Contents

Section 1 – True/False (50 pts – 10 Questions)

OS and Paging

32. **True** / **False** A Process' Working Set Size comprises the number of unique Pages that were referenced within a past Working Set Window of accesses. This number can be used to assess the Process' Dynamic Locality within that specific window. **(L10S30+31)**

33. **True** / **False** In the Clock approximate implementation of the Least Recently Used paradigm for Page Eviction, we make clever use of Hardware capabilities that come for free. Specifically, we scan Page Table Entries with a sliding window (like 2 Clock hands moving in lockstep). When the first Clock hand passes over the Page, we reset its Accessed bit, and when the second Clock hand passes we check to see if its Accessed bit has been set again within the time it took for the sliding window to move over it. This allows us to understand that the Page was Recently Used (and should therefore not be Evicted), as the Hardware will have automatically set the Accessed bit of the Page Table Entry to 1 within that time, if the Page was referenced. **(L10S22)**

    The two hands move in lockstep, the first one resets the access bit from 1 to 0, and the second one checks to see whether the access bit has been set again to 1 in the time it took the second clock hand to pass over the page, which means it has been accessed recently.

Dynamic Memory

1. **True** / **False** Dynamic Memory Allocation can involve a Freelist of Blocks of Memory that can be used to fit allocation requests. These Blocks can be dynamically adjusted via Coalescing, to yield larger contiguous Virtual Address ranges and thus help fight External Fragmentation. **(L11S8)**

    Free lists and coalesceing, as well as fighting external fragmentation

2. **True** / **False** When Coalescing a Freelist of Memory Blocks we can either perform it at Block freeing time (Immediate Coalescing) or defer this to when we are actually searching to allocate new Dynamic Memory (Deferred Coalescing). **(L11S8)**

3. **True** / **False** Different Dynamic Memory Allocator algorithms have been proposed historically, but there exists an optimal Allocator that can guarantee no Fragmentation under any stream of allocation / deallocation requests. **(L11S9)**

    "There cannot be an 'always best' result"

4. **True** / **False** The Best-Fit Allocator algorithm suffers from the need to search the entire Freelist to find the closest fit to the requested amount of Dynamic Memory, but also from the "Sawdust" effect where the remainder Memory Blocks after Splitting end up being very small. **(L11S11)**

    The best fit allocator does search the freelist until it finds the best block to fit the requested amount of memory. It also suffers from the sawdust effect.

5. **True** / **False** The First-Fit Allocator algorithm suffers from the fact that it uses the first available Memory Block that can fit the requested amount of Dynamic Memory, and therefore tends to reuse the same regions of the Virtual Address Space (addresses that are closer to the start of the Freelist). **(L11S13)**

    The first fit allocator does indeed search a list to find the first block that fits, but the list is **sorted**. It can either be LIFO, FIFO, or by address. It depends on what the sorting method is going to be.

6. **True** / **False** The First-Fit Allocator algorithm has different variants, i.e. the LIFO, the Address-Sort, and the FIFO, wherein a freed Memory Block can be placed at a differently-sorted order within the Freelist. This yields different performance patterns under different allocation / deallocation streams. **(L11S14)**

    The first fit allocator does have different variants that obtain different results in terms of different allocation and deallocation request strips.

7. **True** / **False** The Buddy Allocator algorithm is a less generic strategy that is motivated by the fact that we can handle allocation requests as if they were of the order 2n. The derived benefits include the elimination of External Fragmentation, and the very efficient Coalescing of Memory Blocks. **(L11S23)**

    Generic Allocation strategies can prove to be overly generic Allocation requests are frequently in 2^n order (e.g. allocation of Physical Pages in Linux)

- Fast search (Allocate) and merge (Free)
- Avoid iterating through Freelist
- Avoid External Fragmentation for requests of 2^n
- Keep Physical Pages Contiguous
- Used by Linux, FreeBSD

8. **True / False** A program relying on a specific Dynamic Memory Allocator algorithm will exhibit one of the three distinct behaviors: a) Ramp, b) Peaks, or c) Plateau. The uniquely identifiable type of behavior that will arise during runtime depends primarily on the algorithm utilized for Dynamic Memory allocation / deallocation. **(L11S32)**

   It can exhibit a combination of any number of the three behaviors, not just one of the three at a time.

9. **True / False** Segregated Freelists is an approach to fight Fragmentation, by dedicating a different Freelist to Dynamically Allocating a specific size of object type (or a range of object sizes), e.g. one Freelist for sizes 1-2, one for size 3, one for size 4, one for sizes 5-8, one for sizes 9-above. This increases allocation / deallocation throughput, but suffers from wasting at least 1 Page for every single object size (/range) we need to allocate. **(L11S38)**

   With segregated freelists we fight fragmentation by allocating a different freelist of a different and/or different size range. This indeed helps us increase allocation/deallocation throughput, but as mentioned on the slide, we have to waste one page per object size.

10. **True / False** Slab Allocation is a technique that dedicates a Contiguous range of Physical Pages for the allocation of a specific object type. This is equivalent to having a preallocated Memory Pool of objects. It is rarely used in practice because it cannot be implemented alongside the most commonly used strategy of Buddy Allocation. **(L11S39)**

    Slab allocation is indeed a technique that dedicates contiguous ranges of physical pages for the allocation of a specific object type or class, and this indeed has the characteristics of a pre-allocated memory pool. It is a cache of objects, ready to be allocated for the operating system. However, the statement that it is rarely used in practice because it cannot be used alongside Buddy Allocation is false as it is in fact implemented on top of Buddy Allocation.

