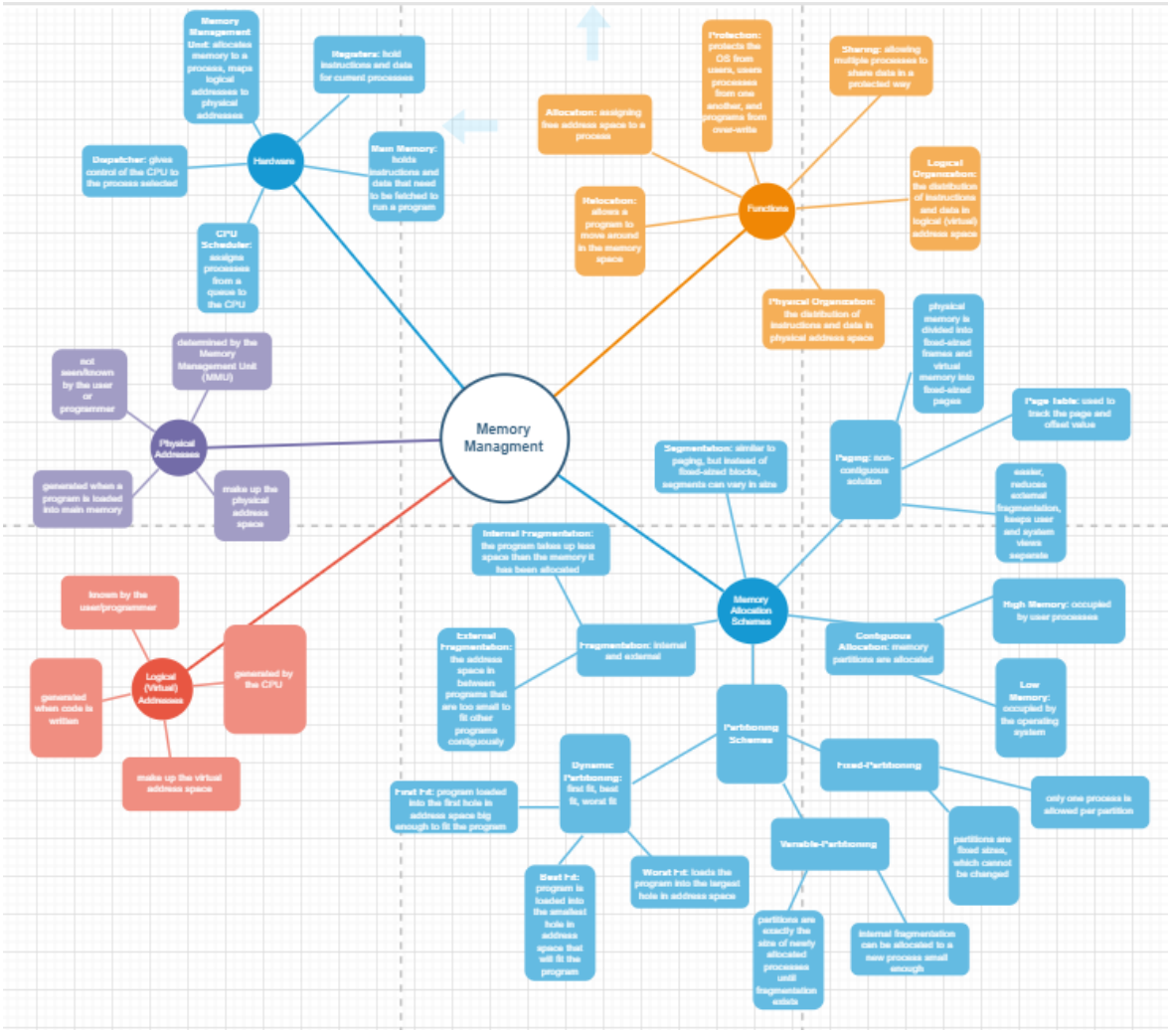
In dynamic partitioning, there are three common algorithms used to determine where a new process gets loaded into the address space: first fit, best fit, and worst fit. First fit loads a program into an address space as soon as a hole (or vacancy) large enough to fit the process has been found. Best fit refers so 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 aims to find 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 refers to the small fragments of memory in between the contiguous address spaces of other processes, which become wasted when they are not large enough to load a new process contiguously (Hand, 2010). In a fixed-partitioning memory allocation scheme, the space created by internal fragmentation within a partition gets wasted, since each partition can contain only one process. In a variable-partitioned environment, internal fragmentation can sometimes be reduced due to the fact that it can be re-allocated to a new process if the process is small enough. Either way, 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 help resolve the issue of fragment due to the fact that they utilize non-contiguous memory addresses. Instead of 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 the logical memory is divided into fixed sized blocks called pages (Hand, 2010). A page table is used to track the page in virtual memory and the offset value that corresponds to a frame in physical memory. This solution makes memory

allocation easier, reduces external fragmentation, and keeps the user and system views of memory usage separate. However, the page table takes up space in main memory, there can still be internal fragmentation (when a frame is larger than the address space required for a process), and there is added overhead for context switching when a page table entry is not presented for the referenced page (Hand, 2010). Segmentation is similar to paging except that instead of all segments being a fixed size, segments of a process can be varying sizes.

