# Questions from Subject GRE

8, 17, 22, 23, 34, 44, 45, 46, 54, 55, (61), 65, 67

# Metrics

1. Solve the following:
   1. How much is 2^13 in decimals?
   2. How much is roughly 1 billion in power of 2?
   3. How much is 2^64/2^21 in power of 2?
   4. How much is 128MB/4KB ?
   5. How much is log2(8192)?
   6. 230bytes of storage equals?
   7. 1 Gbps network bandwidth equals how many bits per second (bps)?

Answer:

* + 1. 8 \* 1024 = 8192
    2. 2^10 \* 2^10 \* 2^10 = 2^30
    3. 2^(64-21) = 2^43
    4. 2^(7-2)KB = 2^5KB = 32K
    5. log2(8192) = log2(2^13) = 13
    6. 1GB
    7. 1000x1000x1000 bps = 109 bps

1. When measuring I/O throughput, what is the difference between the units
   1. MBps and Mbps
   2. KBps and Kbps?
2. How is a Mebibyte different from Megabyte?
3. How much are these units in decimals?
   1. Pico
   2. Nano
   3. Micro
   4. Milli
   5. Kilo
   6. Mega
   7. Giga
   8. Tera
   9. Peta

Hint:

For metric system: See <http://www.chemteam.info/Metric/Metric-Prefixes.html>

For size of information in computers see: <https://web.stanford.edu/class/cs101/bits-gigabytes.html>

1. Replace “?” below with the correct answer
   1. 1 Nanosecond = ? seconds
   2. 500 Milliseconds = ? seconds
   3. 4 KB = ? bytes
   4. 4 Kilometers = ? meters
   5. 4Kbps = ? bits per second

Answer:

* 1. 1 Nanosecond = 10^(-9) seconds
  2. 500 millisecond = 0.5 seconds
  3. 4KB = 4096 bytes
  4. 4 Kilometers = 4000 meters
  5. 4Kbps = 4000 bits per second

# 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)

Answer:

* + ISA: The specification of instructions supported by a given architecture. This is the hardware-software interface.
  + User ISA: Set of instructions that user-level unprivileged code is allowed to execute
  + System ISA: Set of instructions that only the privileged OS code is allowed to execute.
  + ABI: The runtime interface seen by programs during execution. It consists of the combination of system calls and user ISA.
  + API: The programming interface used by developers to write programs. It consists of libraries and user ISA.

1. 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)?
   1. Answer: The system calls supported by Windows and Linux are different, hence their ABIs are different. A program compiled on Linux would have a different ABI than required to run on Windows. In addition, the format of executable files may also be different. But even if we can somehow make the executable format the same, the ABIs would still be different across operating systems.
2. What is meant by virtualization? Give examples of many(virtual)-to-one(physical, one-to-many, and many-to-many resource virtualization.
3. What was the first computer? First OS? First programmer? First language?

# Processes

1. (a) What is a process? (b) How is a process different from a program?

Answer: A process is a program in execution. When a binary executable file on the disk is loaded into memory and executed, a process is created. A program is just a passive binary file on the disk, whereas a process is an actively executing entity. Besides instructions, a process state contains static data, dynamically allocated memory (heap), call stack, registers, program counter, stack pointer, open files, open sockets, etc.

1. In the memory layout of a typical process, why do stack and heap grow towards each other (as opposed to growing in the same direction)?
2. In terms of call-return behavior, how are the fork() and exec() system calls different from other system calls?

Answer: Fork returns twice if successful, once in parent, and once in child. Exec never returns, if successful.

1. Describe the process lifecycle illustrating the states and transitions.

Answer: See class slides.

1. **[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. What is a zombie process? Why does the Operating System maintain the state of zombie processes? List two ways in which a parent process can prevent a child process from becoming a zombie.
2. Why are frequent context switches expensive in terms of system performance?
3. What is cold-start penalty? What are some ways to reduce it?
   1. It is the performance penalty right after a context switch. Immediately after a context switch, the TLB and hardware caches do not have contents reflecting the newly scheduled process. The entries in the TLB may be flushed or may belong to the previous process. The hardware caches may have data belonging to old processes, not the new one. So the new process encounters a number of cache misses right after it starts executing. Since cache miss handling is slow, the process will execute very slowly right after the context switch.
   2. Have fewer context switches. Use tagged TLB. Use superpages to increase TLB coverage. Increase the sizes of TLB and L1, L2,and L3 caches
4. What are some key factors that affect application performance after a context switch?

# Threads

1. What are threads? How do they differ from processes? How are they similar?

Ans: B. A thread is an execution context (or ﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿an execution stream) within a process having its orn program counter, registers, and stack. There can be multiple threads within a process, each sharing a common address space of the process.

1. What state do threads share? What state is different?
2. Why does context-switching between threads incur less overhead than between processes?
3. Briefly explain
   1. User-level threads
   2. Kernel-level threads
   3. Local thread scheduling
   4. Global thread scheduling

Answer:

* 1. User-level threads: Threads that are supported in user-level libraries instead of the operating system. Process gets a timeslice from the kernel and the threads library divides up the time among threads. Blocking calls by any thread block the entire process. Inter-thread context switching overhead is low (among threads of the same process).
  2. Kernel-level threads: Threads that are supported by the operating system. The CPU scheduler in the OS allocates timeslice to each thread. Blocking calls by one thread does not block the entire process. Inter-thread context switching overhead is higher among threads of the same process due to user-kernel transitions.
  3. Local thread scheduling: If a thread yields the CPU and the process still has unused time, then another thread from the same process is scheduled.
  4. Global thread scheduling: If a thread yields the CPU, then the CPU scheduler can pick a thread from any process even though the current process may have unused time.

1. Threads vs processes
   1. When would you (as a programmer) prefer to use multiple threads instead of multiple processes?
   2. When would you prefer to use multiple processes instead of multiple threads (in one process)?

Ans:

* + 1. When we need true concurrency, but the overhead of spawning multiple processes and inter-process communication cost may be too high. Threads can reduce both overheads.
    2. When we need true concurrency, but we want the application to survive crashes or bugs. A bug in one thread will affect all threads in a process. However, a failed process won't affect other concurrent processes.

1. What are the benefits and disadvantages of using user-level and kernel-level threads?
2. What combinations or user/kernel threads and global/local scheduling are feasible and why?

Answer: User-level threads can only support local thread scheduling because threads libraries don’t have visibility across process boundaries and the OS is unaware of process-level threads. Kernel-level threads can support both local and global thread scheduling because they have full visibility into and control of threads in each process.