11. **True / False** Explicit Freelists have to store the size of each Memory Block to be able to access the next Block in them, while Implicit Freelists store pointers to the next (and/or previous depending on the implementation) Block. **(L11S41)**

    Explicit freelists are ones that store **pointers** to next and/or previous blocks, and implicit freelists are ones that store the **block size** in order to be able to offset and access the next block.

I/O and Disks

12. **True / False** Devices can be accessed through special Instructions (e.g. in/out on x86), or through Memory-Mapped I/O. The latter is less clunky in terms of kernel code (we perform traditional loads and stores to Memory Addresses) but the kernel is responsible to ensure these Address Space range mappings remain dedicated throughout the entire OS. **(L12S10+13)**

    Devices can be accessed with specialized instructions like the in-out instruction as well as through memory-mapped I/O. In addition, L12S13 mentions that memory-mapped I/O is less clunky than in/out, but the kernel is responsible to ensure that a specific mapping is maintained across the entire OS.

13. **True / False** Interrupt-based communication with Devices is always preferred over Polling, because it is generally true that the cost of Context Switching/and back into the communicating Process will remain less than spending CPU cycles to wait until Device data are ready (even in cases of Livelock). **(L12S16)**

    Interrupt-based communication is not generally always preferred over polling. When we have very fast communication, it is much more expensive to always have to trigger context switches.

14. **True / False** The Device Registers (i.e. Status, Command, Data Registers) are the external communication Interface the Device Hardware offers to conduct operations with an Operating System. **(L12S8)**

    Device registers are indeed the external interface that is offered by the device such that the operating system may communicate with it.

15. **True / False** For Hard Disk Drives, the overall Disk Latency caused by Seeking (moving the Disk arm to the correct Track) and Rotational Delay (waiting until the desired Sector is aligned) is lesser than the amount of time it takes for Disk Transfer (reading/writing of Data on the Sector). **(L12S31)**

    Transfer latency is much lower than the latency required to do mechanical motions with seek and rotation delays on a hard disk drive.

16. **True / False** Hard Disk Hardware is responsible for mapping of Physical Sectors to Linear Addresses, and handling the possible remapping due to Bad Sectors. The Hardware also supports a layer of Read-Ahead and Write-Back Cache. These are internally handled by the Disk and therefore the Operating System cannot directly account for them. **(L12S37)**

    The hard disk is responsible for the mapping of physical sectors to linear addresses, and for remapping any bad sectors. The hardware is also responsible for doing re-ahead and write-back caching. These are hardware functionalities, and they are the operating system's such that it cannot account for them directly.

17. **True / False** In Hard Disk Drive Scheduling, due to the large delays associated to mechanical motion of the Disk arm, the use of either First-Come-First-Serve (FCFS), Shortest Seek Time First (SSTF), or Circular-SCAN (C-SCAN) plays a major role. FCFS is simple to implement, and SSTF is an improvement that takes better advantage of Locality than C-SCAN, but it can lead to Starvation. **(L12S40+42+44)**

    FCFS (L12S40) is easy to implement, SSTF (L12S42) leverages locality, but it can lead to starvation, and C-SCAN (L12S44) scheduling also takes care of locality, not as effeciently however, as SSTF, but it is preferred such that due to the fact that it doesn't lead to starvation.

Filesystems

18. **True / False** In typical Filesystem abstractions, an inode is a structure that stores metadata and the location of data, (but itself is not a File – i.e. it can be a File type inode, a Directory type inode, etc). **(L13S38)**

    In a file system, an inode can be a filetype inode, a directory inode, etc... It also stores metadata about the file, directory, or general object that it corresponds to.

19. **True / False** A Hardlink is a Directory Entry that corresponds to a File. Effectively this means it is just another path to reach the File, therefore different Hardlinks can correspond to the same File (point to the same inode). The OS keeps a reference count to the number of Hardlinks (paths) to the file, and once all of them are removed this means that the File is deleted. **(L13S16)**

    A hardlink is indeed a directory entry, i.e. just another name that points to a particular file. The operating system also does keep a reference count to the number of hardlinks that point to the same file, i.e. it holds that information within the files' inode `links_count` reference counting variable, and when all of them are removed, then the file is considered to be removed as well.

20. **True / False** A Softlink is an inode with a special meaning. It stores the Path to a Filesystem target (File or Directory) by name (i.e. within the content of its data). The Operating System treats it specially by reading that Path and using it instead to perform File/Directory access by that name (i.e. it uses is as a "synonym"). In effect, this means that the stored path name might be invalid (not correspond to an actual Filesystem target). **(L13S17)**

    A softlink is a synonym for a filename or a path. Effectively, this is done by having a separate inode, i.e. a special file so to speak, that has a special flag to indicate that it is a symbolic link. This special link, this symbolic link, has its own data that stores the path to the actual link that it points to. Since this is a path, it might be an invalid path, so the underlying filesystem target might be invalid within the operating system at a point in time.

21. **True / False** Access Control Lists (ACLs) and Capabilities are two different ways to implement Filesystem Protection mechanisms in an Operating System. ACLs are object-centric and easier to manage, although in the case of a large number of Subjects we end up using Subject (/User) Groups to increase efficiency. Capabilities are like tickets granting access to an Object in the Operating System, but once a Subject obtains a Capability it has to use it very sparingly because of the associated large overhead. **(L13S21+22)**

    Access Control Lists and Capabilities are indeed different ways to manage protection. ACLs are easy to manage, and they can handle a large number of subjects by having subject or user groups. Capabilities are easy to transfer, and are indeed like tickets, as they grant permission to use objects. However, Capabilities facilitate faster access. Once the Capability is obtained, read and write accesses become much faster than the operation required to obtain the Capability in the first place. ACL checks are expensive (like an `open(...)` system call).

