I. Driver installation

1. How to install and start a driver on the system using sc.exe.

i. **Open Command Prompt as Administrator**:
Search for cmd in the Start menu, right-click on "Command Prompt," and select "Run as administrator."

ii. **Install the Driver**:
Use the sc create command to install the driver. Here is the syntax:

- sc create <ServiceName> binPath= <PathToDriverFile>

- Replace <ServiceName> with the name you want to give your driver service.
- Replace <PathToDriverFile> with the full path to your driver file (e.g., C:\Drivers\mydriver.sys).
  - sc create mydriver binPath= C:\Drivers\mydriver.sys

iii. **Start the Driver**:
Use the sc start command to start the driver service:

- sc start <ServiceName>
  Replace <ServiceName> with the name of your driver service.
- sc start mydriver

2. What changes happen in the registry on driver installation and startup?

Ans: When you install and start a driver using sc.exe, several changes occur in the Windows registry.

Creation of a Service Entry:
A new entry is created in the registry under the following path:
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\<ServiceName>

This entry contains several subkeys and values that define the driver's configuration.

1. **Key Registry Values**:

- **ImagePath**: This value specifies the path to the driver's file (e.g., C:\Drivers\mydriver.sys).
- **Type**: This value indicates the type of service. For drivers, this is usually set to 1 for kernel drivers or 2 for file system drivers.
- **Start**: This value determines when the driver is started. Common values include:
  0: Boot start (starts during the system boot).
  1: System start (starts during system initialization).
  2: Auto start (starts automatically during boot but after the boot and system-start drivers).
  3: Manual start (starts when manually triggered).

Other Registry Changes:

- Additional configuration settings, such as security descriptors and dependencies, might also be added under the service's registry key.

II. Driver loading

I. Setup windbg for kernel debugging.
To set up WinDbg for kernel debugging, follow these steps:

i. **Install Debugging Tools for Windows**:
Download and install the Debugging Tools for Windows as part of the Windows SDK or WDK.

ii. **Configure the Target Machine**:
Open an elevated command prompt on the target machine.
Enter the following command to configure network debugging:
- bcdedit /debug on
- bcdedit /dbgsettings net hostip:<host_ip> port:<port> [key:<key>]

- Replace <host_ip> with the IP address of the host machine.
- Replace <port> with any available port (recommended to use port 50000 and up).
- Replace <key> with an optional key for security (e.g., 1.2.3.4).

iii. **Restart the Target Machine**:
After configuring, restart the target machine.

iv. **Configure the Host Machine**:
Launch WinDbg on the host machine.
Select File -> Attach to Kernel (or File -> Kernel Debug in classic WinDbg).
Navigate to the NET tab and enter the network settings used in the target machine.

v. **Establish Connection**:
Click the Break button in WinDbg to establish a connection. If needed, click multiple times.

II. How to break debugger on driver load and try it
-To break the debugger when a driver loads:

1. **Set an Unresolved Breakpoint**:
   - Use the bp command to set a breakpoint on the driver's entry point before the driver loads.

     -bp drivername!DriverEntry

   - This sets a future breakpoint since the driver is not yet loaded.
2. **Load the Driver**:
   - Use sc start drivername to load the driver.
3. **Breakpoint Hit**:
   - When the driver loads, the breakpoint hits, and you can start debugging from the entry point .

III. What does a driver object contain?

- A driver object contains the following key information:

- **DriverEntry**: The entry point for the driver.
- **MajorFunction**: An array of function pointers for the driver's dispatch routines.
- **DriverUnload**: The unload routine for the driver.
- **DriverExtension**: Contains additional driver-specific data .

IV. How does a driver create a new device (when it is a software only driver/when it is a hardware driver)?

- When a driver creates a new device:

- **Software-Only Driver**: Uses IoCreateDevice to create a device object.

  NTSTATUS IoCreateDevice(PDRIVER_OBJECT DriverObject, ULONG DeviceExtensionSize, PUNICODE_STRING DeviceName, DEVICE_TYPE DeviceType, ULONG DeviceCharacteristics, BOOLEAN Exclusive, PDEVICE_OBJECT *DeviceObject);

- **Hardware Driver**: Also uses IoCreateDevice, but additionally interfaces with the hardware.