Link to concept map: [Memory Management]

Memory Management Concept Map



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. The file system resides in secondary storage, such as on a disc, and thus serves as an interface between the user and the disc on which a file system is stored. The two major objectives of a file system are to keep track of where files are located, and to optimize the storage and retrieval of data (Columbia University, n.d.). According to Silberschatz, Galvin, & 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, and files can store various types of data including numeric, character, and binary. The file creator determines the contents of a file. Examples of file types include text files, source files, and executable files. Besides data, files can also contain entire programs. A file system manages file attributes including name (the information that the user sees, which is used to select a file), identifier (a unique number that is used to reference files in the file system), type (directs the operating system to applications or systems that support that file type), location (the pointer to a file location on secondary storage), size (the 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.).

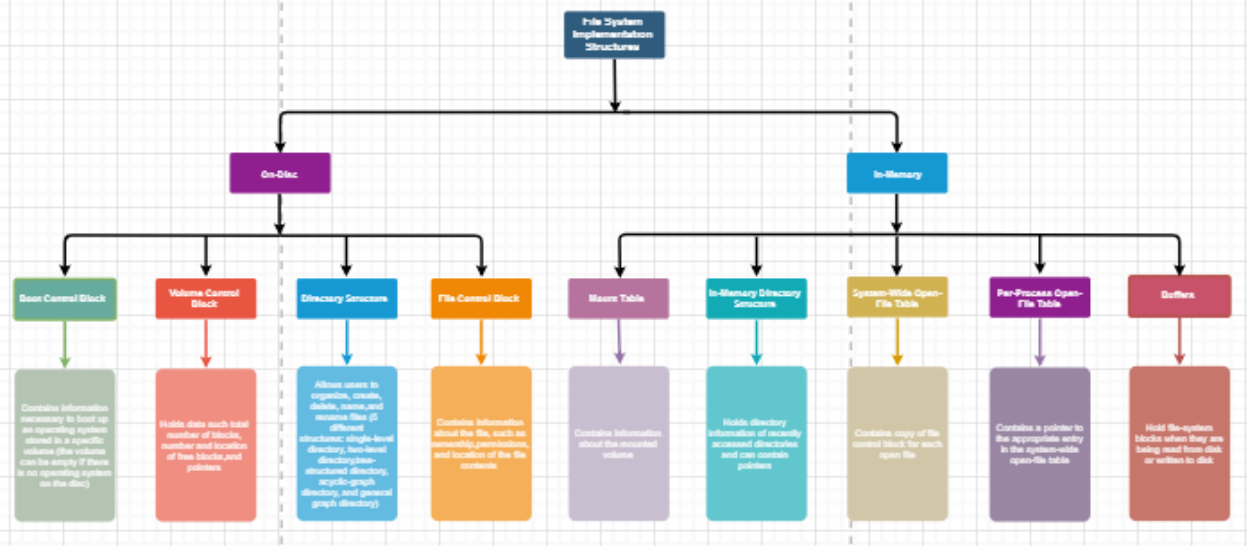
File systems also manage file operations. These operations include creating a new file, writing to a file (at write pointer location), reading a file (at read pointer location), repositioning within a file (seeking a specific value within the file), deleting a file, truncating a file (erasing the contents of a file, but keeping its attributes), opening a file (searching the directory structure on

the disc for the desired content and then pulling it into main memory), closing a file (moving file content from main memory back onto the disc) (Geeks, n.d.). In order 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-disc allow these transfers to occur.

On-disc structures that contribute to implementation of the file system through the operating system include: a boot control block (contains information necessary to boot up an operating system stored there), a volume control block (holds data such total number of blocks, number and location of free blocks, and pointers), a directory structure (discussed later), 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). In regards to 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 (contains mounted volume information), an in-memory directory structure cache (holds directory information of recently accessed directories and can contain pointers), a system-wide open-file table (contains copy of file control block for each open file), a per-process open-file table (contains a pointer to the appropriate entry in the system-wide open-file table), and buffers (hold file-system blocks when they are being read from disk or written to disk) (Silberschatz, Galvin, & Gagne, 2014).

Link to File System Implementation Structures concept map: [File System Implementation Structures]

File System Implementation Structures Concept Map



One of the requirements of operating systems is the ability to support various types 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 thus 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 (separates file-system-generic operations from their implementation and provides a mechanism for uniquely representing a file throughout a network), and the layer implementing the local or remote-file-system protocol. Silberschatz, Galvin, & Gagne (2014) describe the layered design of a file system in Figure 11.1 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 11.1: Layered File System

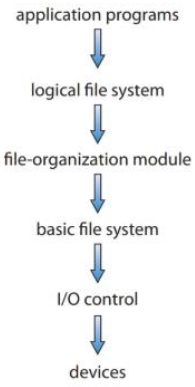


Figure 11.1 Layered file system.