1. What kind of applications benefit the most from kernel-level threads support? What kind of applications benefit most from user-level threads support? Explain why with examples?
2. Explain how a web server could use threads to improve concurrency when serving client requests.
3. What happens if a thread in a multi-threaded process crashes? How can you improve the robustness (fault-tolerance) of a multi-threaded application?
4. Event-driven programming
   1. What is the “event-driven” programming model?
   2. What does the structure of a typical event-driven program look like?
   3. When would you prefer an event-driven programming model over a thread-based programming model?
   4. When would you prefer thread-based programming model over event-driven programming model?

Answer

* + 1. A way of programming in which multiple events are continuously monitored (or polled). Whenever an event happens then a task is executed. If multiple events happen, then tasks are executed sequentially. Each task is executed to completion.
    2. while(1) {

if (event 1) do task 1;

if (event 2) do task 2;

…

if (event N) do task N;

}

* + 1. When tasks are simple, short and stateless. Also when you want to avoid dealing with concurrency-related synchronization issues such as locking, deadlocks, and race conditions.
    2. When some tasks invoke blocking operations that can delay the processing of other tasks in a sequential event-driven model. Also when multiple CPUs are available for parallel processing.

1. What is the problem with long-running event handlers? How do threads solve this problem?
2. What type of applications would be more suitable for thread-based programming compared to event-driven programming?
3. What are callbacks and what problems can they cause when used with threads?
4. (a) How are threads better than processes?

(b) How are processes better than threads?

Answer: Threads within a process share the memory address space. So the context switching overhead is lower between threads of the same process.

Processes are more robust because they have different address spaces. So failure of one process does not affect another process.

1. Assume a single-CPU system. You are given three multi-threaded processes. P1 does a lot of computation, but little I/O. P2 does lots of I/O but little computation. P3 does a reasonable mix of both computation and I/O. What kind of threads would you prefer for each process? Explain why?

Answer:

P1: We’ll prefer user-level threads because its threads do not block much and user-level threads have a lower context switching overhead.

P2: We’ll prefer kernel-level threads because because when threads block on I/O, other threads from the same process can be scheduled and can make progress.

P3: We can use hybrid of kernel- and user-level threads. Kernel-level threads separate I/O bound threads. One kernel-level thread can be designated to handle multiple user-level threads that run CPU-bounds tasks.

# Inter-Process Communication

1. List any five inter-process communication mechanisms, with a one line description for each?

Answer:

* 1. Pipes: A unidirectional communication mechanism between two related processes such as parent and child, or siblings.
  2. Signals: An event notification mechanism between processes
  3. Shared memory: A mechanism for two or more processes to share a common piece of memory to which they can read or write directly without invoking system calls.
  4. Sockets: A bi-directional communication mechanism between any two processes, even if they are on different machines.
  5. Common files on the disk: Different processes can exchange data by reading or writing to common files
  6. Message queues: A FIFO mechanism for any two processes to communicate. Also called named pipes.
  7. Semaphores: A synchronization mechanism by which different processes can coordinate upon specific events.

1. When using a pipe for inter-process communication, why should a process promptly close any unused write descriptors to the pipe? Also give an example of what happens if it doesn’t.
2. Let's say a **chain of filters** refers to a series of commands whose standard inputs and standard outputs are linked by pipes. For example,

“ps -elf **|** grep bash **|** more”

is a chain with three commands.

In the general case,

“command1 | command2 | command 3 | ... | command K”

is a chain of filters with K commands.

Suppose you were implementing a shell (e.g. csh, bash, tcsh, ksh, etc.), how would you go about supporting a chain of filters with ***arbitrary*** number of commands? Explain. Don’t write actual code.

1. What’s the difference between byte-stream vs. message oriented communication?

Answer:

* Byte-stream communication: Provides no explicit message boundaries. Processes can read/write at any arbitrary byte granularity. E.g. Pipes, TCP connections.
* Message-oriented communication: Processes communicate using explicit messages. E.g. UDP sockets.

1. How would you perform input/output redirection from/to a file from a process?

# System Calls, Traps, Exceptions

1. What is the difference between a system call, hardware interrupt, a software interrupt (trap), and an exception? Give examples of each.

Answer:

System call a is an explicit request from a user process for a kernel-level service. E.g. request to read data from a file or device.

Hardware Interrupt is an event generated by a device controller to inform the CPU that a device-related event has occurred. E.g disk or network I/O interrupt, timer interrupt, etc

Exception is an event generated by the CPU’s own execution hardware in response to incorrect execution by a program. E.g. divide by 0 or segmentation violation.

Software interrupt (trap) is an event generated by the CPU’s own execution hardware in response to explicit instructions. E.g. SYSENTER, VMCALL, int x80, etc.

1. What is a system call? How is a system call different from a normal function call?

Answer: They are different in how they are invoked and executed.

* Normal function calls are invoked by jump instructions to user space code. System calls trap to the kernel (either explicitly via int 0x80 or implicitly via SYSENTER instruction), which then invokes the corresponding system call routine.
* Normal function calls are executed in user mode. System calls are executed in the kernel (supervisor) mode.

1. How are trap handlers, exception handlers, and interrupt handlers different from system calls?

Answer: Trap/exception handlers are kernel functions that execute when the CPU generates a trap. Interrupt handlers are kernel functions that execute when hardware devices deliver interrupt to the CPU. OS gets control on a trap/exception/interrupt, figures out what type of event occurred, and calls the corresponding handler. In contrast, a system call is an explicit request from a user process for a kernel-level service. System calls are invoked on modern hardware using either dedicated instructions, such as SYSENTER on x86 systems, or using traps on legacy processors that did not have dedicated instructions for system calls.

1. What steps take place when a system call is invoked by a process?
2. What is a system call table? Why is it needed? OR What role does it play in OS security?

Ans: A system call table is a kernel data structure (such as an array) that store the entry point (function pointer) for each system call. It is needed because user-space code lacks the privileges to directly invoke system call routines via their function pointers. Instead, user-space code references system calls by their system call number, which is used by the kernel code to index into the system call table to invoke the system call function.

1. Explain the CPU-privilege transitions during a system call.

Answer: When system call is invoked, the CPU transitions from user privilege mode (level 3 in x86) to kernel/supervisor privilege mode (level 0 in x86). Upon return from system call, the CPU transitions again from kernel privilege mode to user privilege mode.

1. (a) Why do some operating systems, such as Linux, map themselves (i.e. the kernel code and data) into the address space of each process? (b) What is the alternative?

Answer:

(a) So that system calls from user space don't require a context switch in the CPU and MMU.

(b) Alternative is to have the OS running in a different address space and perform a context switch upon each system call.