V. How are the access rights of the device decided and what functions can driver use to create a device?

- Access rights for devices are determined by specifying security descriptors. Functions used include:
- The *AddDevice* routine is responsible for creating functional device objects (FDO) or filter device objects (filter DO) for devices enumerated by the Plug and Play (PnP) manager.
- **IoCreateDeviceSecure**: Creates a device object with a security descriptor.

NTSTATUS IoCreateDeviceSecure(PDRIVER_OBJECT DriverObject ULONG DeviceExtensionSize, PUNICODE_STRING DeviceName, DEVICE_TYPE DeviceType, ULONG DeviceCharacteristics, BOOLEAN Exclusive, PCUNICODE_STRING DefaultSDDLString, LPCGUID DeviceClassGuid, PDEVICE_OBJECT *DeviceObject);

We can also achieve this with INF File

A setup information (INF) file is a text file in a driver package that contains all of the information that device installation components use to install a driver package on a device. Windows uses INF files to install the following components for a device:

- One or more drivers that support the device.
- Device-specific configuration or settings to bring the device online.

VI. What is sddl?
-SDDL stands for Security Descriptor Definition Language. It is used to define security descriptors in a string format.

VII. Where do most drivers create devices?
- Most drivers create devices in the IRP_MJ_CREATE or DriverEntry routine. The device is typically created in the Windows object namespace.

III. <u>Interacting with drivers</u>

1. How to obtain a handle to a device created by a driver?

- To interact with a driver, you need a handle to its device. You can get this handle using the CreateFile function in your program:

HANDLE hDevice = CreateFile(L"\\\\.\\DeviceName", GENERIC_WRITE | GENERIC_READ, 0, nullptr, OPEN_EXISTING, 0, nullptr);

Replace DeviceName with the actual name of the device. This handle allows you to send requests to the device, which are then handled by the driver.

2. What are dispatch routines in the driver object how can they be invoked given you have a handle to a device associated with that driver.

- Dispatch routines are functions in a driver that handle different types of requests (like read, write, or control commands). When you send a request to the device using functions like DeviceIoControl, these routines are called to process the request.

3. What are IRPs. What kernel component creates them and how?

- IRPs are special data structures used by Windows to send I/O requests to drivers. The I/O Manager creates these packets, which include all the necessary information about the request, and then sends them to the driver's dispatch routines.

4. How are parameters passed from user space to driver through IRP?

- Parameters are passed to the driver via IRPs. The IRP contains a 'Currentstacklocation' which includes a Parameters field. This field holds the details of the request, like what operation to perform and any data being sent.

5. What are the 4 major ways in which input and output buffers of DeviceIoControl are passed into the kernel (TransferType). Briefly describe them.

- **Ways to Pass Input and Output Buffers of DeviceIoControl**

  - There are four main ways to pass data buffers with DeviceIoControl:

i. **Buffered I/O**: Data is copied to and from system buffers.
ii. **Direct I/O (**METHOD_IN_DIRECT and METHOD_OUT DIRECT) : Buffers are mapped directly into the kernel's address space, avoiding extra copying.
iii. **Neither Buffered Nor Direct I/O**: User-mode addresses are passed directly, and the driver handles the access.

6. What is the CTL_CODE macro and how to identify the TransferType of the IOCTL code.

- **MACROS:** Macros in Windows kernel programming are preprocessor directives that define patterns to simplify and automate repetitive code tasks, ensuring consistency and readability.

- **CTL_CODE Macro**: The CTL_CODE macro is used to define I/O control codes. It combines several parameters: device type, function, method, and access. It creates a unique code that identifies a specific operation for a driver.
- **Identifying TransferType**: The TransferType of an IOCTL code is determined by the third parameter of the CTL_CODE macro. The possible values are:
- To identify the TransferType of an IOCTL code, look at the third parameter in the CTL_CODE macro:

  0 corresponds to METHOD_BUFFERED
  1 corresponds to METHOD_IN_DIRECT
  2 corresponds to METHOD_OUT_DIRECT
  3 corresponds to METHOD_NEITHER

a. METHOD_BUFFERED

- 0x22E000
- Breakdown:

DeviceType: FILE_DEVICE_UNKNOWN (0x22)
Function: 0x800
Method: METHOD_BUFFERED (0)
Access: FILE_ANY_ACCESS

b. METHOD_IN_DIRECT

- 0x22E004
- Breakdown:

DeviceType: FILE_DEVICE_UNKNOWN (0x22)
Function: 0x801
Method: METHOD_IN_DIRECT (1)
Access: FILE_ANY_ACCESS

c. METHOD_OUT_DIRECT

- 0x22E008
- Breakdown:
  DeviceType: FILE_DEVICE_UNKNOWN (0x22)
  Function: 0x802
  Method: METHOD_OUT_DIRECT (2)
  Access: FILE_ANY_ACCESS

d. METHOD_NEITHER

- 0x22E00C
- Breakdown:
  DeviceType: FILE_DEVICE_UNKNOWN (0x22)
  Function: 0x803
  Method: METHOD_NEITHER (3)
  Access: FILE_ANY_ACCESS

7. What are device nodes, device stacks and driver stacks?

i. **Device Nodes**: These represent individual devices in the device tree managed by the Plug and Play (PnP) manager. Each node corresponds to a specific hardware or software device.
ii. **Device Stacks**: A stack of device objects associated with a single device node, where each device object represents a layer of the driver stack.
iii. **Driver Stacks**: A series of drivers layered together to handle I/O requests for a device. Each driver in the stack can modify or handle the I/O request before passing it to the next driver.

VIII. Using what function does a driver call another driver lower in the driver stack.

- **Function Used**: A driver calls another driver lower in the driver stack using the IoCallDriver function.

IX. What is FsContext and FsContext2 in the fileObject associated with the IRP. Give an example of how they are used in a common driver.

FsContext and FsContext2: FileObject is associated with the open file handle. They are used to store driver-specific data.

Example Use: In file systems, a file can have multiple streams of data. Typically, there's a default stream and possibly named alternate streams.

When a file stream is opened, a file system-specific context structure (like a file control block) is created and linked to the file object. For local file systems, reopening a file stream shares the same context structure between file objects. However, network file systems may create separate contexts if the file is accessed via different paths (like different share names or IP addresses). This distinction ensures that operations on file streams behave consistently across different file system types.

IV. Advanced

1. What are kernel pools and what are they used for. What are allocation tags?

- **Kernel Pools:** Kernel pools are memory pools used by the Windows kernel to manage memory allocation. They come in two primary types:

i. **Paged Pool:** Memory that can be paged out to disk if needed. It is used for allocations that can tolerate delays due to paging.
ii. **Non-Paged Pool:** Memory that remains in physical RAM and is never paged out, making it suitable for allocations that must be immediately accessible.

- **Allocation Tags:** Allocation tags are 4-byte identifiers used to tag memory allocations. These tags help in identifying memory leaks by allowing you to see which component of a driver or system allocated memory that was not freed. Tags are typically composed of ASCII characters.

2. Give a brief description of the kernel pool allocator.

- The kernel pool allocator manages memory allocations within the kernel. Key functions include:

i. **ExAllocatePoolWithTag:** Allocates memory from a specified pool with an associated tag.
ii. **ExFreePool:** Frees allocated memory and determines from which pool the allocation was made.

These functions allow drivers to request memory from the appropriate pool type based on their needs (paged or non-paged)

3. What is paged and non-paged memory?

i. **Paged Memory:** Can be paged out to disk, meaning it can be temporarily moved out of RAM to make room for other processes. Suitable for allocations that do not need to be immediately accessible.
ii. **Non-Paged Memory:** Always stays in RAM and is never paged out. Suitable for critical allocations that need to be accessed quickly without delays.

4. What is IRQL? why can't you sleep at IRQL >= 2? Why can't you access paged memory safely from IRQL >= 2?

- A prioritization scheme used by the Windows kernel to manage hardware interrupts. Higher IRQL values correspond to higher-priority interrupts.

  Why can't you sleep at IRQL >= 2? At IRQL 2 or higher, the system is handling high-priority interrupts or deferred procedure calls (DPCs), and blocking or sleeping operations are not allowed to ensure timely handling of these operations.

  Why can't you access paged memory safely from IRQL >= 2? Accessing paged memory at IRQL >= 2 is unsafe because the memory could have been paged out to disk, and handling page faults at this IRQL could lead to system instability or deadlocks.

