Review of Midterm Exam for COP 4610 Operating Systems - Fall 2025

Name: _____ **Student ID:** _____

Score: _____

Assignment 2: Minimal Hello World Program

1. (10 points) Write an x86-64 assembly program that prints "Hello, World!".

```
    .global _start
    .section .rodata
st:
    .ascii "Hello, OS World!"
ed:

    .section .text
_start:
    mov     $1, %rax            # sys_write
    mov     $1, %rdi            # fd = 1 (stdout)
    lea     st(%rip), %rsi      # buf = &st
    mov     $ed - st, %rdx      # count = len
    syscall

    mov     $60, %rax           # sys_exit
    xor     %rdi, %rdi          # status = 0
    syscall
```

A system call bridges user space and kernel space. In our minimal Hello World program, the user program prints "Hello, World!" by invoking the kernel's write system call, which is what printf ultimately relies on.

Assignment 3: Fork and Execeve

1. (10 points)

```
#include <unistd.h>
#include <stdio.h>

int main() {
    pid_t x = fork();
    pid_t y = fork();
    printf("%d %d\n", x, y);
}
```

Build and run fork-demo.c. Count how many processes run (2 pt). Explain why these values appear (8 pt).

Answer:

```
{
    "process_count": "4",
    "pairs_structured": [
        { "x": "7126", "y": "7127" },
        { "x": "0", "y": "7128" },
        { "x": "7126", "y": 0" },
        { "x": "0", "y": "0" }
    ],
}
```

- Each `fork()` duplicates the current process. Two calls create 4 processes.
- `x` holds the return of the first `fork()`: parent sees the child PID > 0, child sees 0.
- `y` holds the return of the second `fork()` in the same way.
- The four possible pairs are (pidA, pidB), (pidA, 0), (0, pidC), (0, 0).
- Numbers like `7126` and `7127` are child PIDs returned to the parent that performed that `fork()`, not the caller's own PID.

2. (10 points) Background: When you run the env command in your terminal, it prints all the environment variables of your current shell. These variables are passed from the shell (parent process) to the env command (child process). In other words, a parent process has full control over the environment of its children.

   Your Task: Complete the code in the execve-demo.c file to call the env program to print "Hello, World¡.

   Answer:

```c
#include <unistd.h>

int main() {
    char *const argv[] = {"env", NULL};
    char *const envp[] = {"Hello, World!", NULL};
    execve("/usr/bin/env", argv, envp);
    return 1;
}
```

   Unlike fork() which creates a new process, the execve() system call replaces the current process with a new program. When execve("/usr/bin/env", ...) is called successfully, the operating system stops the current program (execve-demo), loads the env program into the same memory space, and starts executing env from its beginning. The process ID (PID) does not change.

   The parent process has absolute control over the environment that the new program will see. The code doesn't just add a variable to the existing environment; it completely replaces it. The env program will wake up in a world where the only environment variable is "Hello, World!". This is why env will print only that string and nothing else.

Assignment 4: Process and Address

1. (7 points) During the boot sequence, what happens after each step? (Select the correct order for each step.)

   1. F jump to a fixed address in ROM
   2. D load an OS loader
   3. A load MBR (GPT) from the boot device
   4. B load the BIOS (UEFI)
   5. C load the kernel image
   6. G perform POST
   7. E set the kernel mode

   A. load an OS loader
   B. perform POST
   C. set the kernel mode
   D. load the kernel image
   E. jump to the OS entry point
   F. load the BIOS (UEFI)
   G. load MBR (GPT) from the boot device

2. (2 points) Suppose you have a 16-bit machine with a page size of 16B. Assume that any unsigned integer can be a potential memory address. Also, assume that the machine can support up to only 32KB of physical memory. ($1K = 2^{10}$).

   For paging-based memory allocation, how many page table entries (in decimals) do you need to map the virtual address space of a process, with pure paging alone?

