Bahir Dar Institute of Technology

Faculty of Computing: information system

Operating System

Name: Netsanet adane

ID Number: BDU1602271

Submission date -08/05/2018e.c

Submitted to lec.wondmu

Table of content

Introduction

For this project, I chose to install Android-x86 version 9.0-r2 as my operating system.

Android-x86 is an open-source project that aims to port the Android operating system to devices powered by x86 processors, rather than the ARM architecture typically used in mobile devices1. This project was initiated by developers Chih-Wei Huang and Yi Sun in 2009 and has since evolved to support various netbooks, tablets, and ultra-mobile PCs .

The motivation behind this project is to understand the flexibility and adaptability of modern operating systems like Android when deployed in virtual environments such as VMware Workstation. Installing Android-x86 on a virtual machine not only helps simulate a mobile environment on a PC but also provides a platform for testing mobile applications, exploring Android's file systems, and examining the operating system's behavior under different configurations.

This project focuses on downloading, installing, and configuring Android-x86 version 9.0-r2 using VMware Workstation. It also highlights the problems encountered during the process, the solutions applied, and the key technical insights gained from the experience. By conducting this project, we aim to bridge the gap between mobile and desktop operating systems and gain hands-on experience with OS installation and virtualization.

Objectives:

The main objectives of this project are

: 1. To install Android-x86 9.0-r2 on VMware Workstation: Setting up Android-x86 in a virtual environment helps demonstrate the flexibility of mobile operating systems when used outside of their original hardware context.

2. To explore the behavior and performance of Android on x86 architecture: Understanding how Android functions when run on desktop-class hardware provides valuable insights into cross-platform compatibility and resource management.

3. To gain hands-on experience with virtualization tools: This project uses VMware Workstation to create a virtual machine environment. It allows for safe and reversible testing of OS installation and configuration.

4. To identify and troubleshoot issues during installation: Part of the learning process involves recognizing challenges during the setup—such as hardware compatibility problems or boot errors—and applying effective solutions.

5. To analyze Android-x86's filesystem and system-level features: The project includes examining the supported filesystems (like ext4) and understanding why these are suited to Android environments.

6. To reflect on the advantages and limitations of using Android as a desktop OS: Observing both strengths and weaknesses will help evaluate whether Android-x86 can serve as a lightweight desktop alternative.

Requirements

1.Hardware Requirements

To install and run Android-x86 9.0-r2 smoothly in a virtual machine, the following hardware specifications are recommended:

- Processor: x86-based processor (Intel or AMD) with virtualization support (VT-x or AMD-V)
- RAM: Minimum 2 GB (4 GB or more recommended for better performance)
- Hard Disk: At least 10 GB of free space for virtual disk storage
- Graphics: Basic support for VMware SVGA II graphics
- USB/Touchpad/Mouse: Input support for navigation within the Android interface

2.Software Requirements

- Android-x86 Version: ISO Downloaded from the official Android-x86 website
- VMware Workstation (or File: android-x86-9.0-r2.iso VMware Player): Used to create and manage the virtual machine environment where Android-x86 is installed.

