# OS Definition, ISA

1. What is an Operating System? List its primary responsibilities.

Answer: An operating system is a privileged software for managing a computer system’s resources including memory, I/O, and CPU. An OS also provides services to user-level processes via system calls, for example, reading/writing to files and network, inter-process communication, memory management, protection, etc.

1. What are the three (or four) different ways in which OS code can be invoked? Explain.

Answer:

Hardware Interrupts: event notifications from hardware devices to OS,

System calls: Service requests from user processes to OS

Exceptions (also called traps)﻿﻿﻿: Events to OS indicating an incorrect execution by use processes

Kernel threads: Long-running threads in the kernel context.

[Note for TA: Mentioning hardware interrupts and exceptions is necessary. Mentioning one of system calls or Software Interrupts is necessary. Mentioning kernel threads is optional.]

1. Explain the following interfaces in a computer system
   1. Instruction Set Architecture (ISA)
   2. User Instruction Set Architecture (User ISA),
   3. System ISA,
   4. Application Binary Interface (ABI).
   5. Application Programmers’ Interface (API)
2. Why doesn’t a program (executable binary) that is compiled on the linux machine execute on a Windows machine, even if the underlying CPU hardware is the same (say x86)?
3. What is meant by virtualization? Give examples of many(virtual)-to-one(physical, one-to-many, and many-to-many resource virtualization.
4. Describe the process lifecycle illustrating the states and transitions.
5. **[10 pts]** Which state transitions occur in a process lifecycle when a process
   1. Makes a blocking read() system call
   2. Exceeds its CPU timeslice
   3. Is interrupted by a hardware interrupt
   4. Dereferences a NULL pointer.
   5. Attempts to acquire a blocking lock that is taken by another process?
   6. Is pre-empted
   7. Voluntarily yields the CPU

Answer:

1. Transition from Running to Blocked state,
2. Transition from Running to Ready state
3. No transitions. Process remains in running state and interrupt handler runs in the context of the current process.
4. CPU raises an exception. OS terminates process. Process exits running state.
5. Transition from running to blocked state.
6. Transition from Running to Ready state
7. Transition from Running to either Ready or Blocked state, former in case of “yield” operation, and latter in the case of I/O requests or other blocking system calls.
8. Which state transitions (if any) occur in a process lifecycle when a process
9. Runs too long on the CPU?
10. Tries to read keyboard input, but no input is available?
11. Receives a signal?
12. Attempts to execute a System ISA instruction in user space?
13. Attempts to perform down() operation on a semaphore whose value is zero?

Ans:

* 1. Running to ready
  2. Running to blocked
  3. If the process is already in running state, then remains in running state and signal handler is invoked (if set).

If currently ready, then the process remains in ready state till CPU scheduler schedules the process. OS marks the signal as pending in a per-process bitmask.

If currently blocked on an interruptible wait, then process moves from blocked to ready. OS marks the signal as pending in a per-process bitmask.

* 1. An exception is generated and the process goes from running to “terminated”, which is not explicitly shown in the state diagram.
  2. Process goes from running to blocked state.

1. During a process lifecycle, what events can cause the following transitions?

(a) Ready to Running state

(b) Running to Ready state

(c) Ready to Blocked state

(d) Blocked to Ready state

﻿Answer:

(a) Ready to Running state: When a CPU scheduler selects a process from the ready queue and schedules it on the CPU.﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿

(b) Running to Ready state: When the CPU scheduler pre-empts a process, or when a process voluntarily yields the CPU even though it has more﻿﻿﻿﻿﻿ work to do.

(c) Ready to Blocked state: When a process has to wait for an event that might take a while to be received.

(d) Blocked to Ready state: When a process that is waiting for an event is woken up after the event is received by the OS

1. Why are frequent context switches expensive in terms of system performance?
2. What is cold-start penalty? What are some ways to reduce it?
3. What are some key factors that affect application performance after a context switch?

# Memory management

1. Show the typical memory hierarchy of a computer system. Explain the tradeoffs between latency (access times), capacity, and persistence, across different levels of memory hierarchy.
2. What is a page table?
3. How many page tables are maintained by the operating system?

Answer: One per process

1. If there were no TLB, how would memory accesses be affected?
2. What are the following? What do they do? Where are they located?
   1. Memory Management Unit (MMU)
   2. Translation Lookaside Buffer (TLB)
   3. Page tables
   4. Swap device

Answer:

* 1. MMU is the part of execution hardware that translates virtual addresses to physical addresses. It is located alongside the CPU.
  2. TLB is an associative cache, located in MMU, that caches frequently accesses page-table entries, so that MMU doesn’t need to perform a page-table walk in main memory.
  3. Page table is a data structure that hold the translations from virtual page number to physical page number for each process. It is located in the main memory. There is one page table per process.
  4. Swap device refers to the space on hard disk (or any other secondary persistent storage) where pages from the main memory can be temporarily stored till they are needed again by the any process or the OS.

1. Where are MMU, Page Table and TLB located?

Answer:

* 1. MMU is located in the execution hardware next to CPU.
  2. Pa﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿ge tables are located in the main memory.
  3. TLB is located in the execution hardware in MMU.

1. What is a page fault and a TLB miss? Which system component resolves each of them?

Answer:

* 1. TLB Miss: occurs when MMU does not find a valid entry in TLB to resolve a virtual page number to physical page number. MMU then walks the page table in main memory to resolve the TLB miss.
  2. Page Fault: occurs when MMU does not find a valid page table entry in the page table to resolve a virtual page number to physical page number. MMU then generates a trap (Page Fault) to the OS which then handles the page fault.

1. Mark the statements that are true
   1. A page table is an array of physical memory pages
   2. Page table entries dictates whether a process can read, write, or execute the contents of a physical page.
   3. A page table maps physical page numbers to virtual page numbers
   4. Page table entries can be used to track which memory pages are infrequently accessed.
2. How is a virtual address converted to a physical address in a virtual memory system? Explain the roles of MMU, TLB, and Page Tables.

Answer:

* 1. Virtual address is given as input to the MMU.
  2. MMU breaks up the address into virtual page number and byte offset into the page.
  3. MMU then uses the virtual page number to query the TLB for virtual-to-physical page number translation. If the translation is found in the TLB (TLB hit) then MMU uses the physical page number thus found.
  4. Otherwise (TLB miss), the MMU traverses the page table located in the main memory to locate the page table entry (PTE) corresponding to the virtual page number and retrieves the physical page number from the PTE. The PTE is inserted into the TLB for future accesses.
  5. MMU then combines the physical page number (retrieved from either TLB hit or Page table walk) with the byte offset extracted in the first step to construct the physical address.
  6. MMU uses the resulting physical address to access the physical memory.

