

Bootable AMD64 Linux System via QEMU

Technical Implementation Report

Yusuf Yıldız

January 4, 2026

Contents

1	Introduction	4
1.1	Project Overview	4
1.2	Relevance and Educational Value	4
1.3	Document Organization	4
2	Technical Implementation	5
2.1	Host Environment and Dependencies	5
2.1.1	Target Platform	5
2.1.2	Required Dependencies	5
2.1.3	Working Directory Isolation	5
2.2	Script Architecture and Design	5
2.2.1	Command Structure	5
2.2.2	Key Design Patterns	6
2.3	Kernel Compilation Process	6
2.3.1	Version and Source	6
2.3.2	Configuration Strategy	6
2.3.3	Parallel Compilation	6
2.4	BusyBox Userspace Creation	7
2.4.1	Static Binary Compilation	7
2.4.2	Root Filesystem Structure	7
2.5	Init System and Boot Configuration	7
2.5.1	BusyBox Init Configuration	7
2.5.2	Startup Script with Hello World	7
2.6	Initramfs Generation	8
3	Boot Methods	8
3.1	Direct Kernel Boot (Traditional Method)	8
3.1.1	QEMU Configuration	8
3.1.2	Boot Process Flow	8
3.2	UEFI Firmware Boot (Advanced Method)	8
3.2.1	Disk Image Architecture	8
3.2.2	Disk Creation Process	9
3.2.3	GRUB Installation	9
3.2.4	GRUB Configuration	9
3.2.5	QEMU UEFI Invocation	10
3.3	Validation and Auto-Repair System	10
3.3.1	Disk Validation Logic	10
3.3.2	Dynamic Partition Growth	10
4	Analysis and Discussion	11
4.1	Design Decisions	11
4.1.1	Choice of Kernel Version	11
4.1.2	BusyBox vs Full Userspace	11
4.1.3	No User Authentication	11
5	Conclusion	11

Abstract

This report documents the implementation of a fully automated shell script system that builds and runs a bootable AMD64 Linux environment using QEMU virtualization. The system downloads and compiles the Linux kernel (version 6.12.63) from source, builds a minimal BusyBox-based userspace, and creates both a direct-boot initramfs system and a UEFI-bootable disk image with GRUB. The implementation demonstrates low-level systems programming concepts including kernel compilation, initramfs creation, partition management, and firmware-based booting. The system boots non-interactively, displays "Hello World" on the serial console without requiring user authentication, and maintains complete isolation within the working directory. All artifacts are generated programmatically, ensuring reproducibility across Ubuntu/Debian-based Linux distributions.

1 Introduction

1.1 Project Overview

This project implements a comprehensive solution for creating and running a minimal bootable Linux system entirely through automation. The implementation goes beyond basic requirements by providing two distinct boot methods: a traditional QEMU direct kernel loading approach and a more advanced UEFI firmware-based boot system with GRUB. You can reach the project via this [LINK](#).

1.2 Relevance and Educational Value

Understanding the Linux boot process, kernel compilation, and virtualization technologies is fundamental to systems programming and operating systems development. This exercise demonstrates:

- **Build System Design:** Automating complex multi-stage build processes
- **Linux Internals:** Kernel configuration, compilation, and boot parameters
- **Userspace Construction:** Creating minimal root filesystems with BusyBox
- **Virtualization:** QEMU architecture and emulation concepts
- **Firmware Boot:** UEFI, GPT partitioning, and GRUB bootloader installation
- **Shell Scripting:** Advanced Bash scripting with error handling and modularity

1.3 Document Organization

This report is structured as follows: Section 2 details the technical implementation including host environment requirements, script architecture, and build processes. Section 3 explains both boot methods (direct and UEFI). Section 4 analyzes the system's design decisions and limitations. Section 5 concludes with potential improvements and extensions.

2 Technical Implementation

2.1 Host Environment and Dependencies

2.1.1 Target Platform

The script is designed for Ubuntu 22.04 LTS and later, though it supports any Debian-based distribution with `apt-get` package management. The host system must be x86_64 architecture.