22. **True** / **False** File Descriptors are an Application of Capabilities. They are effectively indices to a Process' `filp` Table (maintained a the Kernel Space), which at that location contains a description of already acquired Permissions with respect to a particular Object. For this reason they can become dangerous in their utilization, as if the underlying Object changes (e.g. its ACL Protection properties change), the corresponding Process' Capability will still grant it the same level of authority when accessing that Object. **(L13S23)**

    File descriptors are indeed applications of capabilities. They are exactly as described, and exist for each process `filp` table that is stored in kernel space, the indexes held in that kernel space table which correspond to a Capability (what we call the file descriptor, and that's why it's an integer number). On the other hand, it is indeed dangerous to use Capabilities because once obtained, they are stored inside the process' `flip` table, we now have the key that unlocks a particular protection access to the object, and even if the object changes in terms of permissions, then we can still use that key to maintain the same type of access.

23. **True** / **False** The FAT Filesystem is implemented as a Linked-List of Filesystem Blocks, with the key optimization that the list of Block-to-Block Links is kept at a separate Region of the Disk (the "File Allocation Table" region). **(L13S33)**

    The FAT Filesystem indeed contains a linked list of file system blocks, but any pointer chasing has to be done at a separate region that keeps those links, named the File Allocation Table (FAT).

24. **True** / **False** Multilevel Indexed Files (effectively the implementation carried in Unix inodes) allow to access Data Blocks through a combination of Direct Accessing and Single, Double, and Triple-level Indirection. This facilitates fast accesses for small Files (or for the initial data within a File), but also unlock the ability to create very large Files (using the multiple levels of indirection). **(L13S37)**

    Multi-level indexed files, or the implementation that is carried in UNIX inodes involves the first direct access pointers (direct block pointers) facilitating immediate access to blocks, and then single indirect, double indirect, and triple indirect blocks allow the system to address larger spaces to store the files' data.

25. **True** / **False** Operating System-level Caching of Read and Write operations (Read-Ahead & Write-Back) aims to increase the efficiency of Disk accesses. There exist similar Caching functionalities implemented on the Disk Hardware however, and it is therefore required to enable only one (either Kernel or Hardware-supported Caching) due to possible conflicts between them. **(L13S45)**

    Operating system level caching of read and write operations coexists with hardware cache implementations.

Fast Filesystem

26. **True** / **False** The Unix Fast File System (FFS) leveraged Block Fragments to utilize a high Blocksize (but at low wastage), Bitmaps for the more compacted encoding and efficient search of the Data Block Freelist, and Cylinder Group Clustering (narrowing down the "distance" between Group Superblock, Bitmap, and Inode Table -to- the corresponding Data Blocks) in order to leverage spatial Locality of Hard Disk Sectors which is known to significantly affect Disk Latencies. **(L14S6+14+20)**

    The Fast File System leverages three things: block fragements (L14S6), a bitmap of free blocks (L14S14), and cylinder groups (L14S20) in order to improve upon the original UNIX filesystem.

Log-Structured Filesystem

27. **True** / **False** The Log-structured Fast File System (LFS) is an approach that tries to exploit Hard Disk Drive Locality by performing Filesystem updates as purely Sequential operations (by maintaining a rolling-record of Filesystem updates as if they are recording on a forward-moving tape drive). It requires a complicated implementation of the structure of the Data on the Disk, including maintaining an inode Map (map of inode number → Disk linear location) that is itself kept in Memory and stored on the Disk in pieces (not in the starting Region of the Disk, but spread along it), as well as Cleaning & Compaction of Segments because due to its fundamental operation, data are not overwritten (old data are left behind during recording the log of updates). **(L15S3+7+14+16)**

    The Log-Structured Filesystem essentially treats the disk like a tape drive and sequentially logs updates to the filesystem. In order to make it work efficiently, however, certain optimizations are required, like using an inode map that is itself spread around different pieces and locations along the disk, and also there needs to be cleaning and compaction done because of the sequential update nature of the writes, where it would leave previous information behind.

Linux Filesystems, FSCK, and Journaling

28. **True / False** The ext2 Linux Filesystem comprised multiple Block Groups, and within these were contained: a) a Superblock (1 Block-size), b) the Group Descriptors (multi-Block size), c) the Data Block Bitmap (1 Block-size), d) the inode Bitmap (1 Block-size), e) the inode Table (multi- Block size), and f) the Data Blocks of the Group. **(L16S4)**

Super Block	Group Descriptors	Data Block Bitmap	inode Bitmap	inode Table	Data Blocks
1 Block	$\geq$ 1 Block(s)	1 Block	1 Block	$\geq$ 1 Block(s)	n Blocks
		Free Blocks Bitmap	Free inodes Bitmap	Fixed-Size Table	

29. **True / False** The Superblock of each Block Group of the ext2 Linux Filesystem contains information describing the entire Filesystem. It is therefore redundant to maintain a copy of it in every single Block Group and therefore it is only in the first Group of the Filesystem that it contains valid values. **(L16S4)**

    The Superblock of each block group is indeed as described, but it is not just upkept in the first group of the file system but instead in all groups of the filesystem. This adds redundancy and guards against data corruption of the filesystem, which is similar to the redundancy approach of having a copy of the File Allocation Table in the FAT Filesystem.