1. If the kernel has highest privileges, why would a kernel fault result in a crash?  
     
   User-space memory errors are handled by page fault (PF) handler in the kernel, which usually kills the offending process. If kernel code makes a mistake, the kernel must handle the error in its own PF handler. If the error is critical, like accessing an unallocated memory address, then kernel kills itself.  Some other situations, the handler may spit out an error message (kernel OOPs) and continue executing, but the kernel state may be corrupted, causing strange things to happen.

Not every virtual address has an associated physical memory allocated. Allocation must be requested explicitly, such as when a process starts up, calls malloc() etc. For kernel code, some memory is allocated at boot time, and others dynamically, through routines like kmalloc() etc. If a kernel bug accesses an unallocated virtual address, there's nothing the kernel can do to continue intact, whereas an offending user process can be killed.

1. If you increase or decrease the page size in a system, how (and why) will it affect ***(a)*** the size of the page tables, and ***(b)*** the TLB miss ratio?
2. What’s TLB Coverage? Why is TLB coverage important?
3. How can one increase the TLB coverage?
4. In memory management, what is meant by relocation and protection? Why are they needed?

Answer: Relocation refers to shifting all addresses of code in main memory by a BASE bytes, so that the code can be loaded into, and executed from, any location in main memory.

Protection means checking each memory reference by the code to verify that it is below a maximum value and that it doesn’t try to access memory belonging to others.

Relocation and protection together allow a programmer to write code using relative addressing (0-MAX) without having to know the final memory address where the code and data may actually reside.

1. How are relocation and protection implemented in Pentium architecture? Consider the roles of both segmentation and paging.
2. How is segmentation different from paging? Why was each technique invented?
   1. Segmentation allows each process to have multiple address spaces (segments). Paging allows each address space to be partitioned into equal sized pages.
   2. Segmentation was invented to allow programmers to separate logically distinct parts of their program into different segments. Paging allows parts of an address space to be independently moved in and out of main memory without having to remove/retrieve the entire address space at once.
3. Using either Multics or Pentium architecture as an example, explain how segmentation is used in enforcing protection?
4. How is a virtual address converted to a physical address, considering both segmentation and paging in (a) Multics and (b) Pentium architectures?

Answer:

* + **Multics**: Virtual address consists of Segment number, page number and byte offset into a page. Segment number is used to index into the segment descriptor table to find the segment descriptor. Segment descriptor contains the address of the page table for that segment. Page number is used to index into the page table to find the page table entry. Page table entry gives the physical page number. Byte offset is used to index into the page specified by the physical page number to access the specific byte of data.
  + **Pentium**: Virtual address consists of Segment number (contained in a register called Pentium selector) and the relative address being accessed in the segment. Segment number is used to index into the segment descriptor table to find the segment descriptor. Segment descriptor contains the Base address of the segment in the linear address space and the size of the segment (limit). The relative address is added to the base address of the segment to obtain the linear address. The linear address is broken down into page number and byte offset into the page. The page number is used to index into a page table (common for all segments) to find the page table entry. Page table entry gives the physical page number. Byte offset is used to index into the page specified by the physical page number to access the specific byte of data.

1. Consider a machine with B-bit architecture (i.e. virtual address and physical address are B bits long). Size of a page is P bytes.
   1. What is the size (in bytes) of the virtual address space of a process?
   2. How many bits in an address represent the byte offset into a page?
   3. How many bits in an address are needed to determine the page number ?
   4. How many page-table entries does a process’ page-table contain?

Answer:

1. 2^B bytes
2. log2(P) bits
3. B - log2(P) bits
4. 2^B/P pages OR 2^(B-log2(P)) pages – same answers.
5. A machine has a 32-bit address space and an 8-KB page. The page table is entirely in hardware, with one 32-bit word per entry. When a process starts, the page table is copied to the hardware from memory, at the rate of one word every 100 nsec. If each process runs for 100 msec (including the time to load the page table), what fraction of the CPU time is devoted to loading the page tables? (Assume that each process uses its entire virtual address space during execution.)
6. A machine has a 32-bit address space and an 4KB page. The page table is entirely in hardware. Each page-table entry is 4 bytes in size. When a process starts, the page table is copied to the hardware from memory, at the rate of one byte every 25 nano-second. If each process has a CPU burst of 200 msec (including the time to load the page table), what fraction of the CPU time is devoted to loading the page tables? (Assume that each process uses its entire virtual address space during execution.)
   1. Ans: Address space = 2^32 bytes.
   2. Number of PTEs = 2^32/4KB = 2^20 PTEs.
   3. Size of page table = 2^20\*4 = 2^22 bytes
   4. Time to copy PT from memory to MMU = 2^22 \* 25nsec = 100\*2^20 nsec
   5. Fraction of CPU time to load PT = 100\*2^20 \* 10^(-9) / 200 \* 10^(-3) = 2^19 \* 10^(-6) = roughly 0.5 or 50%
7. Consider a machine having a 32-bit virtual address and 8KB page size.
   1. What is the size (in bytes) of the virtual address space of a process?
   2. How many bits in the 32-bit virtual address represent the byte offset into a page?
   3. How many bits in the 32-bit address are needed to determine the page number ?
   4. How many page-table entries does a process’ page-table contain?
8. Consider a machine having a 64-bit virtual address and 32KB page size.
   1. What is the size (in bytes) of the virtual address space of a process?
      1. 2^64 bytes
   2. How many bits in the virtual address represent the byte offset into a page?
      1. log2(32K) = log2(2^5 \* 2^10) = 15
   3. How many bits in the virtual address are needed to determine the page number ?
      1. 64-15 = 49
   4. How many page-table entries does a process’ page-table contain?
      1. Max # of PTEs = Max # of pages = 2^49
9. **[10 pts]** Consider a machine having a 64-bit virtual address and 16KB page size.
   1. What is the size (in bytes) of the virtual address space of a process?
   2. How many bits in the virtual address represent the byte offset into a page?
   3. How many bits in the virtual address are needed to determine the page number ?
   4. How many page-table entries does a process’ page-table contain?

Answer:

* 1. 264 bytes
  2. log2(16K) = log2(24 \* 210) = 14 bits
  3. 64 - 14 = 50 bits
  4. Max # of PTEs = Max # of pages = 250

1. For each of the following decimal virtual addresses, compute the virtual page number and offset for a 4-KB page, an 8 KB page, and a 16KB page: 20000, 32768, 60000.
2. A computer with a 32-bit address uses a two-level page table. Virtual addresses are split into a 9-bit top-level page table field, an 10-bit second-level page table field, and an offset. How large are the pages and how many pages are there in the address space?
3. Which system components handle TLB misses and page-faults, and how, in a machine with
   1. architected page-table?
   2. architected TLB?