5. On x64 what concurrency primitives are present? Describe and explain the difference between semaphores and mutexes.

- x64 architecture provides several concurrency primitives to manage synchronization and ensure proper access to shared resources. The primary concurrency primitives include:

1. **Interlocked Operations**: These are atomic operations provided by the hardware that do not involve any software locks. Examples include InterlockedIncrement, InterlockedDecrement, and InterlockedCompareExchange.
2. **Dispatcher Objects**: These are kernel objects used for synchronization. Examples include events, mutexes, semaphores, and timers.
3. **Executive Resources**: Used for managing synchronization in more complex scenarios, allowing multiple readers or single writers access.
4. **Critical Sections and Spinlocks**: For low-level, high-performance synchronization.

5. **Futures and Promises**: For handling asynchronous operations and communication between threads.

- Both semaphores and mutexes are synchronization primitives, but they serve slightly different purposes and have distinct characteristics.

PARAMETERS	SEMAPHORES	MUTEXES
DEFINITION	A semaphore is a synchronization primitive that controls access to a resource through the use of a counter. It can be used to signal the availability of multiple instances of a resource.	A mutex (mutual exclusion object) ensures that only one thread can access a resource at any one time. It is a locking mechanism used to synchronize access to a resource.
TYPES	Semaphores can be counting semaphores, which have a count indicating the number of available resources, or binary semaphores (similar to mutexes but without ownership).	Mutexes have ownership, meaning the thread that locks the mutex must be the one to unlock it. This prevents other threads from releasing the mutex accidentally.
USAGE	Suitable for managing a resource pool, where a fixed number of resources are available. For example, a semaphore initialized to 3 allows up to three threads to access the resource concurrently.	Suitable for protecting critical sections of code where only one thread should be executing at a time.
EXAMPLE	Imagine a print server with three printers. A semaphore initialized to 3 ensures that only three print jobs can be handled at any one time.	When a thread is updating a shared variable or writing to a log file, a mutex can ensure that only one thread performs the update at a time.

Differences:

Concurrency: Semaphores allow multiple threads to access a finite number of resources, while mutexes allow only one thread to access a resource.

Ownership: Mutexes have thread ownership, meaning the thread that acquires the mutex must release it. Semaphores do not have this concept.

Functionality: Mutexes are simpler and provide mutual exclusion, while semaphores are more flexible, allowing counting of available resources.

6. Explain what is a refcount and how it can be used to prevent use-after-free. (Study how the object FSStreamReg in mskssrv has a refcount at ((DWORD*)FSStreamReg + 6), try to make it 0 without closing the file handle as a BONUS)

- **Refcount:** A technique used to manage the lifetime of objects. Each object has a reference count that tracks how many references to the object exist. When the count drops to zero, the object can be safely deleted, preventing use-after-free errors.

  Example: The FSStreamReg object in mskssrv has a refcount at ((DWORD*)FSStreamReg + 6). Decrementing this count to zero without closing the file handle could lead to resource leaks or crashes.

7. Explain what are reader-writer locks. Suggest how they can be implemented.

- **Reader-Writer Locks:** Synchronization primitives that allow multiple threads to read shared data concurrently while ensuring exclusive access

for writers. They can be implemented using a combination of mutexes and condition variables to manage the read and write counts.

8. When a class is instantiated in C++, how are it's methods maintained in the object?

- When a class is instantiated in C++, its methods are maintained using the virtual table (vtable) mechanism. Each object has a pointer to a vtable that holds pointers to the virtual methods of the class. This allows for dynamic dispatch of methods at runtime.

V. Back to the user

1. What are completion routines, how to register them in an IRP?

- **Completion Routines** are used in Windows drivers to perform actions after an I/O operation has been completed. They are typically registered in an IRP (I/O Request Packet) using the IoSetCompletionRoutine or IoSetCompletionRoutineEx function. These routines are invoked in the reverse order of their registration when IoCompleteRequest is called.
- To set up a completion routine, use IoSetCompletionRoutine or IoSetCompletionRoutineEx. The prototype for IoSetCompletionRoutineEx is as follows:

  NTSTATUS IoSetCompletionRoutineEx(PDEVICE_OBJECT DeviceObject, PIRP Irp, PIO_COMPLETION_ROUTINE CompletionRoutine, PVOID Context, // driver defined BOOLEAN InvokeOnSuccess, BOOLEAN InvokeOnError, BOOLEAN InvokeOnCancel);

i. **DeviceObject**: The target device object.
ii. **Irp**: The IRP for which the completion routine is set.

iii. **CompletionRoutine**: The routine to be called on completion.

iv. **Context**: Optional driver-defined context.

v. **InvokeOnSuccess**: Whether to invoke on successful completion.

vi. **InvokeOnError**: Whether to invoke on error.

vii. **InvokeOnCancel**: Whether to invoke on cancellation.