   Answer: 4096

- Parameter 1: Virtual address size (16-bit machine)
  A 16-bit address implies that the virtual address space is 2^{16} bytes, which equals $65,536$ bytes (or 64KB).
- Parameter 2: Page size (16 bytes)
  Each page in memory is 16 bytes. To calculate the total number of pages in the virtual address space, we divide the virtual address space by the page size:

$$\text{Number of pages in virtual address space} = \frac{\text{Virtual memory size}}{\text{Page size}} = \frac{65,536}{16} = 4,096.$$

- Parameter 3: Physical memory size (32KB)
  While the machine has 32KB of physical memory, this parameter is not relevant for determining the number of page table entries. The page table is responsible for mapping the entire virtual address space, regardless of the size of the physical memory.

3. (3 points) Suppose you have a 16-bit machine with a page size of 16B. Assume that any unsigned integer can be a potential memory address. Also, assume that the machine can support up to only 32KB of physical memory. (1K = 2^{10}).

   Translate the virtual address $0x0000$ to its physical address (base 16) via the segmented-paging scheme. Assume that a program has two segments and a page size of 16B.

   Segment Table:

Segment Number	Page Table Base	Page Table Bound
0	$0x8000$	$0x8$
1	$0x0000$	$0x8$

   Physical Memory Address $0x0000$**:**

Virtual Page Number	Physical Page Number
$0x000$	$0x500$
$0x001$	$0x504$
$0x002$	$0x508$
$0x003$	$0x50B$
...	

   Physical Memory Address $0x8000$**:**

Virtual Page Number	Physical Page Number
$0x000$	$0x100$
$0x001$	$0x104$
$0x002$	$0x108$
$0x003$	$0x10B$
...	

   Answer: 1,000

   Step-by-Step Process:

   1. Understand the structure of the virtual address:
      The virtual address is composed of three parts:

      Segment Number (1 bit) || Virtual Page Number (11 bits) || Page Offset (4 bits)

      - The machine uses a total of 16 bits for its address. - The Segment Number is the most significant bit (1 bit). - The Virtual Page Number is the next 11 bits. - The Page Offset is the least significant 4 bits.
      For $0x0000$ (in binary: 0000 0000 0000 0000):

- Segment Number = 0 (most significant bit).
- Virtual Page Number = $0x000$ (next 11 bits).
- Page Offset = $0x0$ (last 4 bits).

2. Locate the segment table entry:
   Using the **Segment Number** 0, look up the corresponding entry in the segment table:

$$\text{Page Table Base} = 0x8000, \quad \text{Page Table Bound} = 0x8.$$

   This tells us that the page table for this segment starts at physical address $0x8000$.

3. Use the Virtual Page Number (VPN) to find the physical page:
   The Virtual Page Number $0x000$ is used to index the page table starting at $0x8000$. Since the **Virtual Page Number** is 0, the first entry of the page table contains:

$$\text{Physical Page Number (PPN)} = 0x100.$$

4. Combine the PPN and Page Offset to compute the physical address:
   The final physical address is computed by combining the **Physical Page Number (PPN)** with the **Page Offset**:

$$\text{Physical Address} = \text{PPN} \parallel \text{Page Offset}.$$

   Substituting the values:

$$\text{Physical Address} = 0x100 \parallel 0x0 = 0x1000.$$

Key Points:

- Virtual Address Structure: The virtual address is split into **Segment Number (1 bit)**, **Virtual Page Number (11 bits)**, and **Page Offset (4 bits)**.
- Page Table Address: The **Segment Number** determines the base address of the page table in physical memory. For $0x0000$, it points to $0x8000$.
- Physical Memory Access: The segment and page tables determine the mapping, and the final address combines the physical page number with the page offset.

4. (3 points) Suppose you have a 16-bit machine with a page size of 16B. Assume that any unsigned integer can be a potential memory address. Also, assume that the machine can support up to only 32KB of physical memory. ($1\text{K} = 2^{10}$).

   Translate the virtual address $0x0003$ to its physical address (base 16) via the segmented-paging scheme. Assume that a program has two segments and a page size of 16B.

   Segment Table:

Segment Number	Page Table Base	Page Table Bound
0	$0x8000$	$0x8$
1	$0x0000$	$0x8$

   Physical Memory Address $0x0000$**:**

Virtual Page Number	Physical Page Number
$0x000$	$0x500$
$0x001$	$0x504$
$0x002$	$0x508$
$0x003$	$0x50B$
...	