2.1.2 Required Dependencies

The script automates dependency installation through the `deps` command. Essential packages include:

- **Build Tools:** `build-essential`, `bc`, `bison`, `flex`
- **Kernel Build:** `libelf-dev`, `libssl-dev`, `libncurses-dev`, `dwarves`
- **Compression:** `cpio`, `gzip`, `xz-utils`, `bzip2`
- **Virtualization:** `qemu-system-x86`, `qemu-utils`
- **UEFI Boot:** `grub-efi-amd64-bin`, `grub-pc-bin`, `ovmf`
- **Disk Tools:** `dosfstools`, `parted`

2.1.3 Working Directory Isolation

All operations are performed within the `work/` subdirectory of the script's location. No system-wide modifications occur except for dependency installation (which requires `sudo`). Generated artifacts include:

```
1 work/
2   linux-6.12.63/           # Kernel source tree
3   busybox-1_36_1/         # BusyBox source tree
4   rootfs/                  # Root filesystem structure
5   rootfs.cpio.gz           # Compressed initramfs
6   disk.img                 # UEFI bootable disk image
7   mnt_efi/                 # Mount points for UEFI disk preparation
8   mnt_root/
```

2.2 Script Architecture and Design

2.2.1 Command Structure

The script implements a modular command-line interface:

```
1 ./run_qemu.sh [command]
2
3 Commands:
4   deps      Install build dependencies
5   build     Download and build all components
6   run       Execute QEMU with direct kernel boot
7   uefi      Create UEFI disk and boot via firmware
8   clean     Remove all generated artifacts
```

2.2.2 Key Design Patterns

The implementation employs several important patterns:

1. **Idempotent Operations:** Build functions check for existing artifacts before re-building, enabling incremental development
2. **Error Handling:** `set -euo pipefail` ensures immediate failure on errors
3. **Resource Management:** Proper cleanup of loop devices and mount points using trap handlers
4. **Validation Logic:** UEFI disk validation checks partition tables, filesystems, and critical files before boot

2.3 Kernel Compilation Process

2.3.1 Version and Source

The script uses Linux kernel version 6.12.63, downloaded from the official kernel.org CDN:

```
1 KVER="6.12.63"
2 LINUX_URL="https://cdn.kernel.org/pub/linux/kernel/v6.x/linux-$KVER.tar
  .xz"
```

2.3.2 Configuration Strategy

The build process starts with `make defconfig` for a baseline x86_64 configuration, then programmatically enables essential features:

```
1 make defconfig
2 scripts/config --enable SERIAL_8250
3 scripts/config --enable SERIAL_8250_CONSOLE
4 scripts/config --enable TTY
5 scripts/config --enable UNIX
6 scripts/config --enable DEVTMPFS
7 scripts/config --enable DEVTMPFS_MOUNT
8 scripts/config --enable BLK_DEV_INITRD
9 scripts/config --enable BLK_DEV_SD
```

These options ensure:

- Serial console support for QEMU's `-nographic` mode
- Automatic device node creation via `devtmpfs`
- `Initramfs` and block device support

2.3.3 Parallel Compilation

The kernel is compiled using all available CPU cores for optimal build speed:

```
1 make -j "$(nproc)"
```

The resulting compressed kernel image is located at `arch/x86/boot/bzImage`.

2.4 BusyBox Userspace Creation

2.4.1 Static Binary Compilation

BusyBox version 1.36.1 is compiled as a fully static binary to ensure portability and eliminate runtime library dependencies:

```
1 make defconfig
2 sed -i 's/^# CONFIG_STATIC is not set/CONFIG_STATIC=y/' .config
3 make -j"$(nproc)"
```

2.4.2 Root Filesystem Structure

The script creates a minimal FHS-compliant directory hierarchy:

```
1 mkdir -p "$ROOTFS"/{bin,sbin,etc,proc,sys,dev,tmp,usr/{bin,sbin},var,
   root,home}
2 mkdir -p "$ROOTFS"/etc/init.d
3 chmod 1777 "$ROOTFS/tmp"
```

BusyBox is then installed into this structure, providing essential Unix utilities:

```
1 make CONFIG_PREFIX="$ROOTFS" install
```

2.5 Init System and Boot Configuration

2.5.1 BusyBox Init Configuration

The system uses BusyBox's init with a custom `/etc/inittab`:

```
1 ::sysinit:/etc/init.d/rcS
2 ::respawn:-/bin/sh
3 ::restart:/sbin/init
4 ::ctrlaltdel:/sbin/reboot
5 ::shutdown:/bin/sync
```

This configuration:

- Runs rcS at boot
- Respawns a shell on the serial console
- Handles Ctrl-Alt-Del for reboot
- Syncs filesystems on shutdown