30. **True / False** File System Consistency checKing (FSCK) addresses the problem of a crash happening before a sequence of Disc Sector writes associated to a single Filesystem update have been completed. For example, when creating a file we may end up with a non-updated inode Table, but an updated inode Bitmap, due to a crash in between atomic Disk writes. Although every possible Filesystem Inconsistency state can be detected and fixed, it is a costly operation because the entire Filesystem needs to be examined. **(L16S15)**

    FSCK operates as described, however, there is no way to fix certain inconsistent states due to different disk crash sequences that can occur.

31. **True / False** The ext3 Linux Filesystem's Journaling solution to Crash Consistency relies on first writing on the Disk the sequence of intended updates to the Filesystem (a Journal), in separate steps that guarantee that: i) either the Journal will contain the complete information related to the entire update, or ii) it will be detectable that the update never made it to the Journal completely. This allows to either "replay" Journal entries as Filesystem updates after a crash, or discard them completely, such that the Filesystem never ends up in an Inconsistent state. **(L16S22+27)**

    The Ext3 Linux Filesystem Journaling does operate in steps that allow us to write a journal of intended operations and/or transactions with the filesystems such that we end up in situations after a crash where we either have a complete record of the intended transactions (and therefore we can replay them or discard them completely).

32. **True / False** Ordered Mode Journaling is a key optimization that addresses the efficiency problem that comes from having to journal updates to the Data Blocks section of the Filesystem. With this approach, we first perform a direct update of the Data, and apply Journaling only for the Metadata. This is the default ext3 Journaling mode. **(L16S29)**

    Ordered Mode Journaling is an optimization that addresses the efficiency problem of journaling. In this mode, we write the file dated to the system first and then journal only the metadata. This is indeed the default mode fo the Ext3 solution.

Networking and Network File System

33. **True / False** The principle of Packet Encapsulation allows the various Layers associated to Networking to encode -or- extract the information related to their own level of functionality in/from the corresponding Protocol Data Unit section (e.g. Header), and keep the remaining message intact. This allows each message to self-contain all the information related to its transmission/reception. **(L17S10)**

    At each layer packets can embed the information of or extract information from packets from the subsequent layer, which allows them to self-contain information about the reception and transmission within them.

34. **True / False** Datagram and Stream Sockets are Operating System functionalities that allow the implementation of either a) Unreliable, atomic-message-level communication, or b) Bi-directional (potentially multi-Frame) message

pipes. The corresponding Transport-Layer protocols implemented for these two are named User Datagram Protocol (UDP) and Transmission Control Protocol (TCP). **(L17S15)**

There are two different types of sockets in operating systems: datagram sockets and stream sockets, and they each provide the functionalities listed. Both are associated to the UDP protocol and the TCP protocol, each.

35. **True** / **False** A Network Interface Card (NIC) Driver is OS software that is responsible to examine an incoming message and appropriately Route it, or handle it by figuring out its Packet Header-encoded Protocol and enqueue it for Protocol-Layer (UDP/TCP) Kernel processing. **(L17S28)**

    A network interface card driver is software that's responsible for appropriately handling routing, or handling received messages by examining packet headers and routing and forwarding them to the system's appropriate protocol queues.

36. **True** / **False** The Network File System (NFS) v2 enabled not just a network-based (even disk- less) Filesystem interface (like the Network Drive (ND) protocol), but also the ability for File Sharing, while complying with Unix Filesystem semantics. This effectively unlocked the ability to treat regular Filesystem-transaction System Calls as the unifying Interface to either perform: a) local Filesystem operations to a Disk, or b) sequences of Remote Procedure Calls (RPCs) to a NFS server. **(L17S31+32)**

    Network Filesystem V2 not only just enabled remote filesystem, but also the sharing of that filesystem through RPCs (Remote Procedure Calls) acting through the interface.

37. **True** / **False** The Network File System (NFS) v3 protocol was an extension that improved performance during WRITE operations, by not forcing a Client to have to flush their local Cache after each write, as writing from a different Client could occur between two WRITE Remote Procedure Calls (RPCs). This was done by adding several fields to the associated XDR structures, related to: a) Write Stability (requested and reported), b) a Server crash verification indicator, and c) NFS Server-reported modification / creation time attributes before and after the WRITE operation. **(L17S42-46)**

    NFSv2 requires clients to flush their cached version of a file between subsequent writes because there could exist a race condition with a different client accessing and writing to the same file. NFSv3's extensions include the ability to report metadata modification and creation time before and after write operations, allowing the client to detect whether there has been an in-between modification of that file.

Security

38. **True** / **False** The Bell-La Padulla model regards the OS implementation of a Mandatory Access Control (MAC) framework that addresses Secrecy, i.e. the ability to ensure that: a) a Subject cannot read information above their Confidentiality Level (no "Read-Up"), and b) a Subject cannot leak information to a Confidentiality Level below them (no "Write-Down").

39. **True** / **False** The Biba model regards the OS implementation of a Mandatory Access Control (MAC) framework that addresses not only Secrecy but also Integrity, i.e. the ability to ensure that: a) a Subject cannot corrupt data above their Integrity Level, and b) a Subject cannot become corrupted by another Subject at an Integrity Level below them.