   Physical Memory Address $0x8000$**:**

Virtual Page Number	Physical Page Number
$0x000$	$0x100$
$0x001$	$0x104$
$0x002$	$0x108$
$0x003$	$0x10B$
...	

Answer: 1,003

Step-by-Step Process:

1. Understand the structure of the virtual address:
   In segmented-paging, the virtual address is divided into three parts:

   $$\text{Segment Number (1 bit)} \parallel \text{Virtual Page Number (11 bits)} \parallel \text{Page Offset (4 bits)}.$$

   For the 16-bit virtual address $0x0003$ (in binary: 0000 0000 0000 0011):
   - Segment Number: The most significant bit is 0, indicating **Segment 0**.
   - Virtual Page Number: The next 11 bits are $0x000$.
   - Page Offset: The least significant 4 bits are $0x3$.
2. Locate the segment table entry:
   Using the **Segment Number** 0, we find the corresponding entry in the segment table:

   $$\text{Page Table Base} = 0x8000, \quad \text{Page Table Bound} = 0x8.$$

   This tells us that the page table for this segment starts at physical address $0x8000$.
3. Use the Virtual Page Number (VPN) to find the physical page:
   The Virtual Page Number $0x000$ is used to index the page table starting at $0x8000$. Since the Virtual Page Number is 0, the first entry of the page table contains:

   $$\text{Physical Page Number (PPN)} = 0x100.$$

4. Combine the PPN and Page Offset to compute the physical address:
   The final physical address is computed by combining the **Physical Page Number (PPN)** with the **Page Offset**:

   $$\text{Physical Address} = \text{PPN} \parallel \text{Page Offset}.$$

   Substituting the values:

   $$\text{Physical Address} = 0x100 \parallel 0x3 = 0x1003.$$

5. (3 points) Suppose you have a 16-bit machine with a page size of 16B. Assume that any unsigned integer can be a potential memory address. Also, assume that the machine can support up to only 32KB of physical memory. ($1\text{K} = 2^{10}$).

   Translate the virtual address $0x8020$ to its physical address (base 16) via the segmented-paging scheme. Assume that a program has two segments and a page size of 16B.

   Segment Table:

Segment Number	Page Table Base	Page Table Bound
0	$0x8000$	$0x8$
1	$0x0000$	$0x8$

   Physical Memory Address $0x0000$**:**

Virtual Page Number	Physical Page Number
$0x000$	$0x500$
$0x001$	$0x504$
$0x002$	$0x508$
$0x003$	$0x50B$
...	

Physical Memory Address $0x8000$**:**

Virtual Page Number	Physical Page Number
$0x000$	$0x100$
$0x001$	$0x104$
$0x002$	$0x108$
$0x003$	$0x10B$
...	

Answer: 5,080

Step-by-Step Process:

1. Understand the structure of the virtual address:
   In segmented-paging, the virtual address is divided into three parts:

   $$\text{Segment Number (1 bit)} \parallel \text{Virtual Page Number (11 bits)} \parallel \text{Page Offset (4 bits)}.$$

   For the 16-bit virtual address $0x8020$ (in binary: 1000 0000 0010 0000):
   - Segment Number: The most significant bit is 1, indicating **Segment 1**.
   - Virtual Page Number: The next 11 bits are $0x002$.
   - Page Offset: The least significant 4 bits are $0x0$.
2. Locate the segment table entry:
   Using the **Segment Number** 1, we find the corresponding entry in the segment table:

   $$\text{Page Table Base} = 0x0000, \quad \text{Page Table Bound} = 0x8.$$

   This tells us that the page table for this segment starts at physical address $0x0000$.
3. Use the Virtual Page Number (VPN) to find the physical page:
   The Virtual Page Number $0x002$ is used to index the page table starting at $0x0000$. From the provided page table:

   $$\text{Virtual Page Number (VPN)} = 0x002 \implies \text{Physical Page Number (PPN)} = 0x508.$$

4. Combine the PPN and Page Offset to compute the physical address:
   The final physical address is computed by combining the **Physical Page Number (PPN)** with the **Page Offset**:

   $$\text{Physical Address} = \text{PPN} \parallel \text{Page Offset}.$$

   Substituting the values:

   $$\text{Physical Address} = 0x508 \parallel 0x0 = 0x5080.$$

6. (5 points) When starting a program, what happens after each step to switch from the kernel level to the user level? Match each step on the left with its corresponding action on the right.