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

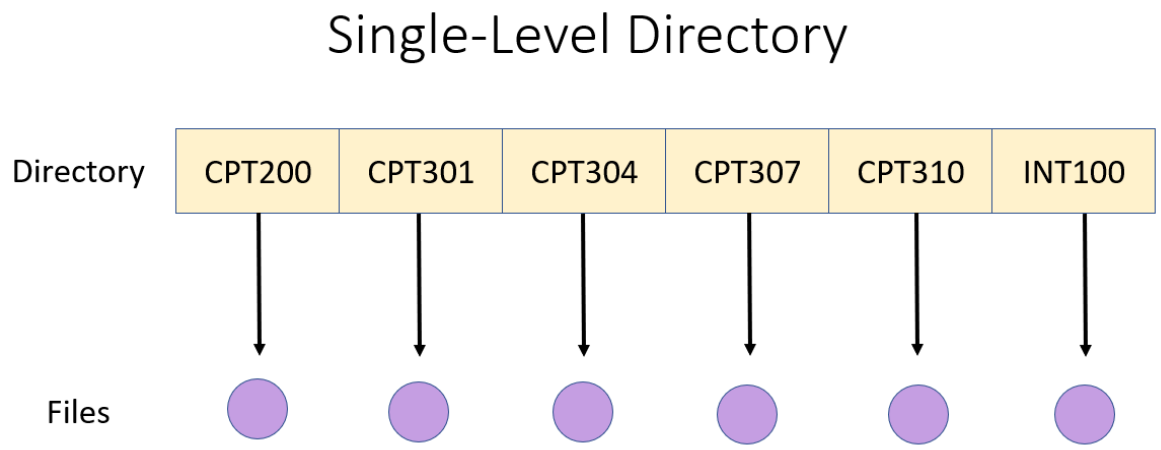
As mentioned previously, a file system consists of two parts: the collection of files and a directory structure. A file directory is an abstraction from the hardware, which allows users to organize, create, delete, name, and rename files. Users can also exploit the file directory in order to list out 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 of the files in one large directory. This is the simplest scheme, but becomes problematic 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.

The two-level directory provides a partial solution to the challenges of a single-level directory in that a two-level directory contains user file directories within the master, so that each

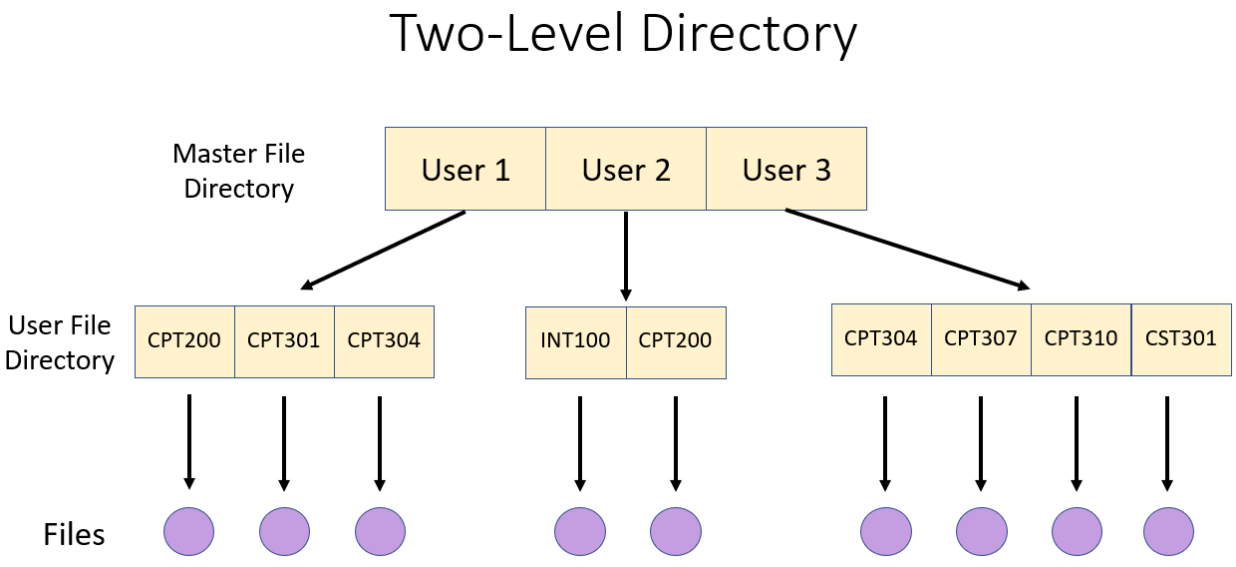
user maintains their own file directory. This scheme avoids the issue of multiple users trying to utilize the same file name. However, a disadvantage of this approach is that it isolates users from one another, preventing the sharing of files. The third directory scheme is a tree-structured directory. 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, & 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.).