40. **True** / **False** In the Low water-Mark Access Control (LOMAC) framework which is a non- military-grade implementation of an Integrity model, every distinct OS Process ("Job") is considered to be the Subject in terms of Security, regardless of whether it performs data sharing with another Process. This is especially important a Process' change of Integrity as it observes lower-Integrity data needs to immediately lower every other OS Process' Integrity it interacts with.

Virtual Machine Monitors

41. **True** / **False** Using a Virtual Machine Monitor (VMM) we can achieve Software Compatibility across different OS versions, better Resource Utilization across multiple Virtual Machines (VMs) by multiplexing Hardware, Checkpointing and Migration of an entire VM state across machines. **(L19S8)**

42. **True** / **False** The Trap-&-Emulate approach of Virtualization allows a VM to directly execute Instructions on the CPU, but any Privileged Instructions Trap back to the Virtual Machine Monitor (VMM) which can then safely Emulate their effects. **(L19S18)**

    The Trap-and-Emulate approach allows a virtual machine to directly execute instructions on the CPU, even if they are priviliged, but then it traps to the Virtual Machine Monitor in order to simulate privileged instructions.

43. **True / False** Shadow Page Tables directly map from Guest OS Virtual Memory Addresses to Host Machine (real Hardware) Addresses. These are the Page Tables actually used by the Hardware (the MMU), and it is a more efficient method than always having to perform a translation by the Virtual Machine Monitor (VMM) from Guest OS Physical Memory to Host Machine Addresses, but it also means that the Virtual Machine Monitor (VMM) is responsible to manage the Consistency of Guest OS Page Tables and the Shadow Page Tables. **(L19S26)**

    Shadow Page tables enable direct translation from guest virutal pages to host machine pages, therefore they are much more efficient and they are the ones that are actually being used by the hardware. However, this means that the virtual machine monitor has to keep consistency of the guest pages with the Shadow Page Tables that are actually being used by the system, by the hardware.

44. **True / False** The technique of Memory Tracing allows a Guest OS operating inside a Virtual Machine (VM) to be informed of the Memory access operations performed by the Virtual Machine Monitor (VMM) on the real (Hardware) Machine Memory. **(L19S32)**

    Memory tracing allows the virtual machine monitor to control or see the accesses that are being made in memory by the guest operating system that's operating in a virtual machine. The virtual machine is not the one being informed by memory accesses performed by the virtual machine monitor, when the opposite is true. The virtual machine must be unaware of the fact that it is operating as a virtual machine, therefore such information should not be leaked to the level of the guest operating system.

45. **True / False** Hardware extensions implemented by chip manufacturers allow for the more efficient implementation of Virtualization. These include even a distinct Execution Mode, "Guest Mode", which can be entered and/or exited while having the Hardware loading/saving the entire corresponding Guest OS Virtual Machine state from/to a special Virtual Machine Control Block (VCMB) data structure. **(L19S37+39)**

    The guest mode is a separate mode added to CPU hardware in order to support virtualization. In this separate mode, there is an associated virtual machine control block that represents machine state of the virtual machine. Separate instructions allow it to enter and exit guest mode, and this involves saving the virtual machine control block, which is a heavyweight data structure.

46. **True / False** Virtual Machine Monitors (VMMs) can force a Guest OS to "naturally" release some of their allocated Pages (their least valuable ones) by injecting multiple Page Allocation requests (to which the Guest OS will respond by Evicting Pages using its native Page Eviction policies). This is done via a pseudo-Device called the "Balloon Driver" operating within the Guest OS, which also communicates with the VMM. **(L19S41)**

    Balloon Driver is a trick that allows the virtual machine monitor to force the guest operating system to evict some of its pages, thus releasing memory for the virtual machine monitor to otherwise exploit. This, being done at the guest operating system level, means that the natural way that the guest operating system exploits page eviction policy will be used.

Section 2 – General Questions (30 pts – 3 Questions)

Dynamic Memory

1. Describe the difference in how malloc() will perform Heap allocation if the requested size is above MMAP_THRESHOLD (128 KB by default), and if it isn't. **(L11S45+43+44)**

   The threshold parameter specifies how `malloc` will perform allocation. For values below the `MMAP_THRESHOLD`, allocation is performed using the set break `sbrk` behavior to allocate memory on the heap by incrementing the program break. For values greater than `MMAP_THRESHOLD`, allocation will revert to using the Virtual Memory Mapping `mmap` in order to place the allocated regions in virtual address space regions.

2. Describe what is happening when the following Dynamic Memory Allocation patterns are observed by a Process: a) Ramp, b) Peaks, c) Plateau. What would be the OS side-effect of using a simple (non-hybrid) Allocator and having it handle an interleaved Ramp & Peaks pattern? **(L11S33+34+36)**

   When a process exhibits the ramp behavior, it means that objects are getting allocated, but the corresponding memory is not being freed. With peaks, the behavior is that many objects are being allocated in a short amount of time, but they are all instantaneously released at the same time. In the case of plateaus, a number of objects is getting allocated by the process that live for a very long time. The side effect of using a simple allocation strategy and having interleaved ramp and peak patterns is the fact that we will be allocating some long lived objects with the ramp pattern and then allocate in an interleaved way many short lived objects. Once the short lived objects disappear, they are going to leave empty slots within our memory but at the same time this means that we will have already reserved possibly new pages due to the increased demand that happened during the allocation of the many objects, therefore our long lived objects might have to be placed at different pages. This means that we are spreading around data that might be more efficient to have on the same page. The only way to achieve that would be to have a hybrid strategy that differently handles the allocation of those long lived objects and the quick rise and fall due to the immediate release of the peaks being placed at different pages of the virtual address space.