Step	Action
Create a process and initialize the address space → A	A. Load the program into the memory
Initialize the translation table → B	B. Set the HW pointer to the translation table
Load the program into the memory → C	C. Initialize the translation table
Set the CPU to user mode → D	D. Jump to the entry point of the program
Set the HW pointer to the translation table → E	E. Set the CPU to user mode

Instead of focusing on matching, we should focus on the logical flow of actions that take place during this transition.

1. Create a process and initialize the address space
   At this step, the operating system sets up the **process control block (PCB)** and allocates an **address space** for the new process. This is the foundational step for any program to start execution.
2. Load the program into memory
   After the address space is set up, the program's binary is loaded into the **code segment** and its associated data into the appropriate memory regions. This ensures the program is ready to run.
3. Initialize the translation table
   Once the program is loaded into memory, the OS sets up a **translation table** (e.g., page tables) that maps virtual addresses to physical addresses. This step is critical for memory isolation and management in a virtual memory system.
4. Set the HW pointer to the translation table
   After the translation table is initialized, the **hardware pointer** (like the `CR3` register on x86) is updated to reference this table. This ensures that subsequent memory access by the program uses the correct virtual-to-physical address mapping.
5. Set the CPU to user mode
   Once the memory is set up, the OS switches the **CPU mode** from **kernel mode to user mode**. This is necessary to allow the program to execute with limited privileges for security.
6. Jump to the entry point of the program
   Finally, the CPU jumps to the **entry point** of the program (usually the `main()` function or a similar starting point), and the program begins its execution.

Assignment 5: Paging and Caching

1. (5 points) Match the terms on the right with the definitions on the left.

Definition	**Term**
A Allowing pages that are referenced actively to be loaded into memory	A. demand paging
B A referenced page is not in memory	B. page fault
C Adding more memory results in more page faults	C. Belady's anomaly
D When the memory is overcommitted	D. thrashing
E Pages that are referenced within the past T seconds	E. working set

2. (3 points) For the following reference stream, what is the content of page 2 at the end of the reference stream under the FIFO policy?

   A process makes references to 4 pages: A, E, R, and N.
   Reference stream: `E A R N R E A R N E A R`
   Physical memory size: 3 pages.

Reference Stream:

Memory Page	E	A	R	N	R	E	A	R	N	E	A	R
1												
2												
3												

☐ A ☐ E ☐ N ☒ R

Memory Page	E	A	R	N	R	E	A	R	N	E	A	R
1	E	E	E	N	N	N	N	R	R	R	A	A
2		A	A	A	A	E	E	E	N	N	N	R
3			R	R	R	R	A	A	A	E	E	E

3. (3 points) For the following reference stream, what is the content of page 2 at the end of the reference stream under the MIN (Minimum Page Replacement) policy?

Reference Stream:

Memory Page	E	A	R	N	R	E	A	R	N	E	A	R
1												
2												
3												

☐ R ☒ E ☐ A ☐ N

Memory Page	E	A	R	N	R	E	A	R	N	E	A	R
1	E	E	E	E	E	E	A	A	A	A	A	A
2		A	A	N	N	N	N	N	N	E	E	E
3			R	R	R	R	R	R	R	R	R	R

4. (3 points) For the following reference stream, what is the content of page 2 at the end of the reference stream under the LRU (Least Recently Used) policy?

Reference Stream:

Memory Page	E	A	R	N	R	E	A	R	N	E	A	R
1												
2												
3												

☐ N ☐ A ☐ E ☒ R

Memory Page	E	A	R	N	R	E	A	R	N	E	A	R
1	E	E	E	N	N	N	A	A	A	E	E	E
2		A	A	A	A	E	E	E	N	N	N	R
3			R	R	R	R	R	R	R	R	A	A

5. (3 points) For the following reference stream, what is the content of page 2 at the end of the reference stream under the LFU (Least Frequently Used) policy?