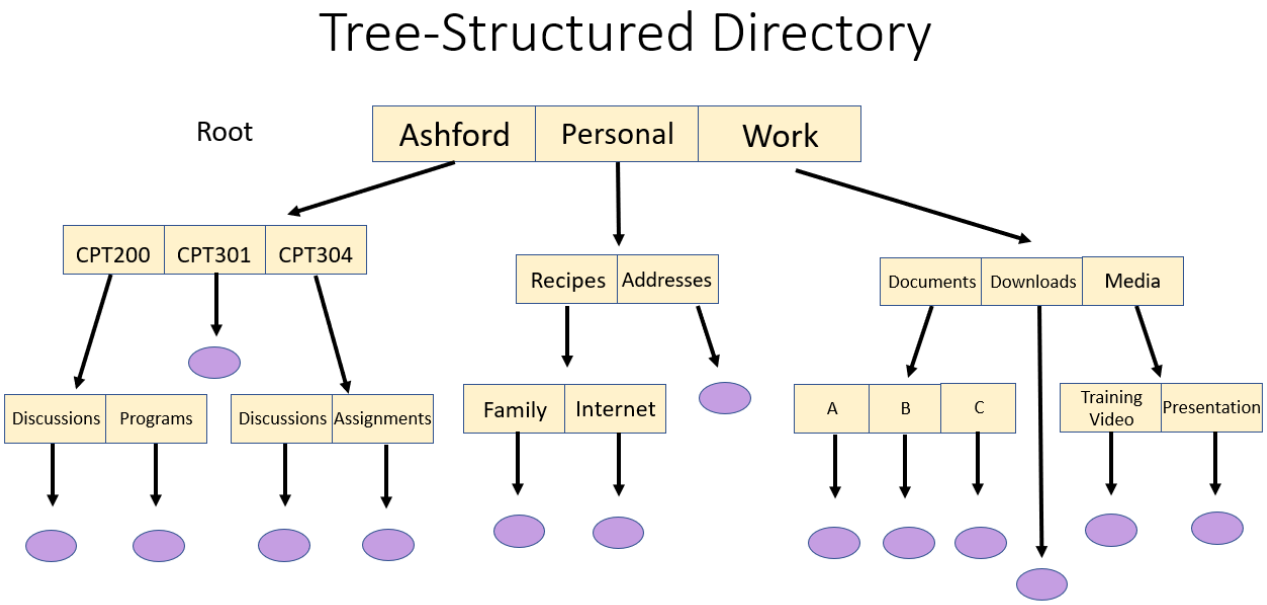
Single-Level File Directory Diagram



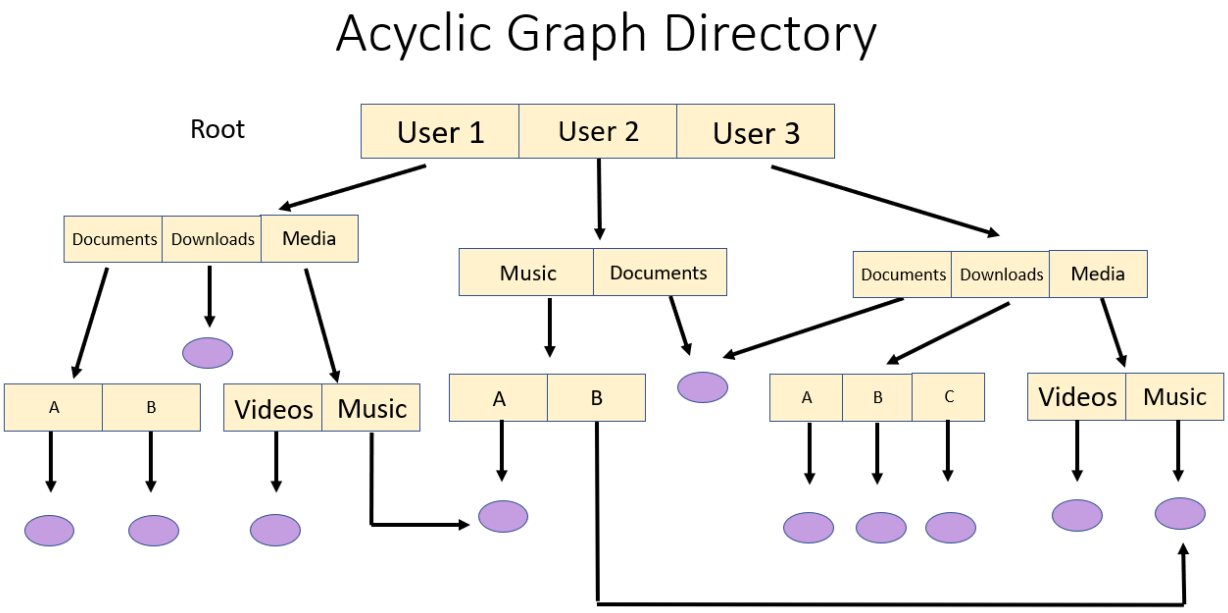
Two-Level File Directory Diagram



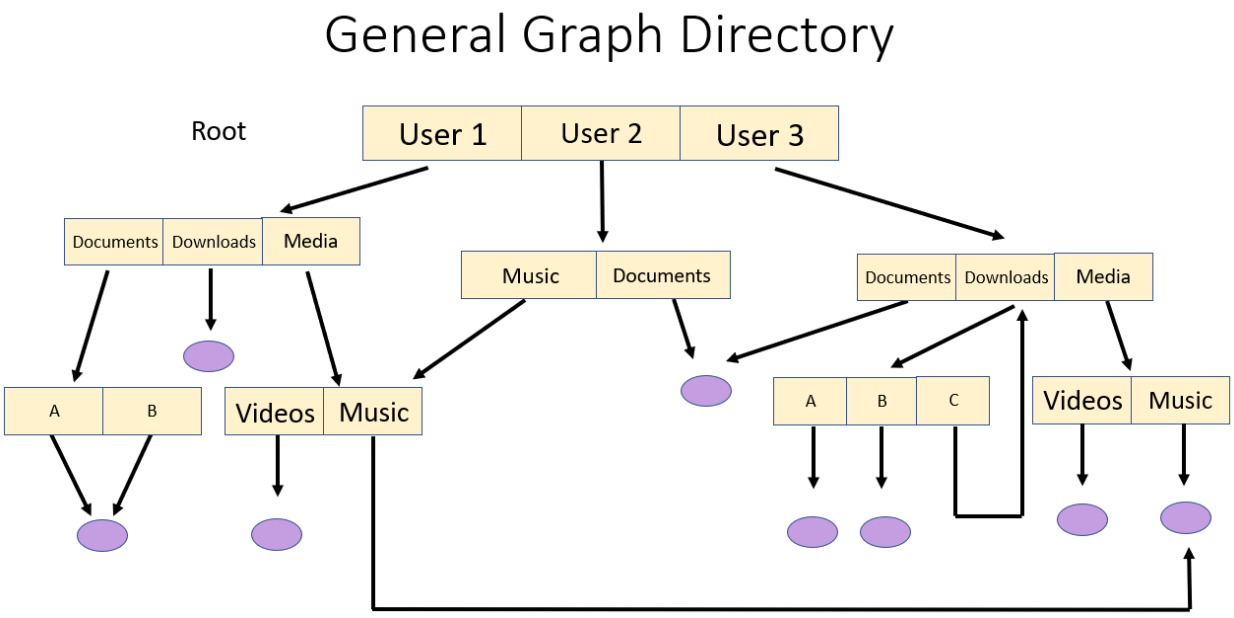
Tree-Structured File Directory Diagram



Acyclic Graph File Directory Diagram



General Graph File Directory Diagram



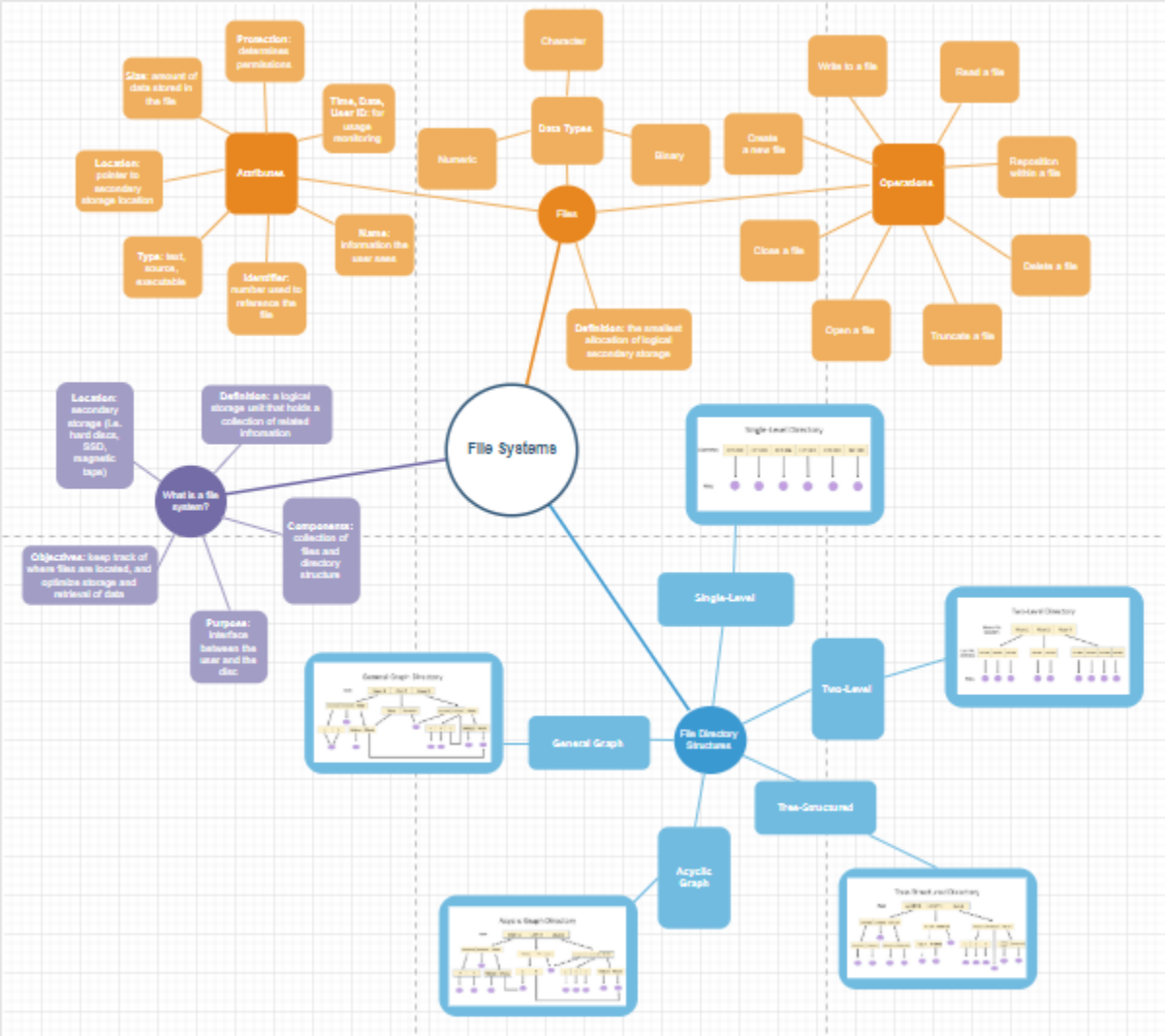
Describe different types of input/output devices, distinguishing between the hardware and software layers and summarizing the integration across I/O and memory components.

The operating system provides an abstraction for the two main functions of a computer: I/O and processing. In regards to 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. So, while it may seem 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, such as the Energizers that my husband's old company produces, which limit the powering on of machines in industrial areas to employees who hold specific trainings and certifications, in order to promote industrial safety. The hardware components of I/O devices consist of buses, device controllers, I/O ports, registers, and the devices themselves. A bus is a group of wires that sends messages via electrical voltages according to a specified protocol. A device controller is an electronic that controls 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 comes 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). Or 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.