Answer:

* 1. Architected Page table: MMU handles the TLB miss by walking the page table in main memory. OS handles the page fault by allocating of paging-in the requested page.
  2. Architected TLB: OS handles the TLB miss by resolving the virtual page number to physical page number. If the page is not in main memory (page fault), then the OS brings the page into memory or allocates a new page.

1. What is the purpose of “Referenced” and “Modified” (“Dirty”) bits in the page table entry? How are they manipulated by the (a) hardware, and (b) operating system?
2. Describe Optimal Page Replacement (OPR) algorithm. Why is it called "Optimal"? Why is it not practical to implement OPR?
3. Explain the Least Recently Used (LRU) page replacement algorithm.
   1. LRU picks that page as victim (to be evicted) which has not been used for the longest time in the past.
4. Why is LRU a good approximation of Optimal Page Replacement (OPR)?
   1. OPR picks a victim page as one that won’t be used for the longest time in the future. LRU is a good approximation of OPR because, for most typical programs, if a page has not been used for a long time then there is a good chance that it won’t be used again for a long time again in the future.
5. OPR (Optimal Page Replacement) and LRU (Least Recently Used)
   1. Why is it impossible to implement OPR?
   2. Why is it hard to implement LRU?

Answer:

* 1. OPR is impossible to implement because it requires precise knowledge of future page accesses by the process, which is impossible to predict accurately.
  2. LRU is hard to implement because it requires maintaining a sorted list of pages in the order of their memory access recency. This list must be updated upon every memory access, which is prohibitively expensive.

1. What is the difference between internal and external fragmentation?
2. What is meant by “Internal” fragmentation?

Answer: Internal fragmentation occurs when part of an allocated memory region is not used by a process and is thus wasted.

1. What is “External” fragmentation of memory? How can it be resolved?

Answer: External fragmentation occurs when all free memory regions are small and distributed at different non-contiguous locations in the main memory. As a result, an allocation request for a large contiguous memory region cannot be satisfied, even though enough free memory exists. External fragmentation is resolved by **compaction**, i.e. moving all allocated memory regions to one end of physical memory so that all free memory is contiguous at the other end.

1. What is a “working set”? Why is it so important?
2. Explain how the following page replacement algorithms work: (a) Clock, (b) WSClock (c) Second Chance.
3. How does the Second Chance page replacement algorithm improve upon the FIFO page replacement algorithm?
   1. FIFO evicts a page that was brought into main memory the earliest. Second Chance is just like FIFO except that it examines the reference (R) bit of the oldest page. If R bit is set then the page is given a second chance, meaning that it is moved to the back of the FIFO list and the R bit for that page is reset. Then Second chance examines the next oldest page, and so on till it finds the page with R=0.
4. How does Clock page replacement algorithm improve upon the Second Chance page replacement algorithm?
5. Briefly explain how the following page replacement algorithms work:
   1. FIFO (First In First Out)
   2. Second Chance
   3. Clock.

Answer: See slides!

1. Suppose that “page table pages are paged”, meaning that (some or all of) the memory allocated to hold page tables can be paged-in and out of the main memory by the operating system. Suppose further that you have two-level page-tables, i.e. a first-level page-directory which tracks the second-level page table blocks.
   1. Which parts of the page-table can be paged (moved in and out of main memory)?
   2. Where are the memory address translations (i.e. page table entries) for the “paged page-table”?
   3. Can the memory used for your answer in (b) be paged? Why? Or Why not?
2. In paged-segmentation implementations: Multics has one page table per segment, whereas Pentium has one page table for multiple segments. Why the difference? Which one is better? Why?
   1. OR (a) How many page tables are there per segment in Multics and Pentium. (b) Which one (Multics/Pentium) is better? Why?

Answer: Multics has one page table per segment. Pentium has one page table for all segments in a process. Pentium’s design is better for segmentation because switching between uses of different segments in a process doesn’t require the MMU to switch states between two different page tables. This eliminates TLB flushes and cold-start penalty.

1. What problem does segmentation solve that paging doesn’t solve? What problem does paging solve that segmentation doesn’t solve?
2. Consider two processes that set up one page of shared memory for inter-process communication with each other. Given what you know about virtual memory management, explain how the OS would set up this shared memory page at the level of page tables?
3. Suppose that the Operating System wanted to track (or intercept) every write performed to a specific memory page by a user-level process. Explain how the OS would achieve this goal?

Answer: The OS would mark the desired pages read-only. When the process attempted to write to the page, an exception would be generated by the hardware giving control to the OS. The OS can then do whatever it wants with the write attempt such as record it and allow the write or deny the write, etc.

1. How does TLB Coverage and TLB miss ratio vary with the size of a page?
2. Consider a virtual memory system running on an architected page-table hardware supporting two-level page tables. Page tables are not locked in memory and may be swapped to disk. An lw (load word) instruction reads one data word from memory; the address is the sum of the value in a register and an immediate constant stored in the instruction itself. Neither machine instructions nor page-table entries nor data words can cross a page boundary. In the worst case, how many page faults could be generated as a result of the fetch, decode, and execution of an lw instruction? Explain why?
   1. In the worst case,
      * both levels of page tables may be swapped out
      * the page containing the lw instruction may be swapped out
      * the page containing the word being accessed by lw may also be swapped out.
      * Total 5 page faults:
        1. One page fault to fetch the page directory (1st level of the page table) from disk
        2. One page fault to fetch from disk the page table holding the address translation for the page containing the lw instruction
        3. One page fault to fetch the page containing the lw instruction from disk
        4. One more page fault to fetch from disk the 2nd-level page table block containing the translation for the memory word being accessed by the lw instruction. (Note that the first level page directory was already fetched in Step 1, and we assume that its not swapped out again before step 4!)
        5. One final page fault to fetch the page containing the memory word being accessed by the lw instruction from disk
3. Which of the following memory designs can cause internal fragmentation only, external fragmentation only, both, or neither? Briefly explain why.
   1. Pure paging
   2. Pure segmentation
   3. Paging with Segmentation
   4. Using superpages of the same size (no base pages or any other superpage size)
   5. Using a mix of superpages of different sizes

Answer:

* 1. Pure paging: only internal
  2. Pure segmentation: both
  3. Paging with segmentation: Only internal
  4. Superpages of the same size: Only internal
  5. Mix of superpages: Internal plus external fragmentation (at super page size granularities).

1. What is hysteresis? Explain how the page-out/page-in mechanism in the OS uses hysteresis and why?

OR

* 1. How does the swap daemon (paging mechanism) avoid rapid oscillations in paging activity when memory pressure increases?