Reference Stream:

Memory Page	E	A	R	N	R	E	A	R	N	E	A	R
1												
2												
3												

☐ R ☐ E ☐ N ☒ A

Memory Page	E	A	R	N	R	E	A	R	N	E	A	R
1	E	E	E	N	N	N	A	A	A	E	E	E
2		A	A	A	A	E	E	E	N	N	A	A
3			R	R	R	R	R	R	R	R	R	R

6. (4 points) Match the terms on the right with the definitions on the left.

Definition	Term
___A___ Storing copies of data at places that can be accessed more quickly	A. caching
___B___ Recently referenced locations are more likely to be referenced soon	B. temporal locality
___C___ Reference locations tend to be clustered	C. spatial locality
___D___ Leaving behind cache content with no localities	D. cache pollution

7. (3 points) ♠ What is the effective access time of memory (in decimals) through L1 and L2 caches for the following hardware characteristics?

	Access Time	Cache Hit Rate
L1 Cache	1 clock cycle	75% or $\frac{3}{4}$
L2 Cache	3 clock cycles	75% or $\frac{3}{4}$
Memory	4 clock cycles	100% or 1

Answer: ___2___

Step-by-Step Calculation:

1. Calculate the Probability and Time for Each Scenario:
   We identify three possible outcomes for any given memory access:
   - **Scenario A (L1 Hit):** The data is found in the L1 cache.

$$\text{Probability} = \text{Hit Rate of L1} = \frac{3}{4}$$

$$\text{Time Cost} = \text{Access Time of L1} = 1 \text{ cycle}$$

   - **Scenario B (L1 Miss, L2 Hit):** The data is not in L1 but is found in L2.

$$\text{Probability} = (\text{Miss Rate of L1}) \times (\text{Hit Rate of L2}) = \left(1 - \frac{3}{4}\right) \times \frac{3}{4} = \frac{3}{16}$$

$$\text{Time Cost} = (\text{Access Time of L1}) + (\text{Access Time of L2}) = 1 + 3 = 4 \text{ cycles}$$

   - **Scenario C (L1 Miss, L2 Miss):** The data is found only in main memory.

$$\text{Probability} = (\text{Miss Rate of L1}) \times (\text{Miss Rate of L2}) = \left(1 - \frac{3}{4}\right) \times \left(1 - \frac{3}{4}\right) = \frac{1}{16}$$

$$\text{Time Cost} = (\text{Access Time of L1}) + (\text{Access Time of L2}) + (\text{Memory Access Time}) = 1+3+4 = 8 \text{ cycles}$$

2. Calculate Final EAT by Summing the Weighted Scenarios:
   The final EAT is the sum of each scenario's time cost multiplied by its probability.

$$\text{EAT} = (\text{Prob. A} \times \text{Time A}) + (\text{Prob. B} \times \text{Time B}) + (\text{Prob. C} \times \text{Time C})$$

   Substituting the values:

$$\text{EAT} = \left(\frac{3}{4} \times 1\right) + \left(\frac{3}{16} \times 4\right) + \left(\frac{1}{16} \times 8\right)$$

   Simplify:

$$\text{EAT} = \frac{3}{4} + \frac{12}{16} + \frac{8}{16}$$

   Find a common denominator:

$$\text{EAT} = \frac{12}{16} + \frac{12}{16} + \frac{8}{16} = \frac{32}{16}$$

   The final result is an exact integer, confirming the previous calculation.

$$\text{Effective Access Time} = 2 \text{ clock cycles}$$

Assignment 6: Stack and ELF

1. (5 points) Match each item to the memory region where it is stored at run time.

```c
int x = 100;

int main() {
    int a = 2;
    float b = 2.5;
    static int y;

    int *ptr = (int *) malloc(2 * sizeof(int));
    ptr[0] = 5;
    ptr[1] = 6;

    free(ptr);
    return 1;
}
```

Variable	Memory Region
`x`	Data segment, global with an initial value
`y`	BSS, static with zero initialization
`a`	Stack, automatic local
`ptr`	The pointer variable lives on the stack
`memory referenced by ptr`	Heap, allocated by malloc

2. (1 point)

```c
void f(int a, int b)
{
    int x;
}

int main(void)
{
    f(1, 2);
    printf("hello world");
    return 0;
}
```

When `main` calls `f`, which return address is pushed first?