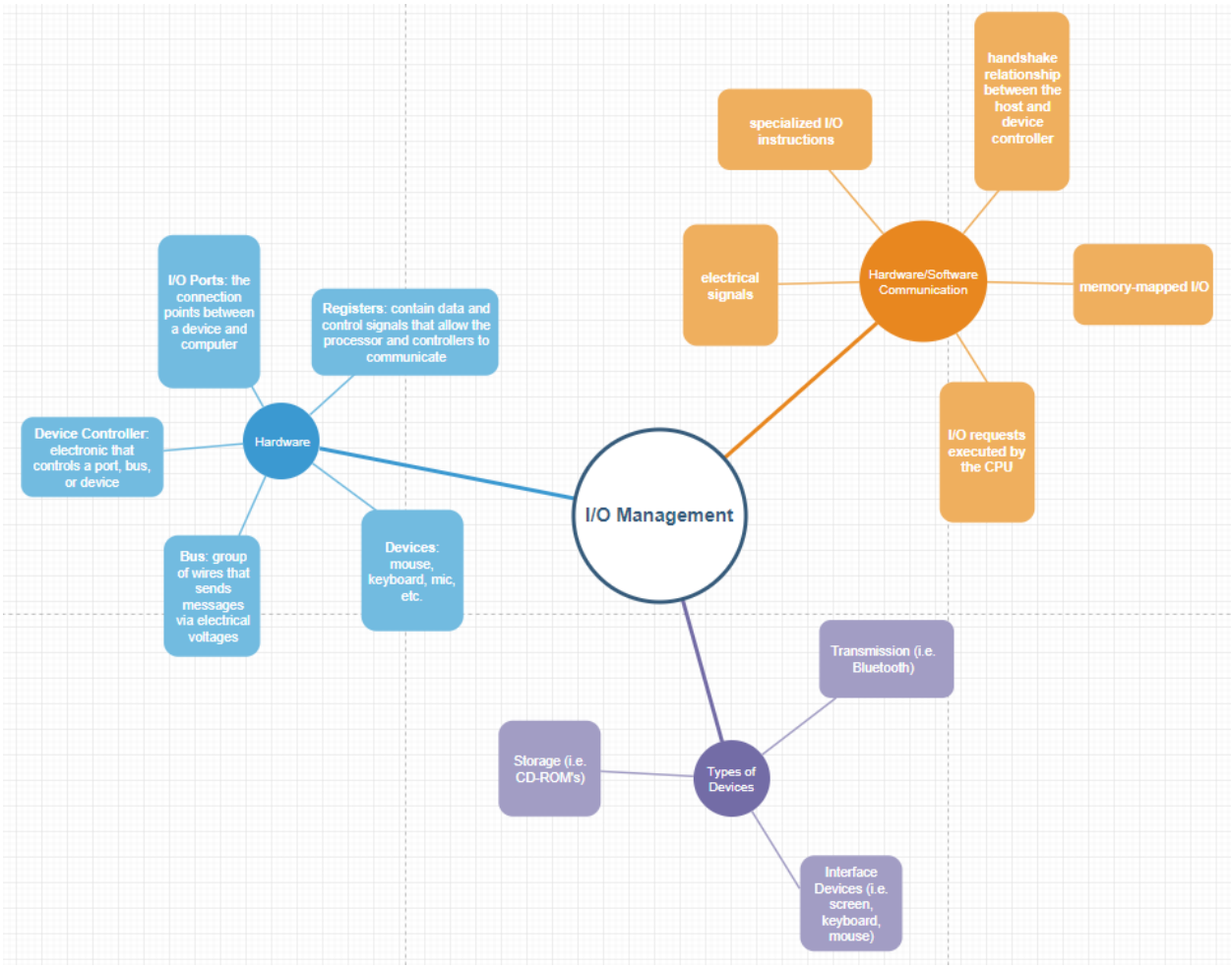
Link to File Systems concept map: [File Systems] (*please click on “Open with drawi.io Diagrams”, since preview is incorrect)

File Systems Concept Map



Link to I/O Management concept map: I/O Management

I/O Management Concept Map



Week 5 Interactive Assignment: Outline the goals and principles of 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 require a method to ensure that both accidental, as well as malicious attempts to violate access restrictions are avoided. In addition, access to the resources in a computer system needs to be controlled through strict policies. The operating system provides a mechanism for deploying both of these kinds of protection. In other words, the operating system

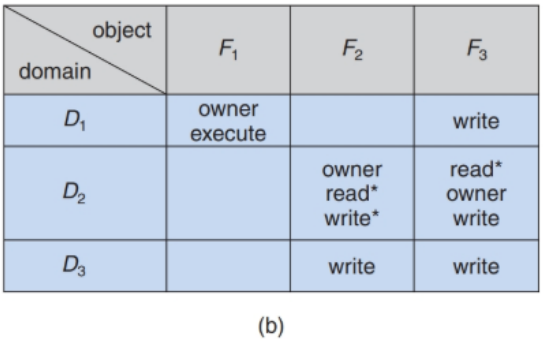
solves the protection problem—that each object (such as a file) is accessed correctly, and only by processes (or users) with specific access rights to that object (Silberschatz, Galvin, & Gagne, 2014). One of the ways in which the operating system deploys this protection is through the principle of least privilege. This principle states that users should be given the least amount of privileges possible that still allow them to complete necessary tasks. An example of this type of protection would be creating separate accounts for users, each of which has 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. This concept is based on the fact that a process should only be allowed to access resources that it currently requires (Silberschatz, Galvin, & Gagne, 2014).