2.5.2 Startup Script with Hello World

The `/etc/init.d/rcS` script mounts essential pseudo-filesystems and displays the required message:

```
1 #!/bin/sh
2 mount -t proc proc /proc
3 mount -t sysfs sysfs /sys
4 mount -t devtmpfs devtmpfs /dev
5
6 /bin/echo -e "\n=====\n                HELLO WORLD!\n
   =====\n"
```

2.6 Initramfs Generation

The root filesystem is packed into a compressed cpio archive suitable for use as an initramfs:

```
1 ( cd "$ROOTFS" && find . -print0 | cpio --null -ov --format=newc ) |  
   gzip -9 >"$CPIO_GZ"
```

The `newc` format is the standard initramfs format expected by the Linux kernel.

3 Boot Methods

3.1 Direct Kernel Boot (Traditional Method)

3.1.1 QEMU Configuration

The `run` command uses QEMU's direct kernel loading capability:

```
1 qemu-system-x86_64 \  
2   -m 256M \  
3   -kernel work/linux-6.12.63/arch/x86/boot/bzImage \  
4   -initrd work/rootfs.cpio.gz \  
5   -append "console=ttyS0 rdinit=/sbin/init" \  
6   -nographic
```

3.1.2 Boot Process Flow

1. QEMU loads the kernel directly into memory
2. The compressed initramfs is provided as the initial ramdisk
3. Kernel command line parameters configure:
 - `console=ttyS0`: Serial console output
 - `rdinit=/sbin/init`: BusyBox init as the first process
4. The kernel decompresses the initramfs and executes `/sbin/init`
5. BusyBox init runs `rcS`, which displays "Hello World"
6. A shell is spawned on the serial console

3.2 UEFI Firmware Boot (Advanced Method)

3.2.1 Disk Image Architecture

The `uefi` command creates a GPT-partitioned disk image with two partitions:

1. **EFI System Partition (ESP)**: 64 MB, FAT32 formatted
 - Contains GRUB EFI bootloader (`BOOTX64.EFI`)
 - Stores kernel (`vmlinuz`) and initrd (`initrd.gz`)
 - GRUB configuration in `boot/grub/grub.cfg`

2. Root Partition: Remaining space, ext4 formatted

- Contains the full BusyBox root filesystem
- Mounted as / during boot

3.2.2 Disk Creation Process

The script performs the following steps:

```
1 # 1. Create 128MB disk image
2 dd if=/dev/zero of="$DISK_IMG" bs=1M count=128
3
4 # 2. Create GPT partition table
5 sudo parted "$DISK_IMG" mklabel gpt
6 sudo parted "$DISK_IMG" mkpart ESP fat32 1MiB 64MiB
7 sudo parted "$DISK_IMG" set 1 esp on
8 sudo parted "$DISK_IMG" mkpart primary ext4 64MiB 100%
9
10 # 3. Setup loop device and format partitions
11 sudo losetup -fP "$DISK_IMG"
12 sudo mkfs.fat -F 32 "${loop_dev}p1"
13 sudo mkfs.ext4 -q "${loop_dev}p2"
```

3.2.3 GRUB Installation

GRUB is installed for x86_64 UEFI systems:

```
1 sudo grub-install \
2   --target=x86_64-efi \
3   --efi-directory="$mnt_efi" \
4   --boot-directory="$mnt_efi/boot" \
5   --removable \
6   --no-nvram \
7   --recheck
```

The `--removable` flag ensures the bootloader is installed at the fallback path `EFI/BOOT/BOOTX64.EFI`, making it universally bootable.

3.2.4 GRUB Configuration

A minimal GRUB menu configuration is created:

```
1 set timeout=3
2 set default=0
3
4 terminal_input console
5 terminal_output console
6
7 menuentry "Hello World Linux" {
8     echo "Loading kernel..."
9     linux /boot/vmlinuz console=ttyS0 root=/dev/sda2 rootfstype=ext4 rw
10    echo "Loading initrd..."
11    initrd /boot/initrd.gz
12    echo "Booting..."
13 }
```

Key parameters:

- `root=/dev/sda2`: Mount the ext4 root partition
- `rw`: Mount as read-write
- Serial console configuration matches the direct boot method