- ☐ main's return address pushed to lower addresses
- ☒ main's return address pushed to higher addresses
- ☐ f's return address pushed to higher addresses
- ☐ f's return address pushed to lower addresses

- When `main` calls `f`, the program needs to know where to resume execution within `main` after `f` finishes. Therefore, the address of the next instruction in `main` (the one for `printf`) is saved. This is **main's return address**.
- On most common architectures, like x86 and ARM, the stack grows from a higher memory address towards a **lower memory address**. When data is "pushed" onto the stack, the stack pointer is decremented to make space at a lower address.

3. (1 point) What is the core idea behind a classic stack buffer overflow attack?

☒ Write past the buffer to replace return address and redirect execution.

☐ Fill the buffer so the OS expands the stack and grants extra privilege.

☐ Compress the input so more bytes fit in the buffer without detection.

☐ Reorder calls so the program skips checks and runs without limits.

- The Stack Frame
  When a function is called, a **stack frame** is created to store its local variables, arguments, and control information. Crucially, this frame also stores the **return address**—the location the program should jump back to when the function completes.
- The Overflow
  A stack buffer is a local variable (an array) of a fixed size. A buffer overflow occurs when a program writes more data into this buffer than it can hold. Because local variables and the return address are stored next to each other on the stack, this excess data spills over and overwrites adjacent memory.
- The Hijack
  The attacker's goal is to carefully craft the input data so that it overwrites the original return address with a new address—one that points to malicious code (**shellcode**) that the attacker also supplied in the input. When the vulnerable function finishes, instead of returning to the legitimate caller, it "returns" to the malicious code, giving the attacker control of the program.

4. (1 point) Why does adding a NOP sled make a stack buffer-overflow attack more likely to succeed?

☒ Create a landing zone so near-miss returns hit NOPs and slide into the shellcode.

☐ Enlarge the buffer so the payload holds more bytes and bigger shellcode blocks.

☐ Mask unsafe bytes so the input bypasses filters and basic string sanitizers.

☐ Turn off NX/DEP so injected stack code can execute without being blocked.

- The Problem of Precision
  In a buffer overflow attack, the attacker needs to overwrite the return address with the exact starting address of their malicious shellcode. However, finding this exact address can be difficult due to variations in program environments or security measures like **ASLR (Address Space Layout Randomization)**.
- The NOP Sled Solution
  A **NOP (No-Operation)** is a CPU instruction that does nothing except advance the instruction pointer to the next instruction. A NOP sled is a long sequence of these NOP instructions placed in memory directly before the actual shellcode.
- The "Landing Zone"
  By using a NOP sled, the attacker no longer needs to guess the exact starting address of the shellcode. They only need to guess an address that falls *anywhere* within the long NOP sled. If they succeed, the CPU will jump to the sled, execute the harmless NOP instructions one by one (effectively "sliding" down the sled), and eventually arrive at and execute the malicious shellcode. This dramatically increases the probability of a successful attack.

5. (1 point) Which set lists valid defenses against classic stack buffer-overflow attacks?

☒ ASLR, stack canaries, NX/DEP, and bounds-checked string APIs.

☐ Code obfuscation, bigger stacks, faster CPUs, static linking tools.

☐ Disk defrag, RAID arrays, IPv6 routing, DNSSEC protection schemes.

☐ TLS over HTTP, content CDNs, rate limiting, data compression.

- ASLR (Address Space Layout Randomization)
  This defense randomizes the memory locations of the stack, heap, and libraries each time a program runs. This makes it difficult for an attacker to predict the address of their shellcode or other useful code, frustrating their attempt to hijack the return address.

- Stack Canaries
  A "canary" is a secret value placed on the stack between the local variables and the return address. Before a function returns, it checks if the canary value is still intact. If a buffer overflow has occurred, the canary will have been overwritten. The program detects this change and terminates immediately, preventing the corrupted return address from ever being used.
- NX/DEP (Non-Executable Bit / Data Execution Prevention)
  This hardware-level defense marks memory regions like the stack as non-executable. Even if an attacker successfully injects shellcode onto the stack and redirects execution to it, the CPU will refuse to execute instructions from that memory region and will raise an exception, stopping the attack.
- Bounds-Checked String APIs
  This is a preventative programming practice. Many classic overflows are caused by unsafe C library functions like `gets()` or `strcpy()`. Using safer alternatives like `fgets()` or `strncpy()`, which require specifying the buffer size, prevents the overflow from happening in the first place.