1. Assume a mainstream monolithic OS, such as Linux. When a process makes a system call, how can the system call mechanism avoid any context switching overhead between the calling process and the OS? (as opposed to the overhead seen when switching between two processes).

Answer: The OS would map itself into the address space of each process. Thus invoking a system call is simply jumping to another location in the same address space, with privilege escalation, and does not involve a context switch across different address spaces.

1. Rootkits (malicious code in the kernel) can intercept system calls made by processes (all processes or a specific process) and replace the original system call behavior with some other behavior.. How would you go about implementing such behavior? Describe the design but don’t write any code.

Answer: The basic idea is to write a kernel module (the rootkit) that obtains the base address of the system call table. The system call table contains pointers to individual functions that implement each system call. T**he kernel module then replaces these function pointers in the system call table with pointers to its own malicious functions**. These malicious functions can then intercept each system call made by user-level processes, examine the process ID, system call arguments etc. If the rootkit wants, it can replace the original syscall behavior with its own malicious behavior, else it can allow the original system call to proceed as intended.

1. How does the OS ensure that a user-level process does not jump to, and execute, arbitrary code in the OS memory?

Answer: All system call invocations are first handled by an entry point code in the kernel which verifies the system call number﻿﻿ being invoked﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿. The system call number is then used to index into a system call table, which contains pointers to specific system call routines that are pre-defined by the OS. Thus the user space process cannot jump to arbitrary points in kernel code because all kernel code invoked by system calls are directed through the system call table.

1. Does a system call invocation by a process trigger a context switch? Why or why not?

Answer: When the OS maps itself to the address space of each process, then the system call invocations do not trigger a context switch. However, if the OS resides in a separate address space, then system call invocations would require context switch from process address space to OS address space.

# 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.

# Superpages

1. What are superpages? Why are they useful? What are the constraints on their sizes, placement, and page attributes (protection, reference, and dirty bits)?

Answer:

Superpages are memory pages whose size is larger than a base page size, specifically multiples of base page size.

Superpages are useful because they increase the TLB coverage. Each TLB entry for a superpage can provide translation for a larger portion  of the main memory.

Superpages have the following constraints:

* Superpage sizes = some power of 2 \* base page size
* Placement: They must aligned at superpage-size boundaries in both virtual and physical memory. All base pages of a superpagemust be contiguous in both virtual and physical memory.
* Attributes: All base ﻿pages of a superpagemust have the same protection attributes and they share a common referenced (R) and modified/dirty (M) bit.

1. Superpages
   1. What are the advantages and disadvantages of superpages?
   2. How many TLB entries and page table entries are there for each superpage?
   3. What are the restrictions on superpage size, allocation, and placement?

Answer:

* 1. Advantage: Greater TLB coverage and contiguity of base pages (useful for greater spatial locality). Disadvantage: More internal fragmentation, restrictions on where superposes can be allocated in physical memory, external fragmentation when using mix of superpose sizes, and more page-in/page-out latency/overhead.
  2. One TLB entry per superpage. As many PTEs as the number of base pages in the superpage.
  3. Superpage must of power of 2 multiple of base page count. Base pages must be contiguous in virtual and physical memory. Superpage must be aligned at super page size locations.

1. If a superpage has a size equal to 4 base pages, how many TLB entries are occupied by the superpage? How many page table entries are occupied by the same superpage? Explain why.

Answer: 1 TLB entry that can cover all base bages. 4 PTEs, one for each base page so that base pages can be combined into a super page or a super page can be broken down back to base pages.

1. How do superpages affect (increase/decrease/don’t change) internal and external memory fragmentation? Why?
2. In the paper, superpage allocations are performed in two steps: preemptible reservations and incremental promotions. Why do we need this two-step mechanism? If we already know how much to reserve, why not create the entire superpage in one shot the first time any of its base pages is accessed?
3. Why is it that using a mix of multiple superpage sizes tends to yield better application speedups than using superpages of only one size?
4. In the superpage paper, explain how ***preemptible reservations***, ***incremental promotions*** and ***speculative demotions*** work.

OR

In the “superpages paper” (Navaro et. al.), when and how are (a) reservation, (b) promotion, and (c) demotions carried out? (d) Why are reservations called “pre-emptible”?

Answer:

**Reservations**: Space for superpages is reserved by OS using **pre-emptible reservation** mechanism. When a process accesses a memory object, the OS reserves a set of contiguous and aligned base pages that matches the expected object size. The expectation is that the process will quickly access the reserved pages, in which case they will be promoted into a superpage. If the process does not use its reservation, and then the OS can pre-empt (or cancel) the reservation to make another superpage reservation.

**Promotions**: The OS uses **opportunistic promotion** policy, meaning that once a set of contiguous and aligned reserved pages have been accessed, the OS will promote them to the nearest superpage size, even if all base pages in the reservation have not been accessed. Superpage size this grows as more contiguous base pages are accessed.

**Demotions**: Superpages may be demoted back to base pages using **speculative demotions.** This means that if the OS notices memory pressure, it may demote superpages back to base pages and incrementally re-promote them as base pages are accessed. This allows the OS to avoid swapping out large superpages (which is expensive), and also accurately identify which base pages are most recently used, and hence must be kept in memory.

1. Why do superpages make both internal and external fragmentation worse?

OR

Which fragmentation (Internal/external) is made worse by superpages? Why?

Answer:

Both internal and external fragmentation are made worse. Internal fragmentation is worse because pages are large, so there is a higher chance that an allocated memory page may not be used. External fragmentation becomes worse because the operating system may allocate superpages of different sizes. Thus it may become difficult over time to find enough base pages that are contiguous (next to each other) to satisfy a superpage allocation request.

1. In the superpage paper, how does the system proposed avoid the need to page-out an entire superpage to the disk upon memory pressure? What is the source of performance overhead in this mechanism?

Answer: When memory pressure increases, OS marks all superpages as read-only. When processes try to write, the OS handles the write exception. At this point, the OS demotes the superpage to base pages around the location of the write operation and to smaller superpages elsewhere. For instance, a write to the 5th base page in a 8-page superpage would result in demoting the 8-page superpage to super gapes of sizes 4, 1, 1, and 2.

The overhead of this mechanism is that overwrite operation to a superpage becomes expensive (when there is memory pressure) because all writes are trapped by the OS for superpage demotion.

1. Use of superpages (as they are currently designed) requires the use of contiguity restoration techniques in the operating systems. To avoid the need for contiguity restoration, one could get rid of the requirement that the base physical pages of a superpage be contiguous - in other words, allow the base pages in physical memory not to be next to each other. If one does so, describe how you would redesign (if at all) the TLB, page-tables, and the mechanism for translating virtual to physical addresses? You can assume either architected page-tables or architected TLB. Remember to list any assumption you make, justifying why they are reasonable.