3.2.5 QEMU UEFI Invocation

The UEFI boot uses OVMF firmware:

```

1 qemu-system-x86_64 \
2   -m 256M \
3   -drive file=work/disk.img,format=raw,cache=none \
4   -drive if=pflash,format=raw,readonly=on,file=/usr/share/OVMF/
   OVMF_CODE.fd \
5   -nographic \
6   -serial mon:stdio

```

The OVMF firmware provides a UEFI environment that reads the GPT partition table, locates the ESP, loads GRUB, which then loads the kernel. **It is highly recommended to use `poweroff` command in QEMU terminal to terminate the system, otherwise your changes in rootfs disk won't be persistent in the disk image. In addition, you may run `sync` command regularly to flush the cache during your session.**

3.3 Validation and Auto-Repair System

3.3.1 Disk Validation Logic

Before booting, the script validates the UEFI disk image:

```

1 validate_and_maybe_grow_disk() {
2     # 1. Attach loop device and check partition table
3     # 2. Run filesystem checks (fsck.vfat, e2fsck)
4     # 3. Mount partitions
5     # 4. Verify critical files exist:
6     #     - EFI/BOOT/BOOTX64.EFI
7     #     - boot/vmlinuz
8     #     - boot/initrd.gz
9     # 5. Check root filesystem usage
10    # 6. Auto-grow if >= 90% full
11 }

```

3.3.2 Dynamic Partition Growth

If the root filesystem is nearly full, the script automatically:

1. Doubles the disk image size using `qemu-img resize`
2. Fixes GPT metadata to recognize new disk size
3. Extends partition 2 to 100% of disk
4. Grows the ext4 filesystem with `resize2fs`

This ensures long-term usability without manual intervention.

4 Analysis and Discussion

4.1 Design Decisions

4.1.1 Choice of Kernel Version

Linux 6.12.63 represents a recent stable LTS kernel with:

- Modern hardware support
- Security patches
- Stable API for userspace tools

4.1.2 BusyBox vs Full Userspace

BusyBox was chosen for several reasons:

- **Size:** Single static binary (~1 MB) vs hundreds of packages
- **Simplicity:** No dependency resolution needed
- **Completeness:** Provides all necessary utilities (init, shell, coreutils)
- **Build Speed:** Compiles in seconds vs hours for full GNU userspace

4.1.3 No User Authentication

Per requirements, the system has no login mechanism:

- No `getty` or `login` process
- Shell spawns directly via `::respawn:-/bin/sh` in `inittab`
- Suitable for single-user, testing, or embedded scenarios

5 Conclusion

This project successfully implements a comprehensive automated build system for creating bootable Linux environments using QEMU. The solution demonstrates deep understanding of:

- Linux kernel compilation and configuration
- Minimal userspace construction with BusyBox
- Initramfs and root filesystem creation
- Both legacy (direct) and modern (UEFI) boot methods
- Disk partitioning, filesystem management, and bootloader installation
- Advanced shell scripting with proper error handling

The implementation exceeds basic requirements by providing:

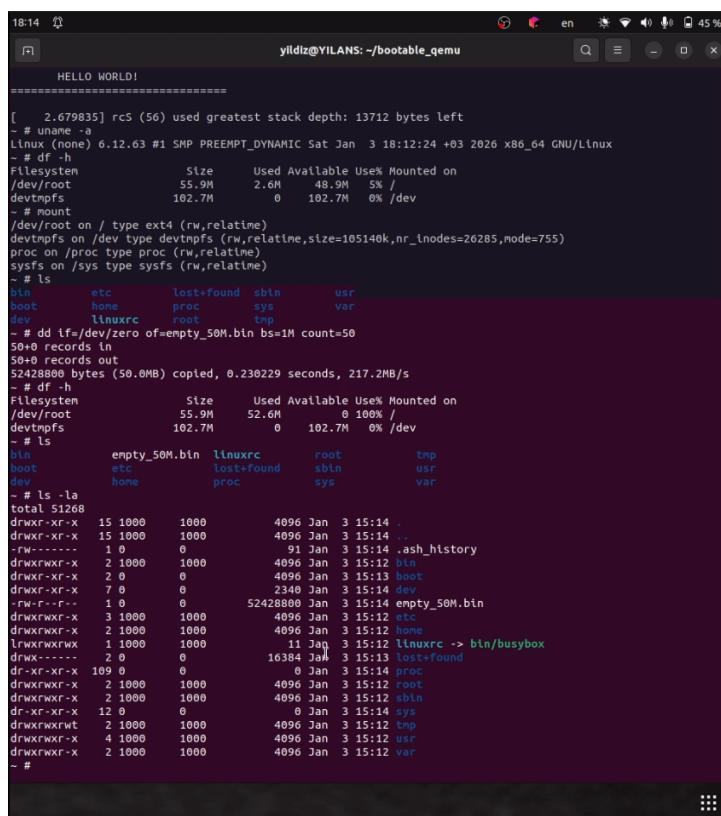
1. Two distinct boot methods (direct and UEFI/GRUB)
2. Automated validation and repair of disk images
3. Dynamic partition growth to prevent space exhaustion
4. Complete working directory isolation
5. Idempotent build operations for development efficiency