6. (1 point) You have two files: `a.txt` and `a.out`. When you double-click them, how does the operating system know that `a.out` is an ELF executable while `a.txt` is not?

   ☒ It reads the file header for the ELF magic number, the first four bytes `0x7F 45 4C 46` ("ELF").

   ☐ It decides from the filename extension and execute bits; if the name looks executable (e.g., `.out`) with x permission, the OS treats it as ELF.

   ☐ It scans the entire file and, if it finds many data, symbols, and relocations, it concludes the file must be ELF; plain text lacks those so it isn't ELF.

   ☐ It checks only for a Unix "shebang" (#!) line; if present the OS treats the file as executable, otherwise it assumes text.

- Beyond File Extensions
  While file extensions like `.txt` or `.exe` are helpful for users, operating systems do not rely on them to identify file types for execution. Extensions can be easily changed and are not a secure or reliable indicator of a file's content.
- Magic Numbers
  Most standard file formats begin with a "magic number"—a unique sequence of bytes that acts as a signature for that format. The OS kernel reads the first few bytes of a file to check for a known magic number.
- The ELF Signature
  The **ELF (Executable and Linkable Format)** standard specifies that every ELF file must begin with a 4-byte magic number: `0x7F` followed by the ASCII characters for 'E', 'L', and 'F'. When a user tries to run a file, the OS loader first checks for this signature. If it's present, it treats the file as an executable; if not, it reports an error (e.g., "cannot execute binary file: Exec format error").

7. (1 point) You have a text file `a.txt` that you want to run directly as a program (without converting it to an ELF binary). Which change will make it executable, and why?

   ☒ Put `#!/bin/bash` as the first line and run `chmod +x a.txt`; the kernel sees the shebang and launches `/bin/bash` with `a.txt`.

   ☐ Put `/* executable */` at the top and run `chmod +x a.txt`; the kernel treats the comment as a run marker.

   ☐ Rename it to `a.sh` and run `chmod +x a.txt`; the kernel uses the file extension to choose an interpreter.

   ☐ Put `#!  ./exec` as the first line and run `chmod +x a.txt`; the kernel accepts a relative shebang path to the interpreter.

- The Shebang (#!)
  On Unix-like systems (Linux, macOS), a special two-character sequence, `#!`, at the very beginning of a file is called a **shebang** or hashbang. It acts as a directive to the operating system's kernel.

- Interpreter Directive
  The shebang tells the kernel, "Do not execute this file directly. Instead, execute the interpreter program specified on this line, and pass this file's path as an argument to it." The path to the interpreter must be absolute (e.g., `/bin/bash`, `/usr/bin/python3`).
- Execute Permission
  In addition to the shebang, the file must have its **execute permission bit** set. The command `chmod +x a.txt` grants this permission. Without it, the system will refuse to run the file, even with a valid shebang.

8. (1 point) Executables and libraries keep getting larger and more complex (many sections, symbols, and relocations). What is the core idea behind ELF that lets tools and the kernel manage this complexity?

   ☒ A standardized data structure/format that describes sections, segments, symbols, and relocations for loaders and tools.

   ☐ A standardized compression scheme that packs binaries so the kernel can store more programs in limited physical memory.

   ☐ A standardized virtual machine layer that interprets instructions instead of the CPU to simplify loading and execution.

   ☐ A standardized scheduler policy that reorders functions at runtime to reduce relocations and improve cache performance.

   - A Common Blueprint
     The core idea of **ELF (Executable and Linkable Format)** is to provide a single, well-defined, and standardized format—a common blueprint—that all development tools (compiler, assembler, linker) and the operating system (loader) can understand.
   - Managing Complexity
     This standard format organizes all the necessary information into a structured container with distinct parts:
     - **ELF Header**: The "table of contents," describing the file's overall layout.
     - **Sections**: Chunks of related data, such as executable code (`.text`), initialized data (`.data`), or symbol information. Sections are primarily used by the *linker*.
     - **Segments**: Instructions on how to map sections into memory for execution. Segments are primarily used by the OS *loader*.
     - **Symbol Tables**: Lists of symbols (function and variable names) and their addresses, used for linking and debugging.