# 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 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?

Answer:

Three mechanisms: Interrupts (including timer interrupt), System Calls, and Exceptions.

Control over timer interrupt is essential. The OS controls the ability to program the timer interrupt as well as execute the timer interrupt handler. Thus the OS can preempt runaway (long-running) processes during the timer interrupt handler.

1. (a) Explain the exponential averaging technique for predicting the length of the next CPU burst for a process. (b) Why is it necessary to “predict” the next CPU burst time (as opposed to using its exact value)?

Answer:

* 1. Predicted(n+1) = alpha x Actual(n) + (1-alpha) x Predicted(n) where Predicted(k) is the predicted CPU burst at scheduling instant k and Actual(k) is the actual measured CPU burst at scheduling instant k. The factor alpha is a value between 0 and 1 and is used to control how much of past execution history affects the predicted value.
  2. It is necessary to predict because it is impossible to exactly tell in advance how long a given set of instructions may take to execute.

1. What is round-robin CPU scheduling? Explain the benefits and drawbacks of selecting a large time quantum versus a small time quantum in round robin..
2. What is starvationin CPU scheduling? What type of scheduling policies can result in starvation? Give at least one solution to avoid starvation.
   1. Starvation happens when a process is not scheduled indefinitely because other processes keep getting selected by the scheduler.
   2. Usually priority-based scheduling policies lead to starvation because a constant stream of high priority processes can prevent lower priority processes from getting the CPU.
   3. Aging is a solution. The longer a process waits in the ready queue, the higher its priority becomes, till it gets sufficient priority to be scheduled. The other solution is to use scheduling algorithms that don’t have starvation, like Round Robin.
3. What is the difference between a work-conserving and a non-work-conserving resource scheduler? Explain with an example each for CPU and memory resources.
4. What is the difference between work-conserving and non-work-conserving CPU scheduling strategies? In what situations would you use non-work-conserving CPU scheduling?
5. What is fair scheduling? Explain with an example.
6. Consider the following process arrival times and CPU burst times in a system.

*P1* :Arrival time = 0.0 CPU burst = 8 time units

*P2* :Arrival time = 2.0 CPU burst = 6 time units

*P3* :Arrival time = 4.0 CPU burst = 4 time units

P4 :Arrival time = 6.0 CPU burst = 2 time units

***Provide the execution schedule (GANTT chart)*** and ***Calculate Average wait times***  for the above workload, with each of the following scheduling algorithms. Ignore context switching times.

* 1. First Come First Serve (FCFS)
  2. Non-preemptive Shortest Job First (SJF)
  3. Preemptive Shortest Job First (SJF)
  4. Round Robin with time quantum = 1 unit. (Assume that newly arriving jobs are queued at the end of the round robin queue)

1. Four processes— P1, P2, P3, and P4 —are scheduled to use a single CPU. The arrival time and duration of each of the four processes are (P1 0 7), (P2 2 4), (P3 4 1), (P4 5 4). If the system uses a preemptive shortest-job-first scheduling algorithm, what is the average waiting time for the four processes, in seconds?
2. **[10 pts]** Four processes— P1, P2, P3, and P4 —are scheduled to use a single CPU. The arrival times and CPU burst times of the four processes are (P1 0 8), (P2 2 6), (P3 4 4) and (P4 6 2). **Provide the execution schedule and calculate average wait times** for the above workload, with each of the following scheduling algorithms. Ignore context switching times.
   1. First Come First Serve (FCFS)
   2. Non-preemptive Shortest Job First (SJF)
   3. Preemptive Shortest Job First (SJF)

Answer:

* 1. First Come First Serve (FCFS)

0(1) 1(1) 2(1) 3(1) 4(1) 5(1) 6(1) 7(1) 8(2) 9(2) 10(2) 11(2) 12(2) 13(2) 14(3) 15(3) 16(3) 17(3) 18(4) 19(4)

Avg wait time = (0 + 6 + 10 + 12) / 4 = 28 / 4 = 7

* 1. Non-preemptive Shortest Job First (SJF)

0(1) 1(1) 2(1) 3(1) 4(1) 5(1) 6(1) 7(1) 8(4) 9(4) 10(3) 11(3) 12(3) 13(3) 14(2) 15(2) 16(2) 17(2) 18(2) 19(2)

Avg. wait time = (0 + 12 + 6 + 2) / 4 = 20/4 = 5

* 1. Preemptive Shortest Job First (SJF)

Either

0(1) 1(1) 2(1) 3(1) 4(1) 5(1) 6(1) 7(1) 8(4) 9(4) 10(3) 11(3) 12(3) 13(3) 14(2) 15(2) 16(2) 17(2) 18(2) 19(2)

Avg wait time = (0 + 12 + 6 + 2) / 4 = 20/4 = 5

OR

0(1) 1(1) 2(2) 3(2) 4(3) 5(3) 6(4) 7(4) 8(3) 9(3) 10(2) 11(2) 12(2) 13(2) 14(1) 15(1) 16(1) 17(1) 18(1) 19(1)

Avg Wait time = (12 + 6 +2 + 0) / 4 = 20/4 = 5

# Concurrency

1. Define Concurrency. How does it differ from parallelism?

Answer:

**Concurrency**: When multiple flows of execution (processes, threads, interrupt handlers, signal handlers etc) execute independently, or with some synchronization, on one or more CPUs

**Parallelism**: Same as concurrency, except that there are more than one CPUs in the system, allowing multiple tasks to make progress simultaneously (in real time).

1. Explain the differences between apparent concurrency and true concurrency.

Answer:

Apparent concurrency = single thread executing multiple tasks in a loop, possibly using select

True concurrency = two or more independent threads of execution. When one thread blocks, other threads can continue execution

1. What is “atomicity” and why is it important? (in concurrency, not physics!)

Answer: Atomicity means that if an instruction/operation consists of a group of steps, then all the steps are executed together without pre-empting the thread/process mid-way through the steps. It is important to guarantee that race conditions do not occur during the execution of multi-step operations due to pre-emption.

1. Briefly explain with examples
   1. Critical Section
   2. Race condition
   3. Deadlock

Answer:

* + 1. A piece of code that accesses or modifies a shared resource. E.g. when a counter is incremented by multiple threads, or a linked list is traversed or modified by multiple threads.
    2. Incorrect behavior of a program due to insufficient synchronization around critical section. E.g. by not using locks around the critical sections described in previous answer.
    3. When multiple threads end up waiting for each other indefinitely because each is waiting for another thread to release a shared resource. E.g. If thread 1 is waiting for thread 2 to release a lock on linked list B and thread 2 is waiting on thread 1 to release a lock on linked list A.