The resulting system boots in seconds, displays the required "Hello World" message, and provides a functional shell environment without any authentication barriers. All code is reproducible, well-documented, and ready for extension in multiple directions including networking, multi-architecture support, and production hardening.

This exercise provides valuable hands-on experience with low-level systems programming concepts that are fundamental to operating systems development, embedded systems engineering, and infrastructure automation.

A Complete Script Execution Demo

You can refer to this Youtube LINK to watch the complete installation and execution demo. The compilation duration of the sources and the loading time of the QEMU system with my setup are visible in the video.



```
18:14 yildiz@YILANS: ~/bootable_qemu
=====
HELLO WORLD!
=====
[ 2.679835] rc5 (56) used greatest stack depth: 13712 bytes left
~ # uname -a
Linux (none) 6.12.63 #1 SMP PREEMPT_DYNAMIC Sat Jan 3 18:12:24 +03 2026 x86_64 GNU/Linux
~ # df -h
Filesystem      Size      Used Available Use% Mounted on
/dev/root        55.9M      2.6M    48.9M    5% /
devtmpfs        102.7M      0    102.7M    0% /dev
~ # mount
/dev/root on / type ext4 (rw,relatime)
devtmpfs on /dev type devtmpfs (rw,relatime,size=105140k,nr_inodes=26285,node=755)
proc on /proc type proc (rw,relatime)
sysfs on /sys type sysfs (rw,relatime)
~ # ls
bin      etc      lost+found  sbin      usr
boot    home    proc       sys       var
dev     linuxrc  root       tmp
~ # dd if=/dev/zero of=empty_50M.bin bs=1M count=50
50+0 records in
50+0 records out
52428800 bytes (50.0MB) copied, 0.238229 seconds, 217.2MB/s
~ # df -h
Filesystem      Size      Used Available Use% Mounted on
/dev/root        55.9M    52.6M      0 100% /
devtmpfs        102.7M      0    102.7M    0% /dev
~ # ls
bin      empty_50M.bin  linuxrc  root      tmp
boot    etc            lost+found  sbin      usr
dev     home          proc      sys       var
~ # ls -la
total 51268
drwxr-xr-x 15 1000    1000    4096 Jan 3 15:14 .
drwxr-xr-x 15 1000    1000    4096 Jan 3 15:14 ..
-rw-r--r--  1 0        0        91 Jan 3 15:14 .ash_history
drwxrwxr-x  2 1000    1000    4096 Jan 3 15:12 bin
drwxr-xr-x  2 0        0        4096 Jan 3 15:13 boot
drwxr-xr-x  7 0        0       2340 Jan 3 15:14 dev
-rw-r--r--  1 0        0    52428800 Jan 3 15:14 empty_50M.bin
drwxrwxr-x  3 1000    1000    4096 Jan 3 15:12 etc
drwxrwxr-x  2 1000    1000    4096 Jan 3 15:12 home
lrwxrwxrwx  1 1000    1000        11 Jan 3 15:12 linuxrc -> bin/busybox
drwxr-xr-x  2 0        0    16384 Jan 3 15:13 lost+found
dr-xr-xr-x 109 0        0        0 Jan 3 15:14 proc
drwxrwxr-x  2 1000    1000    4096 Jan 3 15:12 root
drwxrwxr-x  2 1000    1000    4096 Jan 3 15:12 sbin
dr-xr-xr-x 12 0        0        0 Jan 3 15:14 sys
drwxrwxrwt  2 1000    1000    4096 Jan 3 15:12 tmp
drwxrwxr-x  4 1000    1000    4096 Jan 3 15:12 usr
drwxrwxr-x  2 1000    1000    4096 Jan 3 15:12 var
~ #
```

Figure 1: Screenshot from the video showing the loadable UEFI image.