I/O and Disks

3. Describe the difference between a Polling strategy and an Interrupt-based strategy for Device communication, and mention the benefits of each (i.e. mention a use-case tha each is better suited to address) **(L12S14-16)**

   Polling is a strategy where a task or a process waits until the transaction is completed by the device, therefore spending a certain number of CPU cycles waiting until the transaction has been completed whereas interrupts context switch to a different task until there has been an interrupt raised by the device to indicate that the transaction has been completed. Therefore, not wasting the CPU cycles it would have waited while the transaction completion was being performed. Polling is better for devices with fast communication while interrupts are better as if context switching was used, it would lead to a receive lovelock while slower devices better utilize interrupts.

4. Describe why Sequential access in a Hard Disk usually leads to better performance when compared to Random access. Mention one Filesystem implementation tha leverages the benefits of Sequential access (and describe how in a few words). **(L12S35)**

   For a hard disk, seek and rotation are mechanical motions and therefore the latencies associated with those are large, while transfers are fast. This means that sequential access of data is probably going to be associated to sequential access of sectors that might be on the same track or adjacent tracks, and therefore the entire latency will be transfer dominated. This is why sequential access is faster. Examples of file systems that leverage sequential access include the Log-Structured Filesystem within which we perform sequential write of records that correspond to updates of the filesystem as if we are a rolling tape record, and we might also consider an example of sequential access optimization: the fact that the Fast File System is separated into cylinder groups such that shorter distance jumps are required in the inode table in the beginning of the group to the data that corresponds to the inodes of that particular group.

5. Describe the benefit(s) of RAID 3 over RAID 2. Describe the benefit(s) of RAID 5 over RAID 4. Describe the benefit(s) of RAID 6 over RAID 5. **(L12S59-65)**

   RAID 2 computes parity information that adds reliability, but requires multiple disks to store it. RAID 3 uses a single parity disk because it leverages the XOR based parity, having higher storage utilization.

   RAID 4 has a single parity disk while RAID 5 distributes the parity sectors across multiple disks and thus doesn't have to suffer the bottleneck of having a single disk for every kind of parity-related access.

   RAID 6 uses two parity sectors for every parity group, adding the ability to tolerate two disk failures.

Filesystems

6. Describe the differences between a Hardlink and a Symlink, and specifically mention: a) what they conceptually represent, b) how they are implemented in an OS, c) their connection to a File's actual state of existence in the OS. **(L13S16+17)**

   A hardlink is a synonym name for a file. In an operating system, this is implemented as a directory entry so when a new hardlink to a file is created, the system is creating yet another directory entry that points to it. The implementation also involves updating the reference counting variable `links_count` of the inode of that particular file to show that yet another link or path exists to this file on the filesystem, so the inode's file needs to exist.

   A softlink is a synonym for a filename or a filepath. On an operating system, this corresponds to yet another inode, a special inode, that can be thought of as a special file with a special flag to indicate that it is a softlink file. In terms of implementation, this means that this special symlink file will have the path name that the softlink is pointing to within its data. Since this path is just a name, the path does not necessarily need to exist on the filesystem.

7. Describe the differences between an Access Control List and a Capability. Give an example where one can violate the other and describe how that would happen. **(L13S20+21+23)**

   Access Control Lists and Capabilities are implementations of protection. If we create a table where the rows are subjects and the columns are objects, then the entries of a column will correspond to the Access Control List, and the rows of the table will correspond to separate Capabilities. Capabilities are effectively like tokens or tickets that grant whatever subject possesses them the ability to access, as a specific level of protection, an entity or object of a computer system. On the other hand, access control lists are just that: lists. They are object-centric and therefore they are easier to manage. In the case where there are multiple subjects, they can be optimized by separating and using user groups instead.

   An example of when one can violate the other is the fact that a Capability is a file descriptor and a file descriptor is obtained by making the open system call. This is the point where the Access Control List for the file that is being opened is being accessed by the operating system in order to check the level of capability that will be granted. Once this is granted, however, the processes `filp` table, which is stored in the kernel space, will be updated with the specifications of that capability. Therefore, once obtained, the process will have the same level of access to that object for the duration of its lifetime, and this will happen even if in between accesses the Access Control List is updated to indicate that more strict access will be required for that object from then onward.

Linux Filesystems, FSCK, and Journaling

8. Describe the ext2 Filesystem structure (include a small diagram to help you). Mention each component's size in Blocks and briefly describe their role (what Filesystem related information they contain). **(L16S4)**

Super Block	Group Descriptors	Data Block Bitmap	inode Bitmap	inode Table	Data Blocks
1 Block	$\geq$ 1 Block(s)	1 Block	1 Block	$\geq$ 1 Block(s)	n Blocks
		Free Blocks Bitmap	Free inodes Bitmap	Fixed-Size Table	

   The file system contains different block groups. The Superblock contains a description of the basic size and shape of the filesystem. (As a sidenote, the Superblock is copied over and updated in every single block group for redundancy).

   Then come the Group Descriptors, which can be multiple blocks depending on how many are required to cover all the block groups of the filesystem, and the Block Group Descriptor Table contains the description for every one of the block groups of the file system. This description includes the address of its block bitmap, the address of its inode bitmap, the address of its inode table, the number of free blocks, the number of free inodes, and the number of directories for that particular block group. The block group descriptor table is also duplicated in all of the groups to guard against corruption, therefore every single block group contains the information of every other block group as well in the group descriptor table.