1. What’s wrong with associating locks with code rather than shared resources?
2. Describe the behavior of (a) UP and DOWN operations on a semaphore, (b) WAIT and SIGNAL operations on a condition variable.

Answer:

* + 1. **UP (sem):** Increment sem by 1 and wake up any processes blocked on a DOWN operation on sem.
    2. **DOWN(sem):** If (sem > 0) then decrement the semaphore value by 1, else block the caller till some other thread performs an UP operation. After waking up, attempt DOWN (sem) again till semaphore is successfully decremented.
    3. **WAIT(c):** Release lock on the monitor and block the caller till condition c is satisfied. Upon being woken up, try to acquire the lock on the monitor again.
    4. **SIGNAL(c):** Wake up any process blocked on a WAIT operation on the condition variable c.

1. What is the producer-consumer problem (NOT the solution) and its three synchronization requirements?

Ans: Multiple producer and consumer threads execute concurrently. Producers produce some items and insert them to a shared buffer, which can hold a maximum of N items. Consumer remove items from the shared buffer and consume the item. The three synchronization requirements of this problem are:

1. The shared buffer must not be accessed by more than one thread at a time.
2. Any producer waiting for an empty slot in a full buffer must be woken up  by the consumer who removes an item and creates an empty slot.
3. Any consumer waiting for an item to be available in an empty buffer must be woken up by the producer who inserts a new item into the empty buffer.
4. Under what situations would you use the following locks and how?  In your answer, consider whether the executing code is in process context or interrupt context, and also whether it is running on a multi-processor (SMP) machine.
   1. Blocking locks
   2. Non-blocking locks
   3. Spin locks.

Answer:

* 1. Blocking Locks: When the caller can afford to wait indefinitely for the lock to be available and when the correctness of the code depends upon lock being acquired successfully. Blocking locks should not be called in interrupt context. They can be used in SMP machines.
  2. Non-blocking locks: When the caller cannot afford to wait indefinitely for the lock to be available and when the caller has a fallback option if the lock is unavailable. Non-blocking locks can be used in interrupt context, and also on SMP machines.
  3. Spin Locks: Like blocking locks, the caller can afford to wait indefinitely for the lock to be available and the correctness of the code depends upon lock being acquired successfully. However, the lock is expected to available quickly. So the overhead of giving up the CPU and re-acquiring it is not worth. Hence the caller spins on the lock. Useful only in SMP systems. Can be used in interrupt context in SMP systems if the lock holder is running on another CPU.

1. When should you NOT use (a) blocking locks, (b) non-blocking locks, and (c) spin-locks?
2. (a) Consider (a) Blocking locks, (b) Non-blocking locks, and (c) Spin locks. Which of these locks can be used in interrupt handlers and how?

Answer: Non-blocking locks can be used in interrupt handlers as long as the code correctly﻿﻿ handles the case﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿ where lock acquisition fails.  Spin-locks may be used ﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿in interrupt handlers only on multi-processor (SMP) systems, and then also only when the critical section is expected to be very ﻿short.﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿

1. What is deadlock? How can you prevent it?

Answer: Deadlock occurs when multiple processes (or threads)﻿ attempt to acquire multiple locks, but do so in different order of locks. For example, if Process A acquires L1 first, then tries to acquire L2, and process B acquires L2 first and tries to acquire L1, then A and B might end up in a deadlock. ﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿

Deadlock can be prevented using lock ordering, i.e., by requiring that all processes acquire locks in a pre-determined order. So In the above example, if the lock order is L1 first, then L2, then deadlock can be avoided if process B attempts to acquire L1 first, then L2.

1. What is the main difference between a binary semaphore and a counting semaphore?
2. What are the following? How can you prevent each of them?

Answer:

* + 1. Incorrect behavior of a program due to insufficient synchronization around critical section.. Race conditions can be prevented by enforcing mutual exclusion in the execution of critical section code,  e.g. by  using locks around functions for inserting or removing items from a linked list
    2. Deadlock occurs when multiple processes (or threads) attempt to acquire multiple locks, but do so in different order of locks. For example, if Process A acquires L1 first, then tries to acquire L2, and process B acquires L2 first and tries to acquire L1, then A and B might end up in a deadlock.  Deadlock can be prevented using lock ordering, i.e., by requiring that all processes acquire locks in a pre-determined order. So In the above example, if the lock order is L1 first, then L2, then deadlock can be avoided if process B attempts to acquire L1 first, then L2.
    3. When a high priority process (Ph) is blocked waiting for a low priority process (Pl) to release a lock, and Pl is unable to run because another medium priority process (Pm) keeps pre-empting Pl. Priority inversion can be prevented by priority inheritance, i.e. by increasing the priority of Pl to the priority of Ph when Ph requests the lock being held by Pl. This way, Pm cannot pre-empt Pl while it is in the critical section.

1. Explain how a deadlock can occur in the operating system between code executing in the user-context and code executing in interrupt handlers. Also explain how you would prevent such a deadlock.

Answer:

Assume a system call acquires a lock and is then interrupted by a hardware interrupt. The interrupt handler runs (in the context of the interrupted process and tries to acquire the same lock. If the lock is blocking, then you have a deadlock.

To prevent this kind of deadlock, you can do a few things:

1. Don’t try locking in the interrupt context. Instead defer work to user context, where blocking locks can be safely acquired.
2. Try using non-blocking locks in the interrupt context, but make sure you correctly handle the case where lock acquisition fails.
3. Disable interrupts before entering critical sections that might trigger race conditions with interrupt context, and re-enable interrupts once you exit critical section.
4. Multiple processes are concurrently acquiring and releasing a subset of locks from a set of N locks L1, L2, L3, ….., LN. A process may try to acquire ***any subset*** of the N locks. What is the convention that all processes must follow to guarantee that there would be no deadlocks? Explain with an example where two processes need to acquire ***different but intersecting subsets*** of the N locks above.
5. How does the ***Test-and-Set Lock (TSL)*** instruction work? Why can’t we use separate LOAD and STORE instructions instead?

Answer: TSL instruction has the following format﻿﻿﻿﻿﻿

TSL Lock, Register

TSL instruction ﻿﻿atomically does two steps

(a) copies the old value of Lock to Register and

(b) sets Lock to 1

If the two steps are carried out using separate LOAD and STORE instructions then the process can be pre-empted between the two instructions, leading to race condition, where another process may concurrently perform LOAD and STORE operations on the same LOCK. This can lead to incorrect locking behavior.