Answer:

Hysteresis loosely means designing a system so that it doesn’t oscillate rapidly around a single threshold. For paging, the OS maintains two memory usage thresholds — low and high. When the memory usage reaches the high memory threshold then the paging/swap daemon starts evicting victim pages to the disk. Once enough pages are evicted, so that the memory usage falls below the low memory threshold, then the paging daemon stop evicting pages. This way, the OS avoids the situation where the memory usage oscillates rapidly around a single memory threshold.

1. Under what condition does thrashing occur in memory management? How can the OS resolve thrashing?

Answer:

Thrashing occurs when the sum of the working set of all processes in the system exceeds the size of the main memory. In such a situation, the paging daemon constantly evicts pages which are immediately accessed by a process (because the page is in working set), so the paging daemon has to bring the page back to main memory right away. Thus the system is spending most of its time moving pages between the main memory and the disk and not doing much useful work. The solution to thrashing is to reduce the degree of multi-programming (i.e. the number of active processes), either by killing some of the processes or my swapping out all of their memory content to the disk, till the sum of the working set sizes of the surviving processes is less than the physical memory size.

1. Considering memory protection, explain how the operating system ensures that user-level processes don’t access kernel-level memory?

Answer:

Consider systems in which the OS maps itself into the address space of each process. OS resides in a more privileged segment with privilege bits in the segment register set to 0 (bits 00). The user code and data resides in a less privileged segment in the higher address space with the privilege bits in its segment register set to 3 (bits 11). User code executes with privilege level 3 (in CPU’s EFLAGS register). So if the user process tries to access kernel code and data (which has protection bits 00), then the MMU raised an exception indicating a segmentation violation, the OS gets control, and possibly kills the process.

Alternatively, in some systems, the OS resides in a separate address space by itself and is not mapped to the individual process address spaces. IN such cases, a process cannot refer to OS memory addresses by design.

1. How can the operating system track
   1. Dirty (or updated) memory pages for the purpose of eviction?
   2. Every memory write performed by a process to specific memory pages?

Answer:

1. Dirty bits in the PTE are set to 1 by the MMU upon each write to a page. OS periodically scans the dirty bits in the page table entry and, if under memory pressure, evicts the pages to disk.

OS sets the page access permissions to read-only. When a process tries to write to the page, a write exception is generated by MMU. OS takes control and performs the write (if allowed) on behalf of the process. OS leaves the page permissions to read-only to trap future writes.

# File Systems

1. What is a File system

Answer: OS component that organizes user’s data and meta-data. Three functions: Manages data. Manages meta-data. Manages free space on disk.

1. What’s an i-node? Where is it stored?

Answer: i-node is a data structure maintained by the file system to store the meta-data for each file/directory. There is one i-node per file/directory in the file system. I-nodes are stored on the disk, usually in the same partition where the contents of the file reside.

1. What’s the simplest data structure for an i-node? Then why is UNIX i-node so complicated?

Answer: The simplest data structure for an i-node is an array, having one entry for every logical block in the file. Inode is more complicated to

* + 1. Allow fast access to small files (as with the array), yet allow large files to be accommodated dynamically as the file size increases, as opposed to allocating a large fixed size array at file creation time.
    2. Allow different parts of inode to be stored in different memory locations

1. In a file-system, (a) What is meta-data? (b) Where is meta-data stored? (c) Why is it important for a file system to maintain the meta-data information? (d) List some of the typical information that is part of the meta-data.

Answer:

(a) Meta-data: is the information that describes the properties of the actual data or contents of a file/directory.

(b) Meta-data is stored in the i-node.

(c) Because without meta-data, the corresponding data of a file may not be accessible or understandable.

(d) Meta-data includes information such as owner, time of last change, time of last access, protection attributes, location of data blocks on the storage device etc.

1. If you collect a trace of I/O operations below the file system cache (at device driver or physical disk level), what type of I/O operations do you expect to see more of -- write I/O requests or read I/O requests? Explain why.

Answer: We expect to see more write I/O operations because the file system cache will service most read I/O requests.

1. (a) Suppose you collect a trace of I/O operations above the file system layer (in applications or in system calls). Do you expect to see more write I/O operations or read I/O operations? (b) Now suppose you collect a similar trace of I/O operations below the block device layer (in the disk or device driver). Do you expect to see more write I/O operations or read I/O operations? Explain why?

Answer: (a) Depends on the application, (b) We expect to see more write I/O operations because the file system cache will service most read I/O requests.

1. If you increase or decrease the disk block size in a file system, how (and why) will it affect ***(a)*** the size of the inode, and ***(b)*** the maximum size of a file accessible only through direct block addresses?
2. How does the inode structure in UNIX-based file-systems (such as Unix V7) support fast access to small files and at the same time support large file sizes.
3. What does the file system cache do and how does it work? Explain with focus on the data structures used by the file system cache.

Answer: File system cache stores frequently accessed data blocks from the file system in the main memory. When file system processes a read I/O request, it first looks for the data block in the file system cache. If the block is not found in the cache (miss) then the file system issues an I/O request to the disk. It has two data structures. First is a hash table that is used to perform fast lookup of data blocks during reads. Second is a linked list sorted in LRU order. When a paging daemon needs to evict a victim page, it evicts the least recently used page from the list.

1. Explain therole of *file system cache* during (a) read I/O operations and (b) write I/O operations.

Answer:

Read operations: When file system processes a read I/O request, it first looks for the data block in the file system cache. If the block is not found in the cache (miss) then the file system issues an I/O request to the disk.

Write Operations: The file system temporarily buffers the data in the file-system cache so that the write system call can return back to user-level process immediately. Periodically, the FS cache commits all dirty pages to the disk.

1. Describe two different data structures using which file system can track free space on the storage device. Explain relative advantages/disadvantages of each.

Answer: (1) A linked list of free blocks on the disk. This is a simple data structure. But it makes it hard to perform contiguous allocations of disk blocks and to check whether a specific block on the disk is free or allocated. The size of this data structure decreases as the number of free blocks decrease. (2) A bitmap where each bit represents whether a specific block on the disk is free or occupied. Bitmap has a constant size irrespective of the disk usage. One can also perform compicated allocation operations such as the contiguious allocation mentioned above. it also makes it easier to check if a specific block on the disk is free or allocated.

1. How does a log-structured file system work? Why is its performance (typically) better than conventional file systems?

Answer: A log-structured file system is based on the observation that most I/O operations that are sent to the physical disk are writes, because the file-system cache will filter out most read operations. The entire disk is treated as a log. Whenever a write I/O occurs, the file system finds the closest free block from the current position of the disk head (the end og the log) and commits the write at that location. This avoids the overhead of seeking to a specific location on the disk to perform a write. Reads become a little more expensive because the file system now needs to locate the latest version of a requested data block. This is a typical application of the principle of making the common-case (here, writes) fast.