     By having this consistent structure, each tool knows exactly where to find the information it needs, allowing complex programs to be built, linked, and executed reliably.

9. (1 point) Scenario:

   A programmer has written two files: `main.c` and `math.c`. The main function in `main.c` calls the `add()` function, which is defined in `math.c`. These files are compiled separately into two object files: `main.o` and `math.o`. Now, the linker must combine these two .o files into a single executable file.

   When the linker starts its work, it sees a call to the `add()` function within `main.o`, but `main.o` itself does not know where the `add()` function is located. The linker's first job is to find the definition for this function. What is this process of matching the function call (a reference) with the function's definition called?

   ☒ Symbol Resolution — Matching references to symbols with their definitions found across all object files.

   ☐ Memory Allocation — Reserving space in memory for the function's code.

   ☐ Code Generation — Translating C code into machine instructions.

   ☐ Relocation — Updating placeholders in the code with final memory addresses.

- The Linker's First Job
  The linking process has three main stages: symbol resolution, section merging, and relocation. The very first and most fundamental stage is **symbol resolution**.
- What is a Symbol?
  In this context, a **symbol** is a name for a function or a global variable. The file `main.o` contains a *reference* to the symbol `add`, while `math.o` contains the *definition* of the symbol `add`.
- The Matching Process
  The linker's job is to scan all input object files and create a global table of symbols. For every symbol reference it encounters, it must find exactly one matching definition in the table. If a symbol is referenced but never defined, or is defined more than once, the linker will produce an error (e.g., "undefined reference to 'add'" or "multiple definition of 'add'").

10. (1 point) The linker now knows where the `main` and `add` functions are defined... How will the linker typically process these two sections of code?

    The linker now knows where the `main` and `add` functions are defined. To build the final executable, it needs to handle the machine code (the `.text` sections) from both `main.o` and `math.o`. How will the linker typically process these two sections of code?

    ☒ Section Merging: it combines the code section from `main.o` and the code section from `math.o` into one continuous code section.

    ☐ Keep them separate: in the final file, the code from `main.o` and `math.o` are stored in two independent blocks.

    ☐ Choose one: only keep the code section from `main.o` because it contains `main`.

    ☐ Code Compression: compress both code sections to reduce the final file size.

    - Organization by Sections
      Object files produced by the compiler are organized into **sections**. For example, all the executable machine code goes into a `.text` section, initialized global variables go into a `.data` section, and so on.
    - Combining Like with Like
      After symbol resolution, the linker builds the final executable file. It does this through **section merging**. It takes all the sections of the same name and type from all the input object files (e.g., the `.text` section from `main.o` and the `.text` section from `math.o`) and merges them into a single, large section of that type in the output executable. This groups all the code together, all the data together, etc., creating a clean and organized memory layout for the final program.

11. (1 point) All the code now resides in one continuous block, and the linker has determined the final memory address for the `add()` function (e.g., `0x4010a0`). However, the machine instruction in `main` that calls `add` is still using a temporary or placeholder address. What is the final step the linker takes to complete its job?

    ☒ Relocation: it updates the call in `main` by replacing the placeholder with the final address of `add` (e.g., `0x4010a0`).

    ☐ Do nothing: leave the problem for the operating system at runtime.

    ☐ Compute a relative offset to make the executable position-independent.

    ☐ Create a relocation table for the CPU to look up `add` at execution time.

    - The Final Step
      The last major stage of the linking process is **relocation**.
    - The Problem of Addresses
      When the compiler created `main.o`, it had no idea where the `add` function would end up in the final program's memory. So, for the 'call add' instruction, it either used a placeholder address (like 0) or generated code that was marked as needing a fix-up. The compiler also creates a **relocation entry** telling the linker, "This instruction at this offset needs to be patched with the final address of the symbol 'add'."

- Patching the Code
  After merging sections and assigning final virtual addresses to all symbols, the linker performs relocation. It reads the relocation entries and "patches" the code and data. It goes to the location of the 'call' instruction in the merged `.text` section and replaces the placeholder address with the now-known final address of `add` (`0x4010a0`). This ensures that when the program runs, the call instruction jumps to the correct location.