Answer to Questions :

In the name of God

Operating Systems Laboratory

Lab 1

Mohammad Taha Majlesi

Alireza Karimi

Pouria Mahdian

Question 1: What is the architecture of the XV6 operating system? What reasons do you have to support your view?

Answer:

The XV6 operating system is a simplified re-implementation of **Unix Version 6 (V6)**, originally designed by Dennis Ritchie and Ken Thompson. XV6 follows the structure and style of Unix V6 but is implemented for modern architectures like x86 and written in the C programming language. This operating system is designed as an educational tool to learn the fundamental concepts of operating systems. Its features include simplicity, readable code, and the ability to run on multiprocessor systems.

Reasons supporting this operating system:

- **Simplicity:** The operating system has a very simple structure, making it useful for learning the basic principles of operating systems.
- **Inspired by Unix:** It is based on a popular and historically significant operating system, Unix, which has been widely accepted.
- **Ability to run on modern hardware:** Despite its simplicity, the operating system can run on modern processors.

Question 2: What components make up a process in the XV6 operating system? How does this operating system generally allocate the processor to different

processes?

Answer:

A process in XV6 consists of several components, including:

- **User Space:** Contains memory that includes instructions, data, and the stack.
- **Kernel Mode:** This section includes information accessible only to the kernel, such as CPU registers.

The XV6 operating system uses a **time-sharing** scheduling algorithm to divide the processor among different processes. The operating system periodically switches the processor between processes. When a process is not running, the operating system saves the CPU registers of that process, and when it's the process's turn, this information is restored so the process can resume execution.

Question 3: What is the concept of a file descriptor in UNIX-based operating systems? How does the `pipe` function work in the XV6 operating system, and what is it typically used for?

Answer:

A **file descriptor** is a small integer that represents an object managed by the kernel, which can be a file, device, or pipe. Each process in the operating system manages its open files through these file descriptors. For example, file descriptor number 0 is usually assigned to standard input (stdin), number 1 to standard output (stdout), and number 2 to standard error (stderr).

In the XV6 operating system, a **pipe** works by creating a small buffer in the kernel where data is entered from one end (write end) and exits from the other end (read end). This mechanism is used for inter-process communication, especially when a process needs to send its output as the input to another process.

Question 4: What do the `exec` and `fork` system calls do? From a design perspective, what is the advantage of not combining these two?

Answer:

The `fork` and `exec` system calls are two important functions in UNIX-based operating systems, including XV6:

- `fork`: This call creates a new process called the child, which is an exact copy of the parent process. This means all the memory contents of the parent process (including code, data, and stack) are copied to the child process. After executing `fork`, both the parent and child processes start execution in user mode but with different values in their variables. In the parent process, `fork` returns the child process ID, while in the child process, it returns zero.
- `exec`: This call replaces the memory contents of a process (usually the child process) with a new program. `exec` uses an executable file (e.g., a binary file) and places the code of the new program into the process's memory, starting execution from the entry point. After successfully executing `exec`, the process no longer runs the code it was running before and instead runs the new program.

Advantage of not combining these two calls:

By implementing `fork` and `exec` separately, there is a significant design advantage. This separation allows the operating system and programs to:

- **Have more control over child processes:** Using `fork`, the parent process can make various changes to the child process before `exec` is called. For example, it can change open files, redirect standard input and output to files or pipes, and then execute a new program. This provides high flexibility in managing processes.
- **Use pipes and I/O redirection:** In many cases, `fork` allows the parent to set up the child process to read data from specific sources or write to specific destinations before `exec` runs the new program in that process. For example, `fork` can be used to create pipes, and then `exec` can execute a program whose input or output is connected to a pipe.
- **Simplicity in design and testing:** This separation allows programmers to manage and test each part of the process creation and execution separately.

Question 5: Run the command `make qemu -n`. Two disks are given as input to the emulator. What are their contents? (Hint: These disks contain the three main outputs of the build process.)

Answer:

When you run the command `make qemu -n` in the XV6 operating system, this command simulates the XV6 kernel in the QEMU emulator using two disk images. These disks include the following three main outputs:

1. **Kernel Image:** The compiled kernel image that is executed by QEMU when the system starts.
2. **File System Image:** The XV6 file system, which includes user programs and other system files.
3. **Bootloader:** A simple bootloader responsible for loading the XV6 kernel into memory during system startup.