1. In a file-system, explain how two different directories can contain a common (shared) file. In other words, how do hard links work?

Answer: Both directories may refer to the files by different names. However, they store the same i-node number associated with the contents of the file on the disk. Hence any I/O operation on either of the two filenames will be directed to the same shared file on the disk. The i-node also contains a counter which increments every time a new directory links to a file and decrements when an unlink is performed. When the counter goes to zero, the file is deleted from the disk.

1. How does the inode structure in UNIX-based file-systems (such as Unix V7) support ***fast access to small files*** and at the same time ***support large file sizes***.
2. Explain the structure of a UNIX i-node. Why is it better than having just a single array that maps logical block addresses in a file to physical block addresses on disk?
3. Explain the steps involved in converting a path-name */usr/bin/ls* to its i-node number for the file *ls*.
4. What’s wrong with storing file metadata as content within each directory “file”? In other words, why do we need a separate i-node to store metadata for each file?
5. Assume that the

* Size of each disk block is B.
* Address of each disk block is A bytes long.
* The top level of a UNIX i-node contains D direct block addresses, one single-indirect block address, one double-indirect block address, and one triple-indirect block address.
  1. What is the size of the ***largest “small”*** file that can be addressed through direct block addresses?
  2. What is the size of the ***largest*** file that can be supported by a UNIX inode?

Explain your answers.

Answer:

1. Number of direct block addresses = D

Size of file addressable through only direct block addresses = D x B bytes

1. Number of block addresses that can be stored in a disk block = size of disk block/size of each address = B/A

Size of the largest file supported =

Size addressable through direct block addresses +

Size addressable through single indirect block addresses +

Size addressable through double indirect block addresses +

Size addressable through triple indirect block addresses

=

DB + (B/A)B + (B/A)(B/A)B + (B/A)(B/A)(B/A)B bytes

1. In a UNIX-like i-node, suppose you need to store a file of size 32 Terabytes (32 \* 240 bytes). Approximately how large is the i-node (in bytes)? Assume 8096 bytes (8KB) block size, 8 bytes for each block pointer (entry in the inode)., and that i-node can have more than three levels of indirection. For simplicity, you can ignore any space occupied by file attributes (owner, permissions etc) and also focus on the dominant contributors to the i-node size.

Answer:

Number of pointer entries per block = 8KB/8bytes = 1K = 2^10

Direct Block entries allow access to (roughly) : 2^10\*8KB = 8MB of file data (not enough)

Single indirect block allows access to another 2^10\*8KB = 8MB of file data (not enough)

Double indirect block allows access to another 2^10\*2^10\*8KB = 8 GB of file data (not enough)

Triple indirect block allows access to another 2^10\*2^10\*2^10\*8KB = 8TB of file data (again not enough but getting close...we need about 24TB more)

So we need one more level of Quadruple indirect block in which three of its triple indirect blocks are populated to give an additional 3\*(2^10\*2^10^2^10)\*8KB = 24TB

So one triple and three entries of one quadruple indirect blocks would together cover a 32TB file.

Total inode size is (3\*2^30 + 2^30 + 2^20 + 2^10 + 2^10) \* 8 bytes which is roughly 4\*(2^30)\*8 bytes (considering only triple and quadruple indirects) which is about 32GB.

1. In a UNIX-based filesystems, approximately how big (in bytes) will be an inode for a 200 Terabyte (200 \* 240 bytes) file? Assume 4096 bytes block size and 8 bytes for each entry in the inode that references one data block. For simplicity, you can ignore intermediate levels of indirections in the inode data structure and any space occupied by other file attributes (permissions etc).
2. In a UNIX-based filesystems, approximately how big (in bytes) will be ***an inode*** for a ***400 Terabyte (400 \* 2***40 ***bytes) file***? Assume 4096 bytes (4KB) block size and 8 bytes for each entry in the inode that references one data block. For simplicity, you can ignore intermediate levels of indirections in the inode data structure and any space occupied by other file attributes (owner, permissions etc).
3. Assume that the size of each disk block is 4KB. Address of each block is 4 bytes long. What is the size of the ***largest*** file that can be supported by a UNIX inode? What is the size of the ***largest “small”*** file that can be addressed through direct block addresses? Explain how you derived your answer.
4. Assume all disk blocks are of size 8KB. Top level of a UNIX inode is also stored in a disk block of size 8KB. All file attributes, except data block locations, take up 256 bytes of the top-level of inode. Each direct block address takes up 8 bytes of space and gives the address of a disk block of size 8KB. Last three entries of the first level of the inode point to single, double, and triple indirect blocks respectively. Calculate **(a)** the largest size of a file that can be accessed through the direct block entries of the inode. **(b)** The largest size of a file that can be accessed using the entire inode.

**Answer:**

Size of first level of the inode = 8KB

Size of attributes = 256 bytes

Space taken up by last three entries of the first level = 8 bytes \* 3 = 24 bytes

Space remaining to for direct block entries = (8KB - 256 bytes - 24 bytes)

Largest file that can be accessed through direct block entries = (8KB - 280 bytes)\*8KB/8bytes

Largest size of a file that can be accessed using the entire inode =

Size accessible from direct blocks +

Size accessible from single indirect blocks +

Size accessible from double indirect blocks +

Size accessible from triple indirect blocks

=

8192\*(8192 - 280)/8 + 8192\*8192/8 + 8192\*(8192/8)2 + 8192\*(8192/8)3 bytes

1. In the “UNIX/Ritchie” paper, consider three major system components: files, I/O devices, and memory. UNIX treats I/O devices as special files in its file system. What other mappings are possible among the above three components? (In other words, which component can be treated as another component)? What would be the use for each possible new mapping?
2. Suppose your filesystem needs to store lots of uncompressed files that are very large (multiple terabytes) in size. (a) Describe any alternative design to the traditional UNIX inode structure to reduce the size of inodes wherever possible (NOT reduce the file content, but reduce inode size)? (Hint: maybe you can exploit the nature of data stored in the file, but there may be other ways too). (b) What could be the advantage of your approach compared to just compressing the contents of each file?
3. Why doesn’t the UNIX file-system allow hard links (a) to directories, and (b) across mounted file systems?
4. Why did the authors of the “UNIX” paper consider the UNIX file-system to be their most important innovation?

Answer: Because it provides a unified way of handling conventional files, special files (device files), and inter-process communication (via file descriptors, shared memory descriptors, pipe descriptors, socket descriptors, etc).