1. Explain how you can implement the UP and DOWN operations on a mutex (binary semaphore) using the TSL instruction.
   * 1. See the pseudo-code in slides and in Tanenbaum’s book.
2. How does the **compare-and-set instruction** work? (b) How can you implement a DOWN operation on a mutex (binary semaphore) using a compare-and-set instruction (such as CMPXCHG in x86)?
3. Consider the classical producer-consumer problem. Producers produce items and insert them in a common buffer. Consumers remove items from the common buffer and consume them. In the following skeleton of pseudo-code, ***demonstrate the use of SEMAPHORES and MUTEXES*** to complete the pseudo-code for producer and consumer functions. Your code should have ***no race conditions*** and ***no busy loops***.

You can assume that the following functions are available to you. You shouldn’t need anything more than these functions in your pseudo-code.

**produce\_item**() produces and returns an item

**insert\_item**(item) inserts the item in the common buffer

**remove\_item**() removes and returns an item at the head of the buffer

**consume\_item**(item) consumes the item supplied

**up**(&semaphore) and **down**(&semaphore) have their usual meanings

==========================Pseudo-code Skeleton ===============================

#define N 100 /\* Number of slots in the buffer \*/

typedef int semaphore; /\* semaphores are a special kind of counter \*/

semaphore mutex = (***initialize this***); /\* figure out the role of mutex \*/

semaphore empty = (***initialize this***); /\* figure out the role of empty sem \*/

semaphore full = (***initialize this***); /\* figure out the role of full sem \*/

void ***producer***(void)

{

/\* ***complete this function*** \*/

}

void ***consumer***(void)

{

/\* ***complete this function too*** \*/

}

========================================================================

1. Consider the classical producer-consumer problem. Producers produce items and insert them in a common buffer. Consumers remove items from the common buffer and consume them. Complete the following skeleton pseudo-code to explain how you can solve the producer-consumer problem using **a monitor and condition variables.**

**procedure** Producer

**begin**

/\* complete this procedure \*/

**end**

**procedure** Consumer

**begin**

/\* complete this procedure \*/

**end**

**monitor** ProducerConsumer

**condition** /\* declare the condition variables you need \*/

**integer** /\* declare any other variables you need \*/

**procedure** insert(item)

**begin**

/\* complete this procedure \*/

**end**

**procedure** item \*remove()

**begin**

/\* complete this procedure \*/

**end**

**end** monitor

1. Consider the “events vs threads” argument in the context of monilithic operating system kernels (like Linux or Windows). (a) Which model do these operating systems primarily use -- events or threads? Why? (b) Let’s say you that have to design an operating system that uses the opposite model to what you just answered in (a). What would be the major design changes you would make to the kernel in terms of CPU scheduling, memory management, and I/O processing subsystems?
2. What are the tradeoffs in using semaphores versus monitors with condition variables?

Answer:

* 1. Semaphores enable more concurrency when each shared resource is tracked using a different semaphore. Semaphores provide greater concurrency because multiple producers/consumers can test the counting semaphores without acquiring the lock on the critical section. But they also increase programming complexity, leading to increased chances of race conditions and deadlocks.
  2. Monitors with condition variables provide less concurrency because a number of functions constituting the critical sections are guarded by a single lock. Monitors require the acquisition of the lock on the entire monitor before a the condition variable can be tested. However, they are simpler to reason and program, because only one procedure in the monitor is guaranteed to be active at a time, leading to fewer race conditions and deadlocks.

1. You are given a function f() in the Linux kernel that constitutes a **critical section**, i.e. no two parts of the kernel should execute f() concurrently. Assume that when the function f() is invoked anywhere in kernel, you call it using the following convention.

Do some form of locking;  
  
 Invoke function f()  
  
 Do some form of unlocking.

Explain what type of locking/unlocking mechanism would you choose under each of the following situations and justify your answer:

* 1. Function f() executes for a very **short** time. It can be called concurrently from two or more threads within the kernel (meaning either processes or conventional threads currently in the kernel context, such as within a system call), but **NEVER** from the within an interrupt context. (Interrupt context refers to the code that is executed immediately as a result of a hardware interrupt to the kernel, i.e. interrupt service routine, and also to the code that executes immediately following an ISR, but just before resuming the interrupted thread.)
  2. Function f() can execute for a very **long** time. Otherwise, just as in the previous case, it can be called concurrently from two or more threads within kernel, but **never** from the within an interrupt context.
  3. Function f() executes for a very **short** time. It can be called concurrently from two or more threads within kernel, and **ALSO** from the within an interrupt context.

Justify your answers, keeping in mind that the system can have either just a single-processor or multiple processors. Try to give the best possible locking mechanism, not just something that works. If possible, you can support your answer with real examples from within Linux source code where each of the above types of locking/unlocking approaches are used.

1. Explain how you can implement the WAIT and SIGNAL operations on condition variable using the TSL instruction.
2. In the lecture titled "Why threads are a bad idea" ,
3. what is the key reason that John Ousterhout discouraged the use of threads for most applications?
4. What are the drawbacks of thread-based programming that event-driven programming doesn't have?
5. When should you use threads, then?

Answer:

1. Because concurrent programming with threads can lead to many non-trivial race conditions and deadlocks.  It requires careful consideration of locking, synchronization, consideration of interaction with signals, and so forth. Most applications, such as GUI programming don't need this level of complexity.
2. Possibility of race conditions and deadlocks, which are hard to debug, due to absence of reliable debugging techniques.
3. When true concurrency or parallelism is required to improve performance.
4. When would you use a semaphore? When would you use a condition variable?

Answer:  Semaphore is ﻿﻿﻿used when you need fine-grained concurrency by locking individual resources. Condition variables are used to perform coarse-grained locking on many functions that constitute﻿﻿﻿﻿ critical section code (and data)﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿﻿.

# Input/Output I/O

1. How do CPU, memory, and I/O devices exchange data?
2. What’s the role of a device controller in the I/O subsystem?
   1. Device controller: Is an intermediary between I/O devices, CPU, and memory. Its job is to convey I/O requests from CPU to the device, convey I/O completions (e.g. interrupts) from device to CPU, and perform any data conversions and error correction.
3. How do CPU, memory, and I/O devices exchange data?

Answer: Using a system bus?