These three main outputs are produced by the build process and are essential for running the operating system in the QEMU emulator.

Question 6: In XV6, what file's content is placed in the first sector of the boot disk file? (Hint: Check the output of the `make -n` command.)

Answer:

In XV6, the first sector of the disk contains the **bootloader**. The bootloader is a simple program responsible for loading the operating system kernel into memory and starting its execution. This bootloader is usually stored in files like `bootasm.S` and `bootmain.c`, and after compilation, it is placed in the boot disk. By checking the output of the `make -n` command, you can see that these files are combined into an image called the bootloader, which is responsible for starting the operating system.

Question 7: Compiled programs are stored as binary files. The boot file is also binary. What type of binary file is this? How does this type of binary file differ from other binary files in the XV6 code? Why is this type of binary file used?

Answer:

The boot file in XV6 is a binary file produced in the machine binary format. This file is usually a **raw binary file without additional information**, which can be executed directly by the hardware (or emulator). The difference between this binary file and other binary files is that the boot file must be loaded directly by the BIOS or emulator and usually includes the necessary code to start the boot

process. In contrast, other binary files (like the kernel and programs) depend on this file for execution.

Why this type of binary file is used:

Because the bootloader needs to be loaded directly by the hardware without requiring the operating system, it must be in a format that the BIOS can read and execute immediately upon startup. A raw binary file meets these requirements by excluding headers and metadata that the BIOS cannot interpret.

Question 8: Why is the `objcopy` command used during the `make` operation?

Answer:

The `objcopy` command is used in the build process to convert object files or executable files into the required formats by the system. This tool transforms files like `bootasm.S` and `bootmain.c`, which are written in assembly or C, into a binary file that can be loaded and executed by the bootloader or operating system. The purpose of this tool is to simplify the process of converting object files into binary files and to remove unnecessary sections (like symbol tables and debug information) from the file.

Question 9: The system boot occurs through the files `bootasm.S` and `bootmain.c`. Why was only C code not used?

Answer:

Both the assembly file (`bootasm.S`) and the C file (`bootmain.c`) are used in the boot process because:

- **Initial Hardware Access:** In the early stages of system bootstrapping, the system is in a minimal state, and direct access to hardware resources like registers and memory addresses is necessary. Assembly code is required for this level of direct hardware manipulation.
- **Transition to Protected Mode:** Switching from real mode to protected mode involves setting up the processor's control registers and descriptors, which is done using assembly instructions.
- **Lack of C Environment:** At the very beginning of the boot process, the runtime environment needed for C code execution (like stack setup and memory management) is not yet established.

After the initial setup is complete, control is passed to C code in `bootmain.c`, which handles more complex tasks like reading the kernel from disk. Using both assembly and C allows leveraging the strengths of each language: assembly for low-level hardware initialization and C for higher-level logic and ease of maintenance.

Question 10: Name one general-purpose register, one segment register, one status register, and one control register in the x86 architecture, and briefly explain the function of each.

Answer:

- **General-Purpose Register:**
  - **EAX (Extended Accumulator Register):** Used for arithmetic and logical operations and to hold function return values.
- **Segment Register:**
  - **CS (Code Segment Register):** Holds the base address of the code segment. It is used to locate the executable instructions of a program.
- **Status Register (Flags Register):**
  - **EFLAGS:** Contains flags that represent the current state of the processor, such as the Zero Flag (ZF), Carry Flag (CF), and Sign Flag (SF), which are set or cleared based on the results of arithmetic and logical operations.
- **Control Register:**
  - **CR0:** Used to control operating modes of the processor, such as enabling or disabling protected mode, paging, and cache settings.

These registers play crucial roles in the operation of the processor, managing data operations, memory segmentation, processor state, and system control settings.

Question 11: The x86 processors have different modes. Upon booting, these processors are placed in real mode, the mode in which the MS-DOS operating system ran. Why? State a main drawback of this mode.

Answer:

Why processors start in real mode:

- **Backward Compatibility:** Real mode provides compatibility with older 16-bit software written for the original 8086 and 8088 processors. Starting in real mode allows the BIOS and bootloader to perform basic hardware initialization using well-established, simple instructions.

Main drawback of real mode:

- **Limited Memory Access:** Real mode can only address up to 1 MB of memory due to its 20-bit addressing limitation.
- **Lack of Protection Mechanisms:** There is no memory protection, multitasking support, or advanced features like virtual memory, which makes the system vulnerable to crashes and limits the capabilities of modern operating systems.