System protection can be subdivided into domain-based and language-based. Language-based protection refers to the code written in a programming language which specifies policies for the allocation of resources. Language-based protection is deployed through software and is used to initiate system calls corresponding to protection policies in order to employ resources for an operation (Silberschatz, Galvin, & Gagne, 2014). Domain-based protection works hand-in-hand with language-based protection. A domain consists 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). In addition, domain switching allows a process to switch from one domain to another, and domains allow 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 D_1 is the owner of file F_1

(an object). Therefore, they possess the access right to execute the file. The user associated with domain D_2 is the owner of file F_2 and can both read and write to the file. While the user associated with domain D_3 can also write to file F_2, 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)



object / domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		owner read* write*	read* owner write
D_3		write	write

(b)

Figure 13.6 Access matrix with `owner` rights.

Access matrices can be implemented in two different ways: through access lists, and 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.

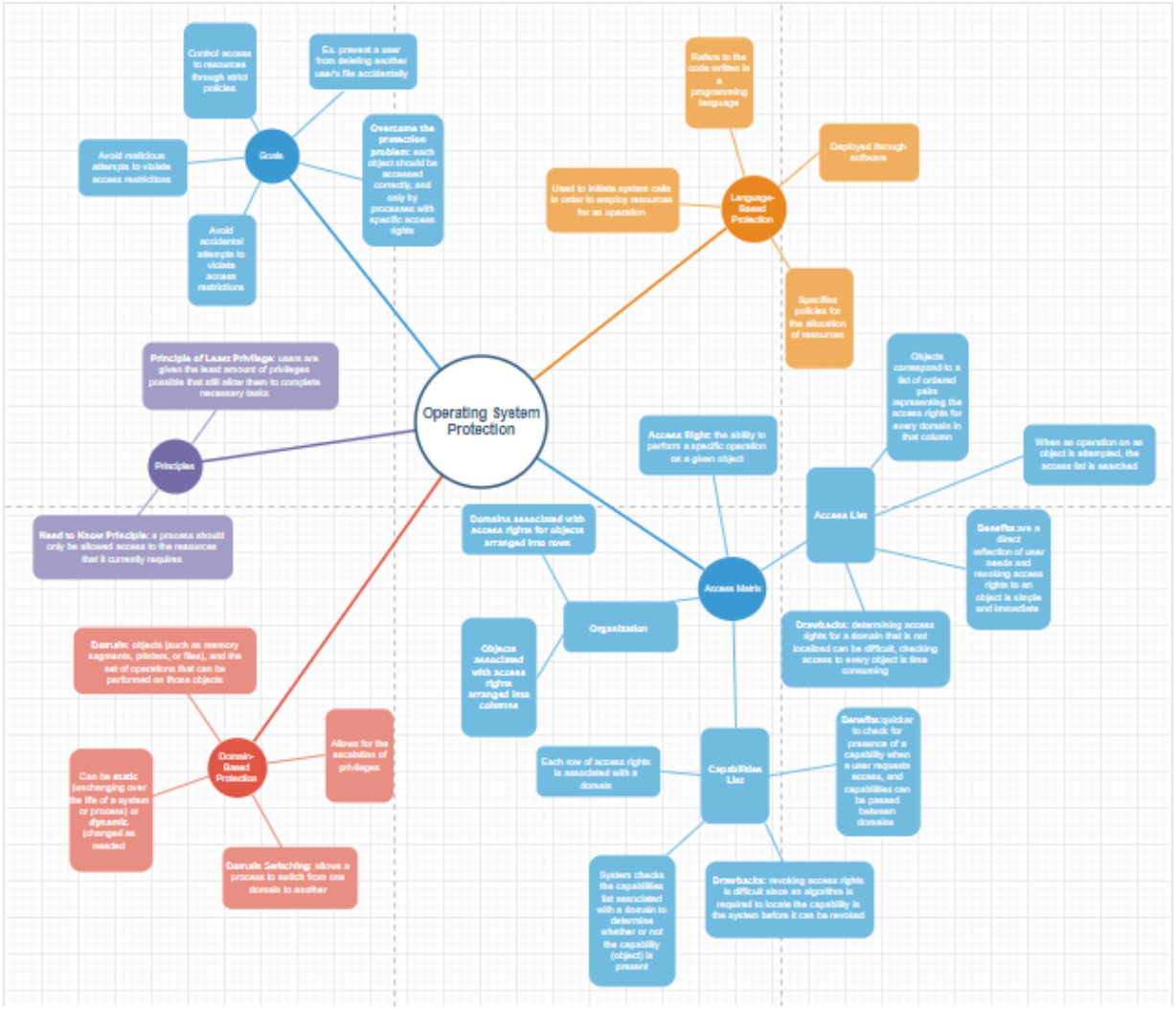
While protection refers to the internal control of access to data and resources, security aims to guard a system from potential external threats. There are five main categories of security violations that operating systems are designed to protect against: breach of confidentiality (unauthorized viewing 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 a program, or those which create vulnerabilities in the system or network. Examples of program threats include trojan horses (steal log-in information which is 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). Examples of system threats include worms (makes copies of itself and then slows 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.).

In order 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.). At work my company employs authentication by requiring us to enter a user password each time we log in to our computer, and 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.