   Then comes the data block bitmap that contains a bitmap of the free and occupied datablocks of that block group.

   Then comes the inode bitmap that contains a bitmap of the free and occupied inodes for that block group.

   Then comes the inode table. The inode table can be more than one block, and it contains the table of all the inode data structures that describe this block group's files and directories.

   Finally, there's a region that stores the actual data, which can be more than one block in size.

9. Describe how Journaling works (include a small diagram to help you). Describe why Steps 1 and 2 are have to be separated (i.e. the OS has to wait until the Disk reports the write of Step 1 has been completed before requesting the write of Step 2) instead of merging them together. **(L16S24+27)**

   Journaling works by first writing the intended transactions with the file system to a separate region of persistent memory, and that is called the Journal. This happens in separate steps, i.e. we first write the dirty blocks to the journal as a single transaction, then we write the commit block as a single transaction, and then we actually copy the dirty blocks to the real filesystem. This is called checkpointing. Finally, we can reclaim the journal space for that entire transaction.

   The reason why we cannot merge steps one and two is the fact that if we try to write out the entire journal transaction as a single transaction, then it might end up being that chunks of it will have to be placed on separate blocks: different sectors of the disk and there is a layer of hardware handled scheduling that might take place in between, therefore in a crash, we may end up with a partially completed transaction on the disk just because some of the blocks corresponded to sectors that were scanned first as part of the C-SCAN scheduling strategy.

Networking and Network File System

10. Describe the problem with the Network File System (NFS) v2 protocol's WRITE operation when used by different Clients sharing the same File. Mention the improvement added by NFS v3 to address this. **(L17S43+45)**

    With the Network File System v2 Protocol, during write operations, the client has to flush the cached state of a file because it would not be possible for it to know whether another client has issued and completed a write transaction with the NFS server between subsequent write transactions on its own side.

    The NFSv3 Protocol improved upon v2 by having additional fields incorporated into the XDR structures. It incorporated metadata related to the file modification time and creation time before and after the write transaction, allowed the client to detect whether its own cached version of the file did not correspond to the actual version of the file at the time it received its transaction because of the fact that another client had issued a transaction and completed it in between.

11. Describe possible ways that even Mandatory Access Control frameworks that ensure Secrecy with Security Labels can be compromised.

Virtual Machine Monitors

12. Describe the key improvement offered by Shadow Page Tables, and describe the process of how a Virtual Machine Monitor sets these up. **(L19S26+29)**

    Shadow Page Tables postpone to the following key improvement: instead of having two layers of redirection from guest virtual pages to guest physical pages and then translating those every time in the virtual machine monitor from guest physical pages to host machine pages, instead it uses Shadow Page Tables that are the ones that are actually being used by the hardware that contain translations from guest virtual pages to host machine pages directly. These are maintained by the virtual machine monitor and their consistency has to be guaranteed as well. A virtual machine monitor sets this up through the following process: they are progressively built up by tracking page faults generated by the guest. For example, when a guest writes a new virtual page to physical page number mapping in its own guest page tables, the hardware will actually end up using the Shadow Page Tables, but these do not contain such a mapping. This will create a page fault, which will then be forwarded to the virtual machine monitor, which can now compare and notice that no virtual machine mapping exists, set it up, and then insert it into its own Shadow Page Table, and then revert back to the guest process.

Section 3 – Practical Questions (30 pts – 3 Questions)

OS and Paging

(L10S16)

8. Given a Least Recently Used (LRU) Page Replacement Policy and 4 Physical Pages available on the system, calculate the number of Page Faults that will occur for the following Page Referencing string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Phys Page	1	2	3	4	1	2	5	1	2	3	4	5
0	1	1	1	1	x		1	X		1	1	5
1		2	2	2		x	2		X	2	2	2
2			3	3			5			5	4	4
3				4			4			3	3	3

Dynamic Memory

(L11S27)

1. The following sequence of Dynamic Allocations:
   (a) A: Alloc 4 Bytes
   (b) B: Alloc 6 Bytes
   (c) C: Alloc 2 Byte

   is handled by a Buddy Allocator. You are given the current state of the Address Space after the allocations are completed (each row represents a subsequent time point in the timeline of the Buddy Allocator Memory operations).

<table>
<tr><td>2</td><td>2</td><td>2</td><td>2</td><td>2</td><td>2</td><td>2</td><td>2</td></tr>
<tr><td colspan="8">2^4 (16) Bytes Total</td></tr>
<tr><td colspan="8"></td></tr>
<tr><td colspan="4"></td><td colspan="4"></td></tr>
<tr><td colspan="2">A: $4/2^2$</td><td colspan="2"></td><td colspan="4"></td></tr>
<tr><td colspan="2">A: $4/2^2$</td><td colspan="2"></td><td colspan="4">B: $6/2^3$</td></tr>
<tr><td colspan="2">A: $4/2^2$</td><td></td><td></td><td colspan="4">B: $6/2^3$</td></tr>
<tr><td colspan="2">A: $4/2^2$</td><td>C: $2/2^1$</td><td></td><td colspan="4">B: $6/2^3$</td></tr>
</table>

   Give the Timeline after the following sequence of Dynamic Deallocations:
   (a) Dealloc C
   (b) Dealloc B
   (c) Dealloc A

2	2	2	2	2	2	2	2
A: $4/2^2$		C: $2/2^1$		B: $6/2^3$			
A		**Dealloc**		B			
A		**Merge**		B			
A				**Dealloc**			
Dealloc							
Merge							
Merge							

2. The following Allocator code implements an Implicit Freelist-based Dynamic Allocation strategy that purely depends on the OS Memory Mapping System Call utility. What is the problem with it?