Question 12: Addressing memory in this mode includes two parts: segment and offset, where the first is implicit and the second is explicitly specified. Briefly explain.

Answer:

In real mode, memory addresses are calculated using:

- **Segment (Implicit):** A 16-bit value stored in one of the segment registers (CS, DS, SS, ES). The segment register used depends on the type of operation (e.g., CS for code execution, DS for data access).
- **Offset (Explicit):** A 16-bit value specified in the instruction, representing the distance from the segment base address.

The physical address is calculated as:

```
Physical Address = (Segment × 16) + Offset
```

This method effectively provides a 20-bit address space (since 16 bits shifted left by 4 bits equals 20 bits), allowing access to 1 MB of memory.

Question 13: The `bootmain.c` code reads the kernel starting from the sector after the boot sector and places it at address `0x100000`. Why was this address chosen?

Answer:

The address `0x100000` (1 MB) is chosen because:

- **Avoiding Reserved Memory:** It is above the lower 1 MB of memory, which is reserved for the BIOS, real-mode interrupt vectors, and other system-critical data.
- **Accessible in Protected Mode:** When the processor switches to protected mode, it can access memory beyond 1 MB. Placing the kernel at this address prepares for operation in protected mode.
- **Standard Location:** It is a conventional location for loading the operating system kernel, ensuring compatibility and avoiding conflicts with hardware devices mapped in the lower memory addresses.

Question 14: What is the equivalent code of `entry.S` in the Linux kernel?

Answer:

In the Linux kernel, the equivalent of XV6's `entry.S` is found in the assembly files that handle early system initialization, such as:

- `arch/x86/boot/header.S`
- `arch/x86/boot/pmjump.S`
- `arch/x86/kernel/head.S`

These files are responsible for transitioning the processor from real mode to protected mode or long mode (for 64-bit systems), setting up initial page tables, and jumping to the kernel's C code entry point.

Question 15: Why is this address physical?

Answer:

During the early boot process, before paging is enabled, the system operates in real mode or early protected mode without virtual memory. All memory addresses used are physical addresses because there is no memory management unit (MMU) or page tables active to translate virtual addresses to physical addresses. Therefore, when the bootloader loads the kernel to address `0x100000`, it is using a physical memory address.

Question 16: In this way, at the end of `entry.S`, it becomes possible to execute the kernel C code so that eventually the `main()` function is called. This function

is responsible for preparing the execution of the kernel. Briefly explain each function. Also, find its equivalent in Linux.

Answer:

In XV6, after the assembly code in `entry.S` sets up the processor state and transitions to protected mode, it calls the kernel's `main()` function. The `main()` function performs several initialization tasks:

1. `kinit1()`: Initializes the kernel's physical memory allocator for use during initialization.
2. `kvmalloc()`: Sets up the kernel's page table and enables paging.
3. `mpinit()`: Detects and initializes multiple processors (for multiprocessor support).
4. `seginit()`: Initializes the segment descriptors for kernel and user code/data segments.
5. `consoleinit()`: Initializes the console for input/output operations.
6. `userinit()`: Creates the first user process (often running a shell or init process).

Equivalent in Linux:

- `start_kernel()`: The main entry point for the Linux kernel's C code, which performs similar initialization tasks like setting up memory management, initializing interrupts, starting the scheduler, and creating the initial user-space process.

Question 17: Briefly explain the contents of the kernel's virtual address space.

Answer:

The kernel's virtual address space includes:

- **Kernel Code and Data:** The executable code of the kernel and its global variables.
- **Memory-Mapped I/O Regions:** Addresses mapped to hardware devices for input/output operations.

- **Kernel Stacks:** Individual stacks for each process when it is executing in kernel mode.
- **Page Tables and Memory Management Structures:** Data structures used for virtual memory management.

This space is structured so that it is inaccessible to user-space processes, ensuring system stability and security by preventing unauthorized access to critical kernel data.

Question 18: In addition to paging, the kernel uses segmentation for protection. Why does this segment translation not affect address translation?

Answer:

In protected mode on x86 processors, while segmentation is technically active, modern operating systems like XV6 and Linux set up the segment registers with base addresses of zero and limits that cover the entire address space. This effectively disables segmentation as a means of memory translation. Instead, these systems rely on paging for memory management and protection.