1. What’s a system bus? Why is its speed important?
   1. System Bus: Is a communication channel to allow CPU, Main memory, and I/O devices to talk to each other. i.e. exchange both data and control information. Its speed determines how quickly data can be moved in and out of the main memory.
2. 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. If there were no interrupts, what would be the alternative?
   1. Instead of using interrupts, the OS can periodically “poll” the device controller to check if there is an I/O event pending and process the event at its own pace.
2. What are the advantages and disadvantages of interrupts?
3. Briefly describe/compare (a) programmed I/O, (b) Interrupt-driven I/O, and (c) DMA (Direct Memory Access).
4. What does the OS do when an interrupt occurs?
   1. The OS
      * pre-empts any user-level process that may be executing
      * saves the state of the process on the interrupt stack,
      * determines what interrupt occurred
      * runs the interrupt service handler, which acknowledges the interrupt
      * returns from interrupt service handler
      * resumes the process that was interrupted by restoring its state from the stack.
5. Suppose that the device controller for a network card is generating so many interrupts (maybe millions per second) that the CPU is unable to meaningfully execute any user-level processes. How can the OS more efficiently handle process I/O operations in this situation?
   1. Instead of using interrupts, the OS can periodically “poll” the device controller to check if there is an I/O event pending and process the event at its own pace.
6. What are character, block, and network devices? Give examples.
7. What are the factors affecting read/write latency in traditional magnetic hard disks?
8. Why is IOPS (I/O operations per second) considered a better measure of hard disk performance than raw bandwidth (bytes per second)?
9. Describe the **Shortest Seek First (SSF)** and the **Elevator (SCAN)** algorithms for servicing disk I/O requests. Which of the two guarantees that there will be no starvation of I/O requests and why?
10. Explain therole of *on-board disk cache* in hard disks during (a) read I/O operations and (b) write I/O operations.

Answer:

**(a)** When a read I/O operations is requested by the OS, the disk head will read-ahead a few data blocksnext to the requested location on the disk. The additional blocks will be stored in the on-board disk cache. When the OS makes actual request for these blocks in the future, the blocks can be returned from on-board disk cache instead of the physical disk media.

**(b)** When a write I/O operation is issued by the OS, the block being written will be temporarily written to the on-board disk cache (which acts as a write buffer in this case) and an I/O completion interrupt will be sent back to the OS. The written block will be committed to the physical media at a later time.

1. How are solid-state drives (SSDs) better than mechanical hard drives? How are they worse?
2. What is wear leveling in SSDs?
3. Briefly explain the ideas behind Network Attached Storage (NAS) and Storage Area Networks (SAN).

# 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.

**Graduate**

1. Compare the following terms with examples: (a) Monolithic kernels, (b) Micro-kernels, (c) Hypervisors.
2. Explain how the "problem" instructions (i.e. sensitive instructions that do not generate a trap in user mode) are handled in hypervisors that provide (a) full virtualization and (b) para-virtualization.
3. Explain how a para-virtualized Xen hypervisor can allow a guest OS to manage its own hardware page tables while preventing it from mapping (and thus accessing) memory that belongs to another guest OS.
4. The Xen hypervisor allows each **para-virtualized** VM to manage its own page tables, meaning that the guest OS can create and modify its own page table mappings from virtual to physical memory addresses. (a) Since there may be multiple VMs running on a single physical machine, how can the hypervisor ensure memory isolation among VMs? In other words, how can the hypervisor guarantee that VM1 cannot map VM2's memory pages to its own address space without VM2's permission? (b) How does Xen's behavior in this respect differ from that of VMWare? (c) What is the advantage and disadvantage of Xen's approach to page-table management compared to VMWare's approach
5. Explain how virtualization techniques that provide full-virtualization (such as VMWare) can virtualize sensitive instructions that are not privileged?
6. How does Intel VT-x solve the problems of ring-deprivileging, ring aliasing, and address-space compression?
7. What is Virtual Machine Control Structure? What role does it play in virtualization?
8. (a) What is a shadow page table? How is it constructed and/or updated by the hypervisor? (b) What type of hardware virtualization support is needed to avoid constructing shadow page-tables in full-virtualization?
9. Xen hypervisor allows each guest OS to manage its own hardware page tables, which contain virtual to physical memory mappings. (a) Explain two mechanisms by which the Xen hypervisor prevents one guest OS from mapping (and thus accessing) a physical memory page that belongs to another guest OS. (b) What is the advantage and disadvantage of Xen's approach in this respect compared to that of VMWare?
10. In Xen’s device driver architecture, explain (a) page-flipping, (b) page-sharing, and (c) page-copying methods for processing I/O operations.
11. How do the following differ in the services they provide to user applications: (a) Monolithic kernels, (b) Micro-kernels, (c) Library operating systems.
12. Explain the protected control transfer mechanism in the Exokernel.
13. 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.

# Live Migration of VMs

1. What constitutes “liveness” during live migration of virtual machines?
2. What are the key states transferred during live VM migration?
3. What are the metrics of performance in live migration of VMs?
4. How do different VM migration techniques work? How do they differ?
   1. Stop-and-copy
   2. Pure demand paging
   3. Post-copy
   4. Pre-copy
   5. Hybrid pre/post copy
5. What are the tradeoffs between different performance metrics when using each migration technique?
6. What are some optimization techniques you can use to speed up pre-copy? Post-copy? Both?
7. Under what situation would you use each migration technique?
8. What is dirty-page tracking? Why is it needed? How does it work? What are its overheads?
9. Consider pre-copy live VM migration
   1. Under what situation would it have excessively long total migration time and downtime?
   2. To solve this problem, describe two optimization techniques (i.e. improvements to pre-copy — NOT post-copy, record-replay or other completely different techniques).
10. Design a system for maintaining a “hot-standby” replica of a running virtual machine (VM). In other words, if the VM fails, then its replica (on a different machine) should “instantly” take over the execution from exactly the point at which the original VM failed, all without losing correctness. Explain how you would maintain a consistent replica and switch over to the replica upon failure. In your answer, identify the design challenges you would need to solve and one or more approaches to solve each of the challenges.
11. Suppose you are asked to develop a system for simultaneous live migration of multiple VMs from one physical machine to another within a LAN. You need to reduce the total migration time (from start of migration of first VM to end of migration of last VM) and the network traffic due to migration, while not unnecessarily increasing the downtime. All VMs run the same guest OS (say Linux), but not necessarily the same applications. Develop a design for your live migration system in as much detail as you can. Justify the design choices you make.
12. For each of the following (a) what interfaces are virtualized? (b) Give an example.
    1. Process virtual machine,
    2. System virtual machine,
    3. Containers (OS-level virtualization)

Answer:

* + 1. Application Binary Interface (ABI)
    2. Instruction Set Architecture (ISA)
    3. Various namespaces such as PID, File System, network, user, IPC, etc.

# 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.

# I/O Models

1. In the I/O subsystem, explain the two stages by which data is delivered from an I/O device to a user-level process.

Answer:

First stage is the reception of data from an I/O device (or another process) into the kernel buffers. The second stage is copying the data from the kernel buffers to user space. The way these two stages are processed determines the category of I/O model used by a process.