**Stupid file system paper:**

1. Consider a RAID 0 array (striped and without redundancy). Compare how the I/O throughput (I/O operations per second) would vary under the Ext3 file system and under a hypothetical randomized-mapping (the so-called stupid) file system as you increase (a) the stripe unit of the RAID 0 array (for a fixed number of disks) and (b) the number of disks in the RAID 0 array (for a fixed stripe size). Justify your answer. (No need to give concrete numbers, just describe the trends accurately).
2. Consider a UNIX i-node for a file of size F bytes. What is the size of the i-node in bytes?

Assume that disk block size is B bytes, each block address size is A bytes. The top level of the i-node contains D direct block addresses,  one single-indirect block address, one double-indirect block address, and one triple-indirect block address.

Answer: Depends on how big is F.

Number of disk block addresses stored in each inode block = B/A

Minimum number of block addresses for a file of size F = F/B

Direct Block entries allow access to DxB bytes of file data.

So if F < DxB then only one disk block is required for inode, hence inode-size = B bytes.

Single indirect block allows access to an additional (B/A)xB bytes of file data

So if F < (DxB) + (B/A)xB then only two disk block are required for inode – one for attributes and direct block addresses and another for a single-indirect block. Hence inode size = 2B bytes.

Double indirect block allows access to another (B/A)x(B/A)xB bytes of file data. However, depending upon F, not all single-indirect blocks may be allocated.

Number of single-indirect blocks pointed to by the double-indirect block for a file of size F

= number of block addresses except in the direct block addresses/ number of addresses per block

= ((F/B) - D) / (B/A) -1

(Minus one because a single-indirect block is accessible from the initial i-node block.)

Plus we need space for one double-indirect block.

To total space = B + B + B + (((F/B) - D) / (B/A) -1)xB

Triple indirect block allows access to another (B/A)x(B/A)x(B/A)xB bytes of file data.

Triple indirect block allows access to another 2^10\*2^10\*2^10\*8KB = 8TB of file data (again not enough but getting close...we need about 24TB more)

So we need one more level of Quadruple indirect block in which three of its triple indirect blocks are populated to give an additional 3\*(2^10\*2^10^2^10)\*8KB = 24TB

So one triple and three entries of one quadruple indirect blocks would together cover a 32TB file.

Total inode size is (3\*2^30 + 2^30 + 2^20 + 2^10 + 2^10) \* 8 bytes which is roughly 4\*(2^30)\*8 bytes (considering only triple and quadruple indirects) which is about 32GB.

# RAID

1. Distinction between logical and physical I/O address spaces.
2. What was the original & current motivation for RAID?
3. Why is a multiple-disk system less reliable than a single disk?
4. How does Mean Time to Failure (MTTF) change as number of components in a system increases?
5. What are the different levels of RAID and how do each of them work?
6. What are the relative benefits/drawbacks of each RAID level?
7. How is data distributed in each RAID level?
8. How is parity calculated and stored in each RAID level?
9. What is the extent of read and write parallelism in each level?
10. How is the parity calculation bottleneck in RAID 4 solved?
11. In RAID-5, explain how can you perform a single logical write operation in no more than one physical read and two physical writes?

Answer:

Parity blocks should be staggered across disks and new parity should be computed without reading the rest of the data blocks.

Pnew = Pold XOR Bold XOR Bnew,

where Bold is the old value of the data block being written and Bnew is the new ball of the data block being written

1. Consider RAID levels 0, 1, 3, 4, and 5: Which RAID level provides the best (a) reliability (b) I/O Parallelism. Explain why.

Answer:

* 1. RAID 1 provides the best reliability, although at the expense of substantial overhead in extra disk space. It guarantees recovery from a single disk failure. It can also recover from most two-disk failures, except the ones in which both a primary disk and its mirror disk fail.
  2. RAID 0 provides the best I/O parallelism. It allows N simultaneous I/O operations on N disks. The I/O operations could be any combination of reads and writes.

1. In order to save power, disks are usually spun down (placed in sleep or low-power mode). This works well if there is only one disk in the system, if all data resides on the single disk, and if performance is not a major concern. Consider a RAID-5 system consisting of N+1 disks. Explain how you can redesign RAID-5 so that all the following requirements are satisfied: (1) fault-tolerance of original RAID-5 is maintained under all conditions, (2) energy consumption is minimized by spinning down one or more disks whenever possible, and (3) performance (read/write throughput) of the system is maximized to the extent possible. Again, while there is no single correct answer, you must explain all salient aspects of your design, justify any assumptions you make, and examine any design tradeoffs (e.g. energy savings to performance).
2. In RAID 5, describe how you can complete a write I/O operation using just 2 disk reads and 2 disk writes.
3. (a) Explain (with formula), how does parity computation differ between RAID 3 and RAID 4? (b) How does parity placement on the disk (not parity computation) differ between RAID 4 and RAID 5? Explain with example.

Answer:

(a) RAID 3: p[k] = b[k,1] XOR b[k,2] XOR ... XOR b[k,N]

where

p[k] is the parity of the kth stripe

b[k,i] = the ith fragment of data block b[k]

RAID 4: p[k] = b[Nk] XOR b[Nk+1] XOR ... XOR b[Nk+N-1]

where

p[k] is the parity of the kth stripe

b[Nk+i] = the data block on ith disk and kth stripe

(b) In RAID 4, all the parity information is stored in the (N+1)th drive.

In RAID 5, the parity information is distributed among the N+1 drives in a staggered manner.

1. How should parity be computed in RAID 5 to increase parallelism of write operations? Explain with parity computation formula.
2. What is the write parallelism problem in RAID and how is it solved?
3. Describe the design of a parity-based RAID system that can survive two-disk failures (as opposed to single-disk failure discussed in class). In your design, be sure to explain the following: (a) How your system would compute the parity required for recovery from a two-disk failure? (b) How your system would recover from two-disk and single-disk failures, (c) How much additional space would parity information occupy, compared to data, and (d) How many parallel read and write I/O operations can your system support?

Answer:

A straightforward solution is to extend RAID 1 with two additional parity disks. (This is equivalent to extending RAID 4 by mirroring every disk, including the parity disk).

**Alternatively**, extend RAID 5 by mirroring every disk.

1. Compute XOR-based parity over the primary data disks. And make a mirror of the parity disk.

**Alternatively**, the parity blocks can be spread out over the primary and mirror disks as in RAID 5.

1. Two-disk failures: If two unrelated disks (not primary and its mirror) fails, then simply copy over the mirror. If both the primary and its mirror fail, then reconstruct the failed primary by XORing the other primary disks. Then copy over the reconstructed primary disk to its failed mirror disk. Single-disk failure: Simply copy over the corresponding mirror disk.
2. Extra space : N/2 data disks would require N/2 mirror disks + 2 parity disks.
3. With first solution (extending RAID 1 or RAID 4), maximum read parallelism = N and maximum write parallelism = 1.