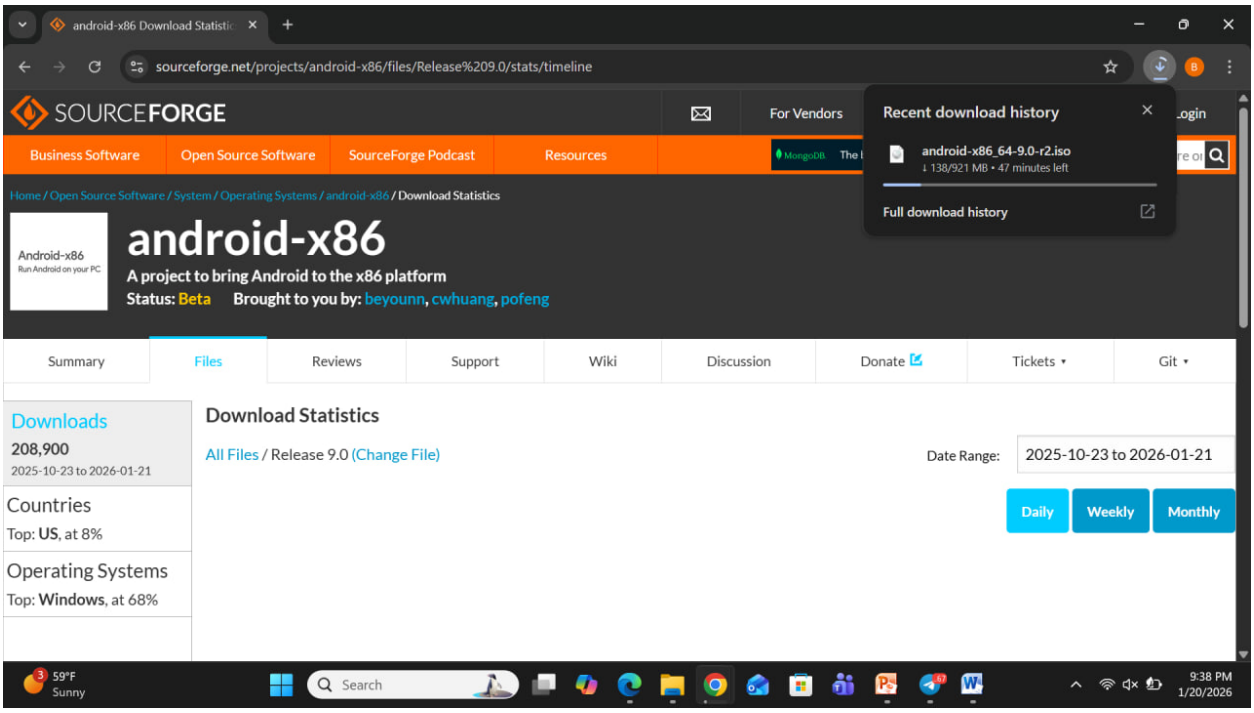
• PC Operating A host OS such as Windows or Linux to run VMware and manage the VM.
System:

Installation Steps

i. Step-by-Step Installation Process

1. Download the Android-x86 ISO File

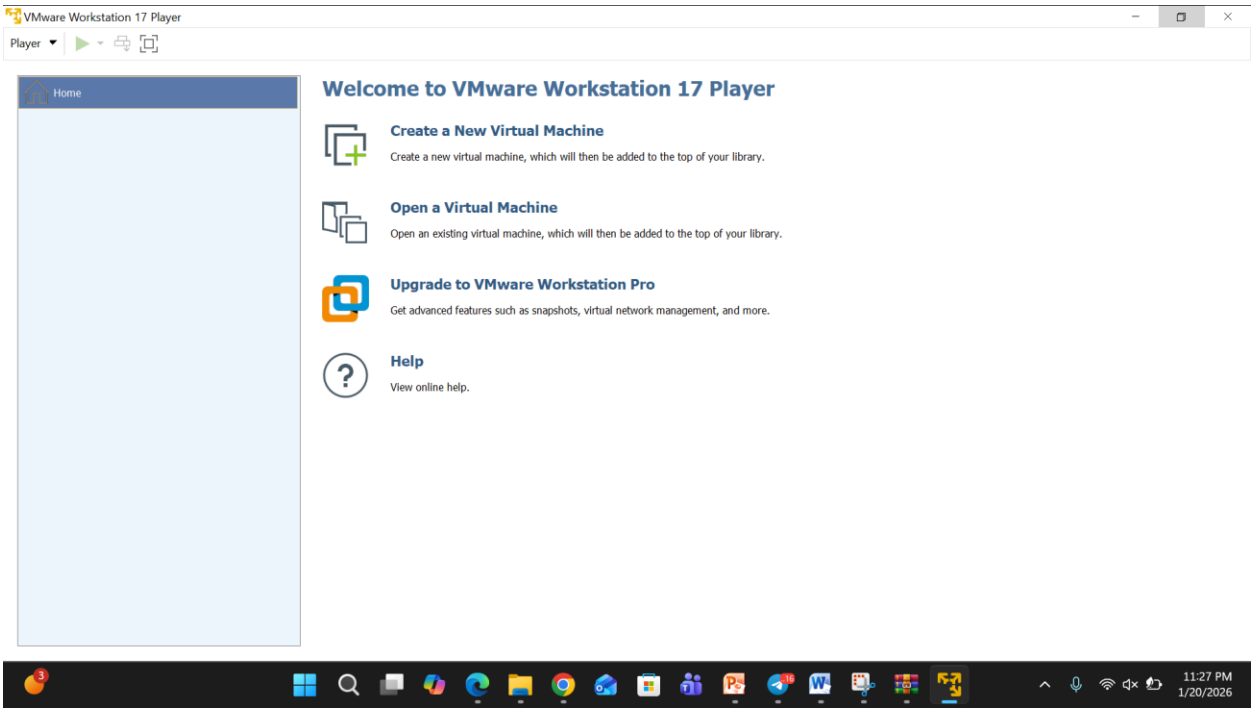
   • Visit the official Android-x86 website and download the android-x86-9.0-r2.iso file.



