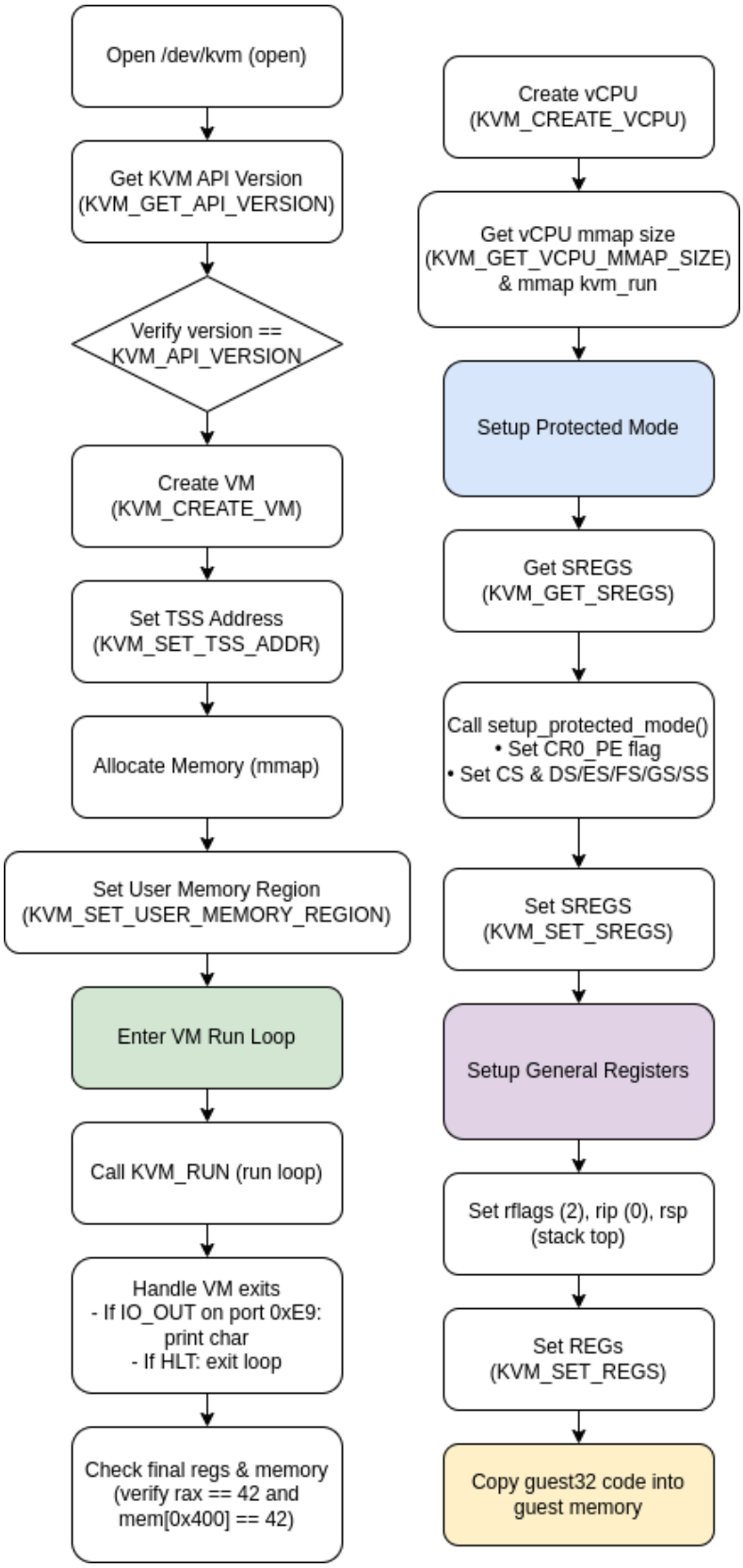
1.)


Open /dev/kvm (open)
Get KVM API Version (KVM_GET_API_VERSION)
Verify version == KVM_API_VERSION
Create VM (KVM_CREATE_VM)
Set TSS Address (KVM_SET_TSS_ADDR)
Allocate Memory (mmap)
Set User Memory Region (KVM_SET_USER_MEMORY_REGION)
Enter VM Run Loop
Call KVM_RUN (run loop)
Handle VM exits
- If IO_OUT on port 0xE9: print char
- If HLT: exit loop
Check final regs & memory (verify rax == 42 and mem[0x400] == 42)
Create vCPU (KVM_CREATE_VCPU)
Get vCPU mmap size (KVM_GET_VCPU_MMAP_SIZE) & mmap kvm_run
Setup Protected Mode
Get SREGS (KVM_GET_SREGS)
Call setup_protected_mode()
• Set CR0_PE flag
• Set CS & DS/ES/FS/GS/SS
Set SREGS (KVM_SET_SREGS)
Setup General Registers
Set rflags (2), rip (0), rsp (stack top)
Set REGs (KVM_SET_REGS)
Copy guest32 code into guest memory

2.)

a.)

These two declarations tell the compiler that there are two external arrays of type unsigned char named guest64 and guest64_end. They are not defined in this file; instead, they are provided (typically via the linker) from another object file or binary blob.

- guest64[]: This array holds the binary code (machine code) for the guest when it is run in 64-bit (long) mode.
- guest64_end[]: This marks the end of the binary code.