With second solution (extending RAID 5), max read parallelism = N+2 and max write parallelism = (N+2)/2.

1. For a RAID system **with N disks, including data and parity,** compare the level of parallelism provided by RAID 1, RAID 3, RAID 4, and RAID 5 for multiple simultaneous (i) read I/O operations, (ii) write I/O operations, and (iii) combination of read and write I/O operations? Explain your answers.

Answer:

(i) Read I/O operations:

RAID 1: Allows up to N reads to be processed at a time

RAID 3: Allows only one read I/O operation to be processed at a time

RAID 4: Allows up to N-1 parallel reads to be processed at a time

RAID 5: Allows up to N parallel reads to be processed at a time

(ii) Write I/O operations:

RAID 1: Allows up to N/2 writes to be processed at a time.

RAID 3: Allows only one write I/O operation to be processed at a time

RAID 4: Allows only one write I/O operation to be processed at a time because a single parity disk becomes a bottleneck.

RAID 5: Allows up to N/2 parallel write I/O operations to be processed at a time

(iii) Combination of read and write I/O operations:

RAID 1: Allows X parallel writes Y parallel reads where (2X+Y = N).

RAID 3: Allows only one I/O operation (read or write) to be processed at a time

RAID 4: Allows EITHER only one write OR N-1 reads to be processed.

RAID 5: Allows X parallel writes Y parallel reads where (2X+Y = N).

A correct answer could also be as follows without the formula: RAID 1 and 5 allows multiple reads and writes simultaneously.

1. Consider RAID levels 1, 3, 4, and 5 (forget about RAID 0 and 2). Which RAID level provides the best (a) reliability (b) I/O Parallelism. Explain why.

Answer:

1. RAID 1 provides the best reliability, although at the expense of substantial overhead in extra disk space. It guarantees recovery from a single disk failure. It can also recover from most two-disk failures, except the ones in which both a primary disk and its mirror disk fail.
2. RAID 5 provides the best I/O parallelism. In the best case, it allows N+1 simultaneous read operations and (N+1)/2 simultaneous write operations.

# Virtualization

**Undergraduate**

1. For system virtual machines, explain how virtual memory addresses are translated to physical addresses when (a) hardware supports EPT/NPT (extended/nested page tables) and (b) hardware only supports traditional (non-nested) page tables.

Answer:

1. Virtual address of a process is mapped to Guest Physical Address using standard page tables by the guest OS. Guest Physical address is mapped to Physical Address using EPT/NPT by the hypervisor. The actual memory memory translation from VA->GPA—>PA is performed by the MMU in hardware.
2. When hardware does not support EPT/NPT then the hypervisor constructs a Shadow Page Table by compressing the two-level of mappings VA—>GPA—>PA. Thus Shadow page table contains the mapping from VA—>PA. The shadow page table is used by the MMU hardware for memory translation as the process executes.
3. How does Intel VTx extending the traditional CPU execution privilege levels to support system virtual machines?
4. Compare different approaches for virtualizing I/O devices for virtual machines.
5. Explain the key hardware-level virtualization support provided by Intel fo
   1. Memory translation for VMs
   2. CPU privilege levels for guest OS execution
   3. Direct I/O device access by VMs

Answer:

(a) Memory translation for VMs

Intel VTx provides Extended Page Tables (EPT)﻿ for efficient memory translation for VMs. Virtual address (VA) of a process is mapped to Guest Physical Address (GPA) using traditional page tables by the guest OS.  GPA is mapped to (machine) Physical Address (PA) using EPT by the hypervisor. The actual memory memory translation from VA->GPA—>PA is performed by the MMU in hardware. During execution, the MMU walks both guest page tables and EPT to translate guest VA to PA.

(b) CPU privilege levels for guest OS execution

In addition to the four traditional execution privilege levels (0,1,2,3), Intel VTx provides two orthogonal privilege levels called the **root mode** and the **non-root mode.** The hypervisor and its processes execute in the root mode. The guest and its processes execute in the non-root mode.

(c) Direct I/O device access by VMs

I**ntel VTd extensions** allow a hypervisor to assign physical devices exclusively to a VM. A hardware feature, called **IOMMU**, allows the hypervisor to configure which physical memory locations a direct-assigned device can access via DMA operations, to prevent malicious VMs or devices from accessing other VMs.

1. Explain briefly with examples (1) Process virtual machine, (2) System virtual machine, (3) Emulator, (4) Binary optimizer, (5) High-level Language Virtual Machine.

Answer:

1. Process VM: virtualizes the execution environment of a single process. E.g. JVM, Digital FX!32.
2. System VM: virtualizes the execution environment of the entire software stack (OS + applications). E.g. VMWare, Xen, KVM, etc
3. Emulator: Translates from one ISA to another. Could be either a process or a system VM. For example, Digital FX!32, VirtualPC, Code Morphing in Transmeta Crusoe.
4. Binary Optimizer: Source and target ISA are the same, but optimizes some functionality of the program, such as speed, or energy usage, or security. For example BSD Jails, or any sandbox, or memory leak detector. (I don’t think I discussed concrete examples in class, so its OK if students don’t give examples here.)
5. HLL VM: Process VM which translated from a virtual ISA (for which there are no physical machines) to to a native ISA. E.g. JVM, .NET CLI.
6. Which interface does a Process VM virtualize? Which interface does a System VM virtualize?

Answer: Process VMs virtualize the ABI, which consists of the system calls and user ISA. System VMs virtualize the ISA, which consists of both user ISA and system ISA.

1. (a) How do Interpreters differ from Dynamic Binary Translators? (b) How do Binary Optimizers differ from Emulators?
2. What are the advantages and disadvantages of Classical System VMs compared to Para-virtualized VMs?
3. What is a co-designed virtual machine? Briefly describe and give an example.
4. What type of virtual machine (VM) is each of the following and why? Be as specific as possible. (a) Java Virtual Machine (JVM) (b) VMWare (c) Xen (d) Digital FX!32 (e) VirtualPC (f) (e) Transmeta Crusoe (Code Morphing)
5. Explain the difference between the concepts of full-virtualization and para-virtualization, giving at least one example of both virtualization techniques.
6. When you have design a system that does emulation, under what circumstances would you opt for Interpretation and when would you opt for Binary Translation? Justify your answer.
7. Let’s say that you are asked to modify the Linux OS so that programs and libraries compiled on Windows OS could run natively on Linux, meaning they should be executed as normal programs (without using any emulator or virtual machine). What would be your high-level approach?