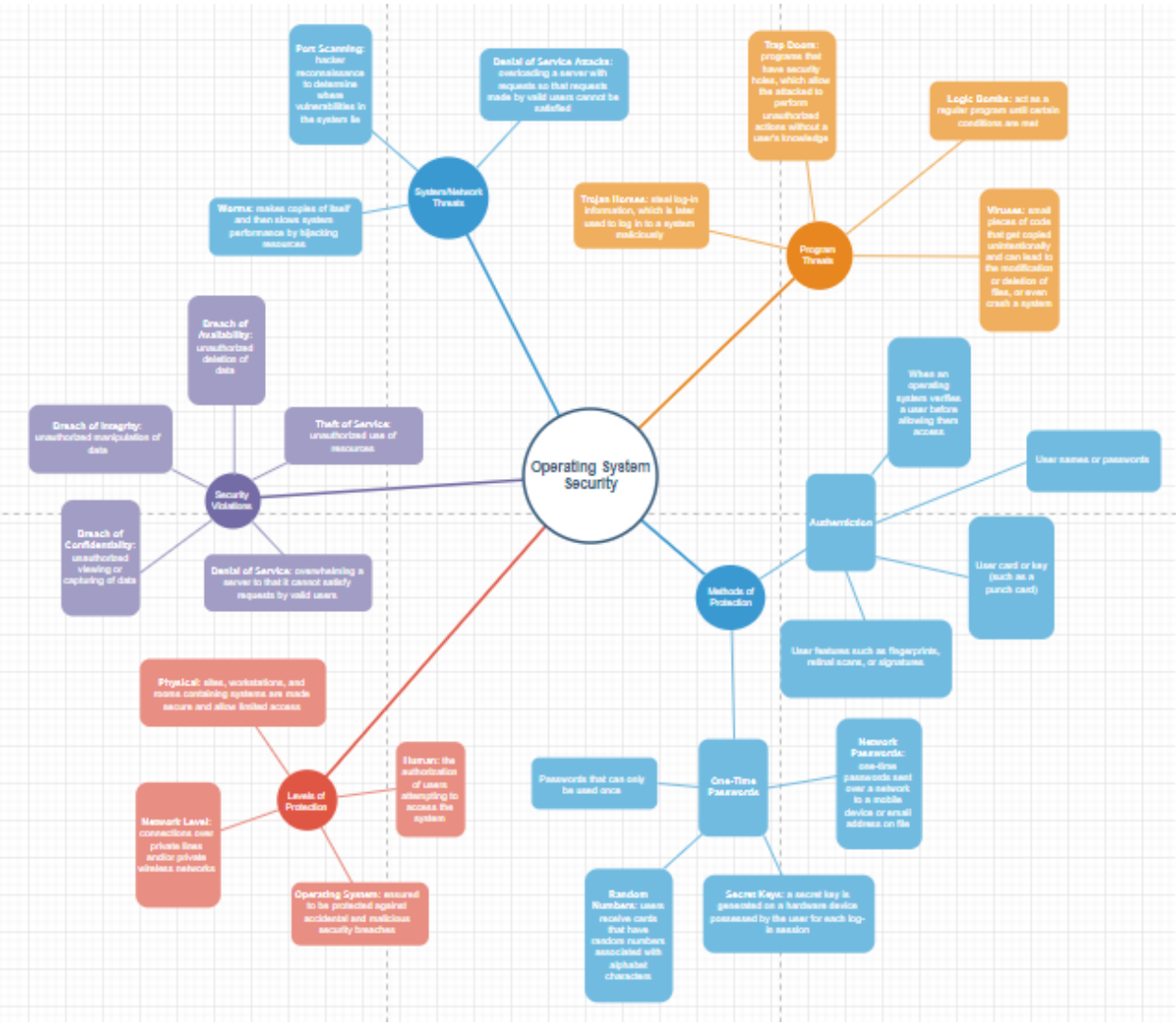
Link to Operating System Protection concept map: [Operating System Protection]

Operating System Protection Concept Map



Link to Operating System Security concept map: [Operating System Security]

Operating System Security Concept Map



Resources

Columbia University. (n.d.) Unit 4: Application and System Software. Retrieved October 4, 2018 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 September 20, 2018, from https://www.includehelp.com/operating-systems/process-control-block-in-operating-system.aspx

Geeks for Geeks. (n.d.). File Systems: Operating Systems. Retrieved October 4, 2018 from https://www.geeksforgeeks.org/file-system-operating-systems/

Hand, S. (2010). Operating Systems. Retrieved September 27, 2018 from https://www.cl.cam.ac.uk/teaching/1011/OpSystems/os1a-slides.pdf.

Shukla, A. (2017). Process Control Block in Operating System. Retrieved September 20, 2018, from https://www.includehelp.com/operating-systems/process-control-block-in-operating-system.aspx

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

Staff, C. (n.d.). Introduction to Operating Systems. Retrieved September 27, 2018 from http://staff.um.edu.mt/csta1/courses/lectures/csm202/os10.html

The Critical Section Problem. Retrieved September 20, 2018, from http://crystal.uta.edu/~ylei/cse6324/data/critical-section.pdf

TutorialsPoint. (n.d.). Operating System – Memory Management. Retrieved September 27, 2018 from https://www.tutorialspoint.com/operating_system/os_memory_management.htm

Tutorials Point. (2018). Operating System - Multi-Threading. Retrieved September 20, 2018, from https://www.tutorialspoint.com/operating_system/os_multi_threading.htm

Tutorials Point. (n.d.). Operating System- Security. Retrieved on October 11, 2018 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 on October 10, 2018 from http://www.cs.unc.edu/~dewan/242/s07/notes/prot.PDF