1. What are the five I/O models? How do they handle the two stages of data reception?

Answer:

1. Blocking : The process blocks in both stages of data reception. The OS receives the data to kernel buffers, copies the data to user space and then returns back to user process.
2. Non-blocking: The process does not block during the first stage. Instead, it polls (queries) the OS periodically till the data is available, at which point it blocks for the second stage while the OS copies data to user space and returns back to the process.
3. Signal-driven: The process does not block in the first stage. However it also does not poll the OS. Instead, the OS delivers a signal to the process when the data is ready, at which point the process calls the read system call and copies OS copies data to user space. The second stage blocks the process till data is copied.
4. Asynchronous: The process does not block in both stages of data reception. The OS receives the data and copies data to user space, while the process continues to run and do other things in the background. After copying the data to user space, the OS delivers a signal to notify the process of I/O completion.
5. I/O multiplexing: This is similar to blocking model. However, during the first stage, the process can block on I/O on multiple file descriptors using the select() system call. If I/O event occurs on any descriptor, the OS returns to the process form the select() call. The process then tests each file descriptor and handles the I/O on ready descriptors in a blocking manner during the second stage. The first stage blocks on multiple descriptors simultaneously. The second stage blocks on each ready descriptor one by one.
6. Why is process blocking generally not a concern for *write* I/O operation?
7. When can a process block on a write I/O operation?
8. Compare the blocking I/O model with the I/O Multiplexing model? How are they similar and how are they different?

Answer: In the blocking I/O model, a process blocks in both stages of data reception, but only on a single I/O descriptor. The OS receives the data to kernel buffers, copies the data to user space and then returns back to user process.

I/O multiplexing is is similar to blocking model. However, during the first stage, the process can block on I/O on multiple file descriptors, typically using the select() system call. If I/O event occurs on any descriptor, the OS returns to the process from the select() call. The process then tests each file descriptor and handles the I/O on ready descriptors in a blocking manner during the second stage. The first stage blocks on multiple descriptors simultaneously. The second stage blocks on each ready descriptor one by one.

Additionally, I/O multiplexing using select system call can acquire a range of behaviors in the first stage from fully blocking, to fully non-blocking, depending upon the timeout value specified in the select call.

1. Which I/O model is used for event-driven programming? Explain why.
2. Way is asynchronous I/O? In practice, why is it hard for any OS to support true asynchronous I/O?
3. In this course you learnt two I/O classifications. Under “I/O Devices” lecture, you learnt about programmed I/O, interrupt-driven I/O, and DMA-based I/O. In “I/O Models” lecture, you learnt about blocking, non-blocking, signal-driven, and asynchronous I/O models. Explain the analogies and differences between these two I/O classification systems.
4. Explain the two key steps in delivering data in packets from a network card to a user-level process reading from a network socket?

Answer:

Step 1: Copying data from the network card to kernel buffers.

Step 2: Copying data from kernel buffers to process in user space code.

1. Explain the difference between (a) signal-driven I/O and asynchronous I/O, (b) blocking I/O and I/O multiplexing.

Answer:

1. Signal-driven I/O vs. Asynch I/O: For signal-driven I/O, the second stage (copying data to user space) blocks the process till data is copied. For asynchronous I/O, the second stage does not block the process.
2. Blocking I/O vs. I/O Multiplexing: The blocking I/O model blocks a process indefinitely on **only one I/O descriptor** at a time. With I/O Multiplexing, a process can block (indefinitely or for a specified time) on **multiple I/O descriptors** at the same time.

# OS-level virtualization

1. What is OS-level virtualization? OS-level as opposed to what other levels?
2. How would you define the notion of isolation for any software or system component? What are the two (or more) extremes?
3. (a) What are “containers”? (b) What type of virtualization do they provide? Explain.

Answer: "Container" are a form of operating-system-level virtualization technique. A container groups together multiple processes and restricts/virtualizes their view of certain system resources, such as process ID space, user ID space, file system, network ID, IPC etc.

1. How does the isolation model of containers differ from traditional system VMs and process VMs?

Answer: System VMs virtualize the ISA seen by the VM, hence they provide a stronger isolation than containers or process VMs. Process VMs virtualize the ABI seen by a single process. Containers virtualize specific namespaces seen by a group of processes.

1. How does the isolation model for a process differ from that for a system virtual machine?
2. How do containers improve upon process-level isolation model?
3. What are jails or sandboxes in the OS world? Why would you use them?
4. What are some key system resources that are virtualized by jails/containers/sandboxes?
5. Linux containers contain consist of two main components: Namespaces and CGroups. Explain the role of each.

# System Design Techniques

1. When you “design” a computer system (hardware or software), what are some common goals that you may have as a designer?
2. What is meant by performance metric? What are resource constraints? How do the two interact? Give some examples.
3. What are some common examples of resources to consider when designing a computer system?
4. What is a “balanced” system? Why is balance important?
5. What are the different ways one can achieve balance? What are the goals of each approach?
6. Explain each of the following concepts and the tradeoffs it offers, with examples
   1. Multiplexing
   2. Pipelining and Parallelism
   3. Batching (versus multi-programming)
   4. Caching (or exploiting locality)
7. What is the “80/20 rule”? (Why is it not the “90/10 rule”?!) Explain with examples.
8. Explain the notion of binding and indirection using virtual memory (or domain names or email aliases or something else) as an example.
9. What is the difference between abstraction and virtualization?
10. What are the benefits/drawbacks of hard state versus soft state design? Examples?
11. What is hysteresis? Explain how the page-out/page-in mechanism in the OS uses hysteresis and why?

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. If you were to apply the principle of separating data plane and control plane to OS design, what OS components would you move out of the kernel to user space and why?
2. Suppose you have a computer with a really fast processor, moderately fast system bus, and a really slow disk. How would you balance the computer for
   1. Performance?
   2. Cost?
   3. Both performance and cost?
3. What are the (a) performance metrics, and (b) resource constraints when designing the following systems?
   1. CPU
   2. Memory
   3. Network
   4. Storage
   5. A supercomputer

# Kernel Modules

1. Why are memory access errors (such as segmentation fault and null pointer dereferencing) more dangerous in kernel space than in user space?
2. What are kernel modules?

Answer:

Kernel modules are dynamic extensions to the core kernel’s functionality. They can be added to the core kernel on-demand, instead of being compiled into kernel statically.

1. What are some advantages of kernel modules?

Answer:

* 1. Enable third parties to add features to OS, such as device drivers or file-systems
  2. Reduce kernel image size
  3. Reduce boot time by loading modules later (on-demand)