Sample Code

Running Linux in a Virtual Machine

Run a Linux operating system on your Mac using the Virtualization framework.

Download

macOS 11.0+ | Xcode 12.4+

Overview

This sample configures a virtual machine for a Linux-based operating system. You run the sample from the command line, and you specify the locations of the Linux kernel to run and initial RAM disk to load as command-line parameters. The sample configures the boot loader that the virtual machine requires to run the guest operating system, and it configures a console device to handle standard input and output. It then starts the virtual machine and exits when the Linux kernel shuts down.

Configure the Sample Code Project

Before you run the sample program:

1. Download a Linux kernel image.

2. Download an initial RAM disk image to load into memory.

Important

The kernel and RAM disk image must support the CPU architecture of your Mac.

You may obtain a kernel image and the corresponding initial RAM disk image for a given release of the Fedora Linux distribution from `https://download.fedoraproject.org/pub/fedora/linux/releases/<release>/Everything/<architecture>/os/i` mages/pxeboot, where <release> is the Fedora release number and <architecture> is x86_64 for Intel Macs and aarch64 for Apple silicon Macs.

To launch the virtual machine, run the sample's executable from Xcode or in Terminal. You'll need to specify the path to the kernel image and initial RAM disk image as parameters. The parameters are position-dependent, so use the following the order:

```
% LinuxVirtualMachine <pathToKernelImage> <pathToRAMDiskImage>
```

Configure the Boot Parameters for the Virtual Machine

The executable uses a VZVirtualMachineConfiguration object and adds some basic information. The sample configures the virtual machine to use two CPUs and 2 GB of RAM. It also configures a serial port using the custom createConsoleConfiguration function.

```
let configuration = VZVirtualMachineConfiguration()
configuration.cpuCount = 2
configuration.memorySize = 2 * 1024 * 1024 * 1024 // 2 GiB
configuration.serialPorts = [ createConsoleConfiguration() ]
configuration.bootLoader = createBootLoader(kernelURL: kernelURL, initialRamdiskURL:

do {
    try configuration.validate()
} catch {
    print("Failed to validate the virtual machine configuration. \(error)")
    exit(EXIT_FAILURE)
}
```

In addition to the resource allocations, the sample configures a VZLinuxBootLoader object with details about the Linux kernel to run in the virtual machine. The sample's createBootLoader function configures the object using the kernel path and RAM-disk path you specified as command-line parameters. It also sets the object's commandLine property with additional information about how to use the RAM disk and console information. Finally, the sample validates the configuration.

```
func createBootLoader(kernelURL: URL, initialRamdiskURL: URL) -> VZBootLoader {
    let bootLoader = VZLinuxBootLoader(kernelURL: kernelURL)
    bootLoader.initialRamdiskURL = initialRamdiskURL

    let kernelCommandLineArguments = [
        // Use the first virtio console device as system console.
        "console=hvc0",
        // Stop in the initial ramdisk before attempting to transition to the root f
        "rd.break=initqueue"
    ]

    bootLoader.commandLine = kernelCommandLineArguments.joined(separator: " ")

    return bootLoader
}
```

Configure the Serial Port Device for Standard In and Out

During the configuration process, the sample specifies the devices that the virtual machine makes available to the guest operating system. Device types can include network devices, virtual storage devices, sockets, and others.

The sample's `createConsoleConfiguration` function configures a serial port device that the guest operating system uses for standard input and output. The function creates a `VZVirtioConsoleDeviceSerialPortConfiguration` object, which the virtual machine uses to create a Virtio console device for the guest operating system. The `VZFileHandleSerialPortAttachment` object specifies information that the host operating system uses to configure the device. In this case, the attachment contains file handles for the host's standard input and standard output. The sample configures the standard input in raw mode, which passes user input unmodified to the virtual machine.

```
func createConsoleConfiguration() -> VZSerialPortConfiguration {
    let consoleConfiguration = VZVirtioConsoleDeviceSerialPortConfiguration()

    let inputFileHandle = FileHandle.standardInput
    let outputFileHandle = FileHandle.standardOutput

    // Put stdin into raw mode, disabling local echo, input canonicalization,
    // and CR-NL mapping.
    var attributes = termios()
    tcgetattr(inputFileHandle.fileDescriptor, &attributes)
    attributes.c_iflag &= ~tcflag_t(ICRNL)
    attributes.c_lflag &= ~tcflag_t(ICANON | ECHO)
    tcsetattr(inputFileHandle.fileDescriptor, TCSANOW, &attributes)

    let stdioAttachment = VZFileHandleSerialPortAttachment(fileHandleForReading: inp
                                                           fileHandleForWriting: out
    consoleConfiguration.attachment = stdioAttachment

    return consoleConfiguration
}
```

Instantiate and Start the Virtual Machine

After building the configuration data for the virtual machine, the sample uses the `VZVirtualMachine` object to start the execution of the guest operating system.

Before calling the virtual machine's `start(completionHandler:)` method, the sample configures a delegate object to receive messages about the state of the virtual machine. When the Linux operating system shuts down, the virtual machine calls the delegate's `guestDidStop(_:)` method. In response, the delegate method prints a message and exits the sample.

```
let virtualMachine = VZVirtualMachine(configuration: configuration)

let delegate = Delegate()
virtualMachine.delegate = delegate

virtualMachine.start { (result) in
    if case let .failure(error) = result {
        print("Failed to start the virtual machine. \(error)")
        exit(EXIT_FAILURE)
    }
}

RunLoop.main.run(until: Date.distantFuture)
```

The `start(completionHandler:)` method starts the virtual machine asynchronously in the background. The virtual machine loads the initial RAM disk and executes the Linux kernel. After the kernel loads, the user interacts with the Linux system using the sample program's controlling terminal, because the `createConsoleConfiguration` attaches the virtual console to standard input and standard output.

See Also

Virtual machine setup

{} Running macOS in a virtual machine on Apple silicon

Install and run macOS in a virtual machine using the Virtualization framework.

Running GUI Linux in a virtual machine on a Mac

Install and run GUI Linux in a virtual machine using the Virtualization framework.

Installing macOS on a Virtual Machine

Download a macOS restore image and install it in a new VM.

Creating and Running a Linux Virtual Machine

Design and run custom Linux guests on Apple silicon or Intel-based Mac Computers.

Virtualize macOS on a Mac

Configure and run macOS guests on Apple silicon.

Virtualize Linux on a Mac

Configure and run Linux guests on Apple silicon and Intel-based Mac computers.

Running Intel Binaries in Linux VMs with Rosetta

Run x86_64 Linux binaries under ARM Linux on Apple silicon.

Accelerating the performance of Rosetta

Improve Rosetta performance by adding support for the total store ordering (TSO) memory model to your Linux kernel.