Answer:

* 1. Modify Linux kernel to interpret the system call made by windows libraries and call the corresponding Linux system call inside the kernel.
  2. Emulate the ABI (application binary interface = system calls + user ISA) of Windows on Linux.
  3. In addition, you’ll also have to interpret the format of Windows executable files and provide any additional external services.

1. What is the difference between a Type-1 hypervisor and a Type-2 hypervisor? Give examples

Answer:

* Type 1 (or Classical): The hypervisor directly controls the native hardware from boot time onwards and provides all the drivers and other necessary components to talk to hardware.
* Type 2 (or Hosted): The hypervisor runs within another commodity OS (the host OS) which controls the hardware. It borrows device drivers and other kernel components from the host OS. The hypervisor is smaller than in the Type-1 case. But the VMs incur more I/O overhead.

# Security

1. What is the difference between security and privacy? Are they entirely the same? Or entirely different?
2. Explain the three key principles of computer security?

Answer:

* Confidentiality: System should disallow unauthorized access to data.
* Integrity: System should prevent tampering (unauthorized modification) of data.
* Availability: System services and data should remain available to authorized users.

1. What is a threat model? What factors should you consider when defining threat model?
2. What hardware mechanism does x86 ISA provide to ensure that Operating System’s code and data are protected from user-level processes?
3. What is the role of privilege levels (defined by the ISA) in a computer system? How many privilege levels are defined in the x86 ISA? In which privilege level does the OS execute?
4. Explain the basic security mechanisms supported by (a) the CPU execution hardware, (b) Memory management hardware and software, (c) File system. Assume that the machine uses x86 ISA.

Answer:

* 1. CPU hardware supports multiple execution privilege levels. For example, x86 CPUs support four privilege levels 0, 1, 2, 3. Operating system code executes at privilege level 0, User applications execute at privilege level 3. EFLAGS register contains two bits that specifies the execution privilege level for the currently executing code.
  2. Memory segment descriptors and page table entries contain privilege level information which specifies what CPU privileges a code needs to access a particular memory area. For example, segment descriptors contain a few bits specifying whether the OS or the applications can access a particular memory segment.
  3. File systems associate access permissions with each file and also provide user accounts. The combination of user accounts and access permissions determines how files are accessed.

1. In x86, how does the MMU figure out whether a code currently executing on CPU has permissions to read/write to/execute a given address in memory?

Answer: CPU hardware supports multiple execution privilege levels. For example, x86 CPUs support four privilege levels 0, 1, 2, 3. Operating system code executes at privilege level 0, User applications execute at privilege level 3. EFLAGS register contains two bits that specifies the execution privilege level for the currently executing code.

Memory segment descriptors and page table entries contain privilege level information which specifies what CPU privileges a code needs to access a particular memory area. For example, segment descriptors contain a few bits specifying whether the OS or the applications can access a particular memory segment.

On each memory access, the two pieces of information: CPU execution privileges and memory segment/page-table privileges are matched to determine if access should be allowed.

1. What is authentication?
2. Describe different techniques to authenticate users.
3. What are some ways in which by which authentication mechanisms can be subverted?
4. What’s a computer virus? What’s a computer worm?
5. Explain a buffer overflow attack.
6. What is sandboxing? List two sandboxing mechanisms.
7. Explain Discretionary, Mandatory, and Role-based access control mechanisms.
8. What is meant by “trust” in computer security?

Answer: "Trust" (in computer security) means relying on a system component that is **assumed** to be remain secure and uncompromised during operation. Trust does not mean that the component is secure. It means that the user assumes that the component is secure.

1. Explain (a) trusted computing base (TCB) including why is it called “Trusted”, (b) Reference Monitor, and (c) relationship between TCB and reference monitor.

Answer:

1. TCB is part of a computer system whose integrity is assumed to be foolproof and on which correctness of rest of the system depends. For example, the operating system is part of the TCB. Normally, the root user is also part of the TCB. Its called “trusted” because the TCB is assumed to be designed and implemented correctly and its integrity is normally verified offline, either manually or through automated tools.
2. Reference monitor enforces the security policies (such as access control policies) of the computer system. Its also called the security kernel. Its part of the TCB. Ideally, a minimal operating system would consist only of the reference monitor.
3. Reference monitor is a subset of the trust computing base. Reference monitor executes as part of the operating system in privileged mode and OS constitutes a large part of the TCB.
4. Explain the two key data access principles of multi-level security (MLS) systems (also called Mandatory Access Control).

Answer:

No READ UP: A user at lower security level should not be able to read data at a higher security level.

No WRITE DOWN: A user at a higher security level should not be able to write data to a lower security level.

1. Why is Mandatory access control called “mandatory”? What’s the alternative?
2. What type of systems require mandatory access control?
3. Give an example of a scenario where the software doesn’t trust the OS, hypervisor, and/or the hardware platform on which it runs? What can the software possibly do to “secure” itself in this situation?
4. Considering memory protection, explain how the operating system ensures that user-level processes don’t access kernel-level memory?

Answer:

Consider systems in which the OS maps itself into the address space of each process. OS resides in a more privileged segment with privilege bits in the segment register set to 0 (bits 00). The user code and data resides in a less privileged segment in the higher address space with the privilege bits in its segment register set to 3 (bits 11). User code executes with privilege level 3 (in CPU’s EFLAGS register). So if the user process tries to access kernel code and data (which has protection bits 00), then the MMU raised an exception indicating a segmentation violation, the OS gets control, and possibly kills the process.

Alternatively, in some systems, the OS resides in a separate address space by itself and is not mapped to the individual process address spaces. IN such cases, a process cannot refer to OS memory addresses by design.

# CPU Scheduling

1. What is a CPU scheduler? When does it execute?
2. What is the difference between a CPU Scheduler and a Dispatcher?
3. How does the operating system (or hypervisor) maintain control of the CPU? In other words, how does the OS prevent a process, such as a *while(1);* loop, from indefinitely running on the CPU without returning control back to the OS?
4. Give at least three mechanism(s) by which the highest privileged software, such as an operating system or a hypervisor, retains control over the CPU? Which mechanism is absolutely essential for the OS/hypervisor to retain control over CPUs? Why?

# Input/Output I/O

1. What is an interrupt? What is an interrupt handler?

Answer: An event from a hardware device to the CPU. Interrupt handler is a kernel function that responds to the event.

1. What are character, block, and network devices? Give examples.
2. What are the factors affecting read/write latency in traditional magnetic hard disks?
3. Why is IOPS (I/O operations per second) considered a better measure of hard disk performance than raw bandwidth (bytes per second)?
4. Explain therole of *on-board disk cache* in hard disks during (a) read I/O operations and (b) write I/O operations.