The completion routine itself must have the following prototype:

NTSTATUS CompletionRoutine(PDEVICE_OBJECT DeviceObject, PIRP Irp, PVOID Context);

Completion routines are executed at IRQL DISPATCH_LEVEL (2) and must follow the rules for this IRQL, such as not accessing paged memory.

2. What are DPCs? Give a few reasons why are they needed.

- **Deferred Procedure Calls (DPCs)** are mechanisms used in the Windows operating system to defer the execution of low-priority tasks to a later time, usually after higher-priority tasks have been completed.

1. **Interrupt Handling**: DPCs allow interrupt service routines (ISRs) to handle critical tasks quickly and defer less critical tasks.
2. **Context Switching**: DPCs help in managing context switching by deferring tasks to be executed in a more appropriate thread context.
3. **Resource Management**: They ensure better resource management by allowing tasks to be scheduled when the system is less busy.

DPCs are used to improve the responsiveness of high-priority tasks by moving non-critical work to a later time, which helps in maintaining the overall efficiency and responsiveness of the system .

VI. Bonus

1. How does a kmdf driver differ from a wdm driver in terms of code?

- KMDF (Kernel-Mode Driver Framework) and WDM (Windows Driver Model) differ in terms of code complexity, structure, and features:

PARAMETERS	KMDF	WDM
Abstraction Level	Provides a higher level of abstraction, encapsulating many common driver functionalities like Plug and Play (PnP) and power management. It reduces boilerplate code and simplifies driver development.	Requires handling lower-level details directly, making the code more complex and prone to errors. Developers must implement PnP and power management routines manually.
Code Structure	Uses a more object-oriented approach with well-defined objects (like device objects, I/O queues, and requests). This structure helps in managing the driver's state and behavior more consistently.	Code is typically more procedural and requires manual management of objects and their states.
Initialization	Driver initialization involves creating framework objects like WDFDEVICE and configuring callbacks using WDF_DRIVER_CONFIG_INIT.	Directly initializes driver and device objects in DriverEntry and AddDevice routines.
Error Handling and Debugging	Provides built-in error handling and tracing facilities which simplify debugging and improve driver reliability.	Error handling is manual and more error-prone.
Plug and Play, Power Management	Automatically manages PnP and power management through framework-provided callbacks.	Requires explicit implementation of PnP and power management routines.

2. How do you reverse kmdf/wdf drivers?

- Reverse engineering KMDF/WDF drivers involves understanding the framework's structures and callbacks. The key steps include:

i. **Identify Driver Entry:**
Locate the DriverEntry function, which initializes the driver and sets Up the framework.

ii. **Analyze Objects and Callbacks:**
Examine the creation and configuration of KMDF objects. Key objects include WDFDEVICE, WDFQUEUE, and WDFREQUEST.

iii. **Understand the Event Callbacks:**
Identify event callback functions like EvtDriverDeviceAdd, EvtIoRead, EvtIoWrite, which handle specific events.

iv. **Use Tools:**
Utilize tools like IDA Pro or Ghidra to disassemble and analyze the driver binaries. Understanding the framework's API is crucial.

3. What are other types of drivers apart from wdm drivers with a brief description of a few.

i. **KMDF (Kernel-Mode Driver Framework):**
Provides a higher-level abstraction over WDM, simplifying driver development for hardware devices by handling common tasks like PnP and power management.

ii. **UMDF (User-Mode Driver Framework):**
Allows writing drivers that run in user mode, suitable for devices that do not require high performance or direct hardware access, such as USB devices. This framework improves system stability as crashes in user-mode drivers do not affect the entire system.

iii. **File System Drivers:**
Implemented using the File System Driver (FSD) model, these drivers manage file systems like NTFS and FAT.

iv. **Network Drivers:**
Based on the Network Driver Interface Specification (NDIS), these drivers handle network protocol operations.

v. **Filter Drivers:**
Modify the behavior of other drivers. Examples include file system filter drivers and network filter drivers.

vi. **Virtual Device Drivers:**
Create virtual devices that emulate hardware functionality, often used for testing and development purposes.