Reason:

- **Simplification:** Using flat segments (base 0, full limit) simplifies memory addressing by making logical addresses equal to linear addresses.
- **Paging Control:** Paging provides finer-grained control over memory protection and address translation, making segmentation unnecessary for these purposes.

Question 19: To store information for managing user-level processes, a structure called `struct proc` is provided. Explain its components and bring its equivalent in the Linux operating system.

Answer:

The `struct proc` in XV6 includes:

- `sz`: Size of the process's memory (number of bytes).
- `pgdir`: Pointer to the page directory (page table) of the process.
- `kstack`: Kernel stack address for the process.

- `state`: Current state of the process (e.g., RUNNING, SLEEPING, ZOMBIE).
- `pid`: Process identifier.
- `parent`: Pointer to the parent process structure.
- `tf`: Trap frame containing saved CPU registers during a system call or interrupt.
- `context`: CPU context used during a context switch.
- `ofile`: Array of pointers to open file structures.
- `cwd`: Current working directory of the process.

Equivalent in Linux:

- `task_struct`: The primary process descriptor in Linux, which contains similar information such as process ID, state, memory management structures, open files, and parent process information.

Question 20: Why is sleeping in the kernel code problematic? (Hint: Pay attention to scheduling in the following.)

Answer:

Sleeping within kernel code can be problematic because:

- **Deadlocks:** If a kernel thread sleeps while holding a lock or resource needed by other threads, it can lead to deadlocks where other processes are blocked indefinitely.
- **Priority Inversion:** Sleeping can interfere with the scheduler's ability to manage process priorities, potentially causing lower-priority processes to delay higher-priority ones.
- **Resource Management:** The kernel must ensure that critical sections are executed atomically and that resources are properly released to maintain system stability.

Therefore, kernel code must be carefully designed to avoid sleeping while holding critical resources and to manage synchronization properly.

Question 21: What is the difference between the kernel address space and the user address space? Why is it set up this way?

Answer:

Differences:

- **User Address Space:**
  - Accessible only when the CPU is in user mode.
  - Contains the user program's code, data, heap, and stack.
  - Lower portion of the virtual address space.
- **Kernel Address Space:**
  - Accessible only when the CPU is in kernel mode.
  - Contains the kernel code, data structures, and memory-mapped I/O.
  - Upper portion of the virtual address space (addresses above `KERNBASE`).

Reason for this setup:

- **Security and Protection:** Separating the address spaces prevents user processes from accessing or modifying kernel memory, which enhances system security and stability.
- **Fault Isolation:** Errors in user-space applications cannot corrupt the kernel, reducing the risk of system crashes.

Question 22: What is the difference between the user address space and the user address space in kernel code?

Answer:

The question seems to have a typo. Assuming it asks about the difference between the user address space and the address space when the process is executing in kernel mode:

Differences:

- **User Address Space:**
  - When a process is running in user mode, it accesses its own user-space memory.
  - It cannot access kernel-space memory addresses.

- **Kernel Address Space during System Calls:**
  - When a process makes a system call or an interrupt occurs, the CPU switches to kernel mode.
  - The process can now access both its user-space memory and the kernel-space memory.
- **Address Translation:**
  - The page tables are set up so that both user-space and kernel-space addresses are valid when in kernel mode.

Reason:

- **Unified Address Space for Kernel Mode:** Allowing the kernel to access user-space memory enables it to read arguments and data passed from user applications.
- **Security Enforcement:** Even though the kernel can access user-space memory, the reverse is not true, maintaining the protection boundary.

Question 23: Which part of the system and which part of the kernel are used in scheduling? (Give one example from each and state the reason for the assignment.)

Answer:

System-Level Scheduling:

- **Component:** The **scheduler** function (`scheduler()`) in XV6.
- **Example:** The scheduler selects the next RUNNABLE process from the process table and performs a context switch to that process.
- **Reason:** Manages CPU time allocation among multiple processes, ensuring fair resource distribution and responsiveness.

Kernel-Level Process Management:

- **Component:** Process state management functions like `sleep()` and `wakeup()`.
- **Example:** When a process performs an I/O operation that cannot complete immediately, it calls `sleep()` to relinquish the CPU, allowing other processes to run.

- **Reason:** Manages the state transitions of processes (e.g., RUNNING to SLEEPING) and synchronizes processes based on events or resource availability.

This separation allows the kernel to efficiently manage processes and resources, improving overall system performance.