   (L11S41)

```
00    void* malloc(size_t n) {
01        if (n == 0) return NULL;
02        size_t* p = mmap(NULL, n + sizeof(size_t),
                           PROT_READ | PROT_WRITE,
                           MAP_PRIVATE | MAP_ANONYMOUS
                           -1, 0);
03        if (p == (void *) -1) {
04            return NULL;
05        } else {
06           return p;     // Issue because of the Implicit Freelist structure
07       }
08    }
```

0	
1	0 byte allocations don't have to do anything
2	Uses mmap to allocate enough space to fit n bytes + an extra size_t to fit the block's header. Remember: the Implicit Freelist implementation where each block has a header that contains its size such that it knows the offset to the next block
3	Check to see if the pointer is invalid. If it is, it was not allocated properly.
4	
5	
6	p is the address to the block, not the payload of the block. p must be offset by size_t before it is returned to return the payload instead of the pointer to the metadata.

I/O and Disks

3. A File is being requested to be read-in by the OS, whose Data are stored within the following sequence of Data Blocks.

   (L12S42+44)

Block	0	1	2	3	4	5	6	7	8
Disk Track	98	173	35	120	19	122	60	63	52

The Disk Head starts at a Track location of 50. What is the sequence that the File Data Blocks are going to be read-in if:

(a) The SSTF Scheduler is used

(b) The C-SCAN Scheduler is used

(assuming that none of the above Data Blocks correspond to Sectors across different Tracks).

SSTF

Cur	Next	Block
50	52	8
52	60	6
60	63	7
63	35	2
35	19	4
19	98	0
98	120	3
120	122	5
122	173	1

C-SCAN

Cur	Next	Block
50	52	8
52	60	6
60	63	7
63	98	0
98	120	3
120	122	5
122	173	1
173	19	4
19	35	2

Filesystems

4. Assume an inode Data Structure containing an array of 15 Block pointers, where:
(L13S37)

(a) 12 are Direct Block

(b) 1 is Single-Indirect Block

(c) 1 is Double-Indirect Block

(d) 1 is Triple-Indirect Block

Assuming the Block Size for the Filesystem is 4KB, and a Block Pointer size is 32-bit (4 Bytes), what is the maximal File size supported by the Filesystem?

The 12 Direct Blocks can hold $12 * 4KB = 48KB$

1 Single Indirect Block contains $\frac{4KB}{4\ bytes} = 1024$ Direct Block locations. Each block address points to a block of 4KB, so the total amount of data it will be able to hold will be $1024 * 4KB = 4,096KB \approx 4MB$

Each address takes up 4 bytes of space, and in a system that has 32-bit pointers, those pointers will take up 4 bytes each. The reason that 1024 block locations are accessible is because the indirect block will store as many pointers to other blocks of memory in the actual block of data, which is $4KB$ (thus the indirection), and so $4KB$ can store 1024 block addresses.

1 Double Indirect Block contains $\frac{4KB}{4} = 1024$ Single Indirect Block locations, so $1024*(1024*4KB) = 4,194,304KB \approx 4GB$

Notice how a double indirection can hold up to $1024 * 1024$ addresses, each of which will point to a $4KB$ block.

1 Triple Indirect Block contains $\frac{4KB}{4} = 1024$ Double Indirect Block locations, so $1024 * (1024 * (1024 * 4KB)) = 4,294,967,296KB \approx 4TB$.

Maximum overall filesize supported on system: $4TB + 4GB + 4MB + 48KB$

Byte Conversion Table		
1 KB	1000 Bytes	$1KB$
1 MB	1000^2 Bytes	$1000KB$
1 GB	1000^3 Bytes	1000^2KB
1 TB	1000^4 Bytes	1000^3KB

Linux Filesystems, FSCK, and Journaling

5. A crash happened before the OS could complete all Filesystem transactions and FSCK is now going to run. The current state of the Filesystem is as follows:

   inode Bitmap contains (1 – means the inode is used, 0 – means the inode is free):

#0	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	#12	#13
none	bad blks	root	rsvd	rsvd	rsvd	rsvd	rsvd	rsvd	rsvd	rsvd			
1	1	1	1	1	1	1	1	1	1	1	1	1	1

   What fixes will be made after FSCK is done?

   inode Table contains entries:

```
...
inode #2
links_count: 1
Data:
- '.' , inode # 2, type: Directory
- '..' , inode # 2, type: Directory
- 'fileA' , inode # 11 , type: File     // inode 11 has two paths to it
- 'fileB' , inode # 11 , type: File
- 'dirA' , inode # 12 , type: Directory
...
inode #11
links_count: 1                          // While links_count is 1 when it should be 2
Data:
- "abcde"
inode #12
links_count: 1
Data:
- '.' , inode # 12, type: Directory
- '..' , inode # 2, type: Directory
inode #13
links_count: 1                           // This file is unreferenced
Data:
- "fghijk"
```

   The first issue, the fact that inode 2 has two paths to it while having a `links_count = 1`, is resolved by updating `links_count` to 2.

   The second issue is that the file in inode 13 is unreferenced by the filesystem, so FSCK will put it in the Lost and Found so that the administrator can re-parent it.


TWICE 1기
해냈어요!