VCC Assignment 2

Arijeet Mondal

February 2025

1

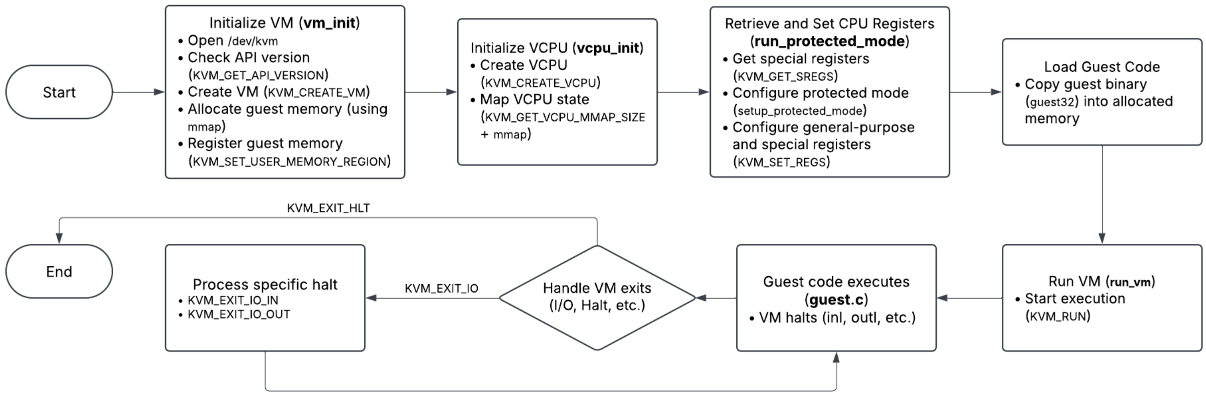

Figure 1: Flowchart for setting up and executing a VM in the sample hypervisor

2

2.a

The line

```
extern const unsigned char guest64[], guest64_end[];
```

declares two external variables that mark the start and end of a 64-bit guest machine code program. These symbols are generated during compilation by converting the guest binary (`guest64.img`) into an object file (`guest64.img.o`). This allows `simple-kvm.c` to access and load the guest program into VM memory using

```
memcpy(vm->mem, guest64, guest64_end - guest64);
```

enabling the VM to execute the guest code.

2.b

```
pml4[0] = PDE64_PRESENT | PDE64_RW | PDE64_USER | pdpt_addr;
pdpt[0] = PDE64_PRESENT | PDE64_RW | PDE64_USER | pd_addr;
```

```
pd[0] = PDE64_PRESENT | PDE64_RW | PDE64_USER | PDE64_PS;

sregs->cr3 = pml4_addr;
sregs->cr4 = CR4_PAE;
sregs->cr0 = CR0_PE | CR0_MP | CR0_ET | CR0_NE | CR0_WP | CR0_AM | CR0_PG;
sregs->efer = EFER_LME | EFER_LMA;
```

The code sets up **long mode paging** in a 64-bit virtual machine. The PML4 (Page Map Level 4), PDPT (Page Directory Pointer Table), and PD (Page Directory) entries are initialized to enable paging. Each entry is marked as:

- Present (`PDE64_PRESENT`)
- Writable (`PDE64_RW`)
- Accessible by user mode (`PDE64_USER`)

The CR3 register is set to `pml4_addr`, which points to the PML4 table. The CR4 register enables Physical Address Extension (PAE) to support 64-bit addressing. The CR0 register enables:

- Protected mode (`CR0_PE`)
- Paging (`CR0_PG`)
- Other necessary CPU features (`CR0_MP`, `CR0_ET`, `CR0_NE`, `CR0_WP`, `CR0_AM`)

Finally, the EFER register is set with `EFER_LME` and `EFER_LMA`, enabling **long mode execution**, allowing the VM to run in 64-bit mode.

2.c

```
vm->mem = mmap(NULL, mem_size, PROT_READ | PROT_WRITE, MAP_PRIVATE |
   MAP_ANONYMOUS | MAP_NORESERVE, -1, 0);
madvise(vm->mem, mem_size, MADV_MERGEABLE);
```

This code allocates memory for the virtual machine using `mmap()`. It reserves `mem_size` bytes with:

- Read and write permissions (`PROT_READ | PROT_WRITE`)
- Anonymous mapping (`MAP_ANONYMOUS`) since it is not backed by a file
- Private mapping and no immediate physical allocation (`MAP_PRIVATE | MAP_NORESERVE`)

After mapping, `madvise()` is called with `MADV_MERGEABLE`, allowing the kernel to deduplicate identical memory pages across processes, improving memory efficiency.

2.d

```
case KVM_EXIT_IO:
if (vcpu->kvm_run->io.direction == KVM_EXIT_IO_OUT &&
   vcpu->kvm_run->io.port == 0xE9) {
char *p = (char *)vcpu->kvm_run;
fwrite(p + vcpu->kvm_run->io.data_offset,
       vcpu->kvm_run->io.size, 1, stdout);
fflush(stdout);
continue;
}
```

This code handles an I/O exit (`KVM_EXIT_IO`) when the guest virtual machine performs an I/O operation. It checks if the operation is an output (`KVM_EXIT_IO_OUT`) and if it is sent to **port** `0xE9`. The data from the guest is extracted using a pointer:

```
p + vcpu->kvm_run->io.data_offset
```

Then, it is written to standard output (`stdout`) using `fwrite()`. Finally, `fflush(stdout)` ensures the output is immediately displayed, and `continue` moves to the next VM execution cycle.

2.e

```
memcpy(&memval, &vm->mem[0x400], sz);
```

This line copies `sz` bytes of data from the virtual machine's memory at offset `0x400` into the variable `memval` using `memcpy()`. The memory at `vm->mem[0x400]` belongs to the guest VM, and this operation allows the host to read a specific value from the guest's memory. This is useful for verifying the state of the VM during execution.