Together, these symbols allow the program to determine the size of the guest binary by subtracting the pointer to the beginning (guest64) from the pointer to the end (guest64_end).
In the function that sets up and runs 64-bit mode (run_long_mode), the program uses these symbols to copy the guest code into the guest memory space:
memcpy(vm->mem, guest64, guest64_end - guest64);

b.)

Sets up page table for long mode.
In 64-bit mode, the processor uses a four-level page table hierarchy. In this snippet, only the first three levels are explicitly set up.
PML4 (Page Map Level 4)
The entry pml4[0] is being set with the following attributes:

- PDE64_PRESENT: Marks the entry as present.
- PDE64_RW: Allows read/write access.
- PDE64_USER: Permits access from user-level (non-privileged) code.
- pdpt_addr: The physical address (or offset) of the PDPT (Page Directory Pointer Table) that will be used.

PDPT (Page Directory Pointer Table)
Similarly, pdpt[0] is configured with the same attributes and points to the base address pd_addr for the Page Directory.
Page Directory (PD)
The entry pd[0] is set up to map a large (4MB) page:

- PDE64_PS: This bit indicates that the entry maps a page of a larger size (rather than pointing to a lower-level page table).
- The other flags ensure that the page is present, writable, and accessible to user-mode.

Register Settings to Enable 64-bit Paging:

- CR3:
  sregs->cr3 = pml4_addr;
  The CR3 register is set to the physical address of the PML4 table. This tells the processor where to start looking for page translations.
- CR4:
  sregs->cr4 = CR4_PAE;
  The Physical Address Extension (PAE) flag in CR4 must be set to enable 64-bit paging.
- CR0:
  sregs->cr0 = CR0_PE | CR0_MP | CR0_ET | CR0_NE | CR0_WP | CR0_AM | CR0_PG;

Several control flags in CR0 are set:
CR0_PE: Enables protected mode.
CR0_PG: Enables paging.
The other flags (MP, ET, NE, WP, AM) are required for proper CPU operation in this mode.

- EFER (Extended Feature Enable Register):
  sregs->efer = EFER_LME | EFER_LMA;
  EFER_LME: Enables Long Mode (64-bit mode).
  EFER_LMA: Indicates that the processor is in Long Mode Active.

c.)

- Memory Allocation with mmap:
  mmap Call:
  NULL: Let the kernel choose the starting address.
  mem_size: The total size of the memory region allocated for the virtual machine.
  PROT_READ | PROT_WRITE: The allocated memory is readable and writable.
  MAP_PRIVATE: Changes are not visible to other processes.
  MAP_ANONYMOUS: The mapping is not backed by any file.
  MAP_NORESERVE: Tells the kernel not to reserve swap space for this mapping. This can reduce overhead when large memory areas are allocated.
  -1, 0: Since this is an anonymous mapping, the file descriptor is -1 and the offset is 0.
- Memory Advising with madvise:
  madvise(vm->mem, mem_size, MADV_MERGEABLE):
  This advises the kernel that the pages in the allocated region are mergeable.
  MADV_MERGEABLE: This is a hint for the kernel's Kernel Samepage Merging (KSM) feature. When KSM is enabled, identical memory pages across different processes (or virtual machines) may be merged to save physical memory.

d.)

When the guest code performs an I/O operation, the virtual CPU (vCPU) exits from the KVM run loop with an exit reason of KVM_EXIT_IO (as shown in figure 1 of assignment). This snippet handles that exit. This code intercepts I/O exit events, specifically when the guest writes to port 0xE9, and outputs the data to the host's standard output. It provides a simple mechanism for the guest to perform output.

- Details of the Condition:
  vcpu->kvm_run->io.direction == KVM_EXIT_IO_OUT:
  The code checks if the exit was due to an output (write) operation.
  vcpu->kvm_run->io.port == 0xE9:
  It further verifies that the output operation was performed on port 0xE9. Guest code writes characters to this port to produce output.
- Handling the I/O Operation:
  Pointer Arithmetic:
  The pointer p is set to the start of the kvm_run structure, and the code then adds the offset vcpu->kvm_run->io.data_offset to locate the data buffer.

fwrite:
The function writes the data from the guest's I/O buffer (of size vcpu->kvm_run->io.size) to stdout.
fflush(stdout):
Immediately flushes the output so that the printed characters appear without delay.
continue:
This causes the loop to restart, resuming the VM run loop without further processing of the current exit.

e.)

This line of code copies data from the guest's memory into a local variable memval.

- Source Location:
  The source is &vm->mem[0x400], which means the copying starts from the offset 0x400 (i.e., 1024 bytes into the guest memory).
- Destination:
  The destination is the local variable memval. The code assumes that the data at 0x400 in guest memory is significant (for example, a result value computed by the guest).
- Size of Data (sz):
  The number of bytes copied is determined by the variable sz. This size value should match the expected size of the data (for example, 2, 4, or 8 bytes).

Context in the Program:
Later in the code (in the run_vm function), after the guest execution completes (typically when the guest issues a HLT instruction), the hypervisor checks if the computation by the guest was successful. In addition to verifying the register values (like rax), it also copies and verifies the value stored at guest memory address 0x400.

This call is used to validate that the guest code produced the expected result (e.g., storing the value 42 at address 0x400). It is a mechanism for the hypervisor to communicate with and verify the outcome of guest execution.