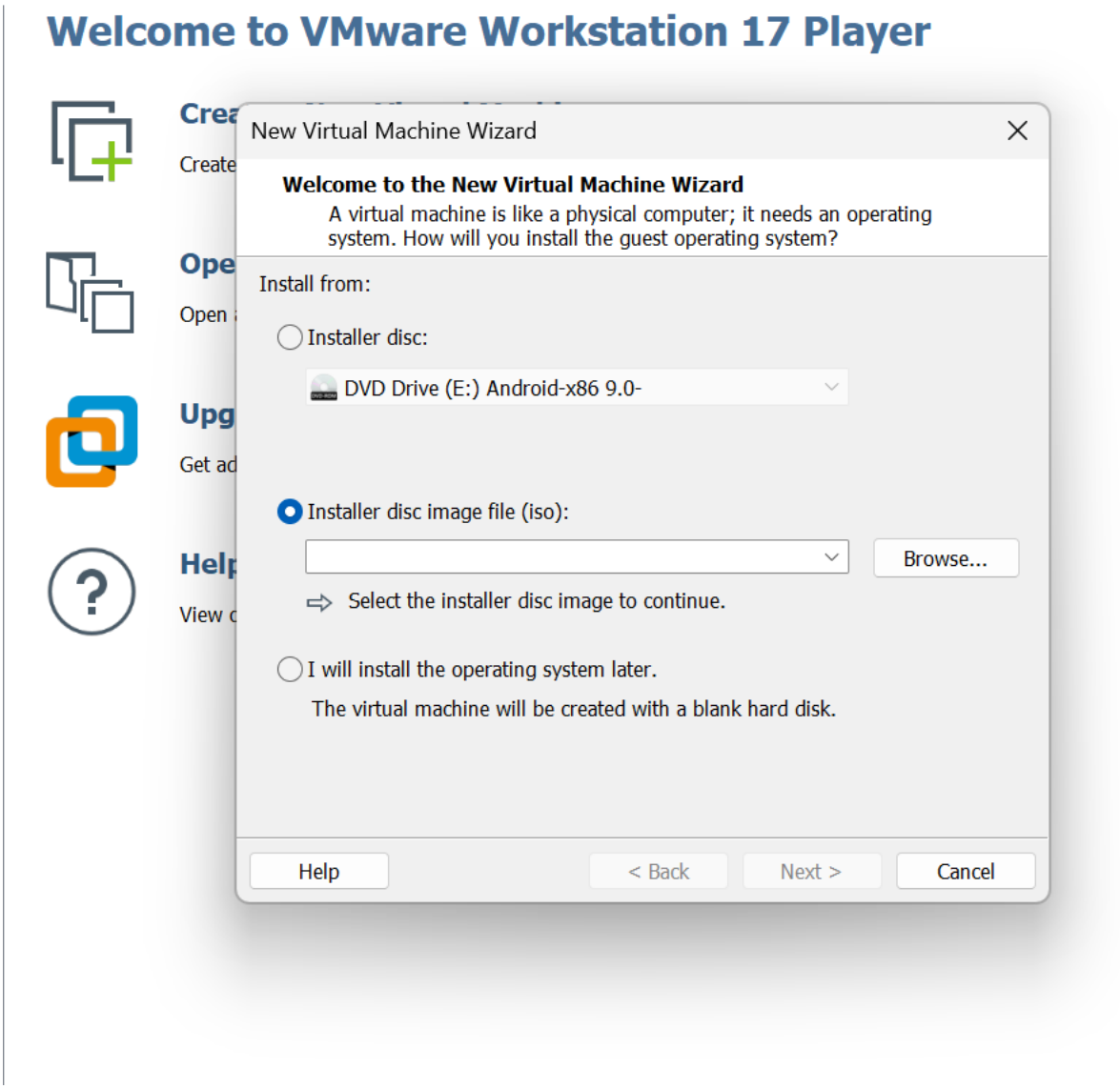
2. Open VMware Workstation

Open the VMware workstation application on pc and it will give us the interface shown in the picture below.

Then we will create a new workstation by clicking “Creating new virtual workstation”. Then it will take us to a page where we can create a new virtual workstation.



3. After we click on the “Create new virtual workstation”, it will take us to the page where the creation of the workstation process starts as shown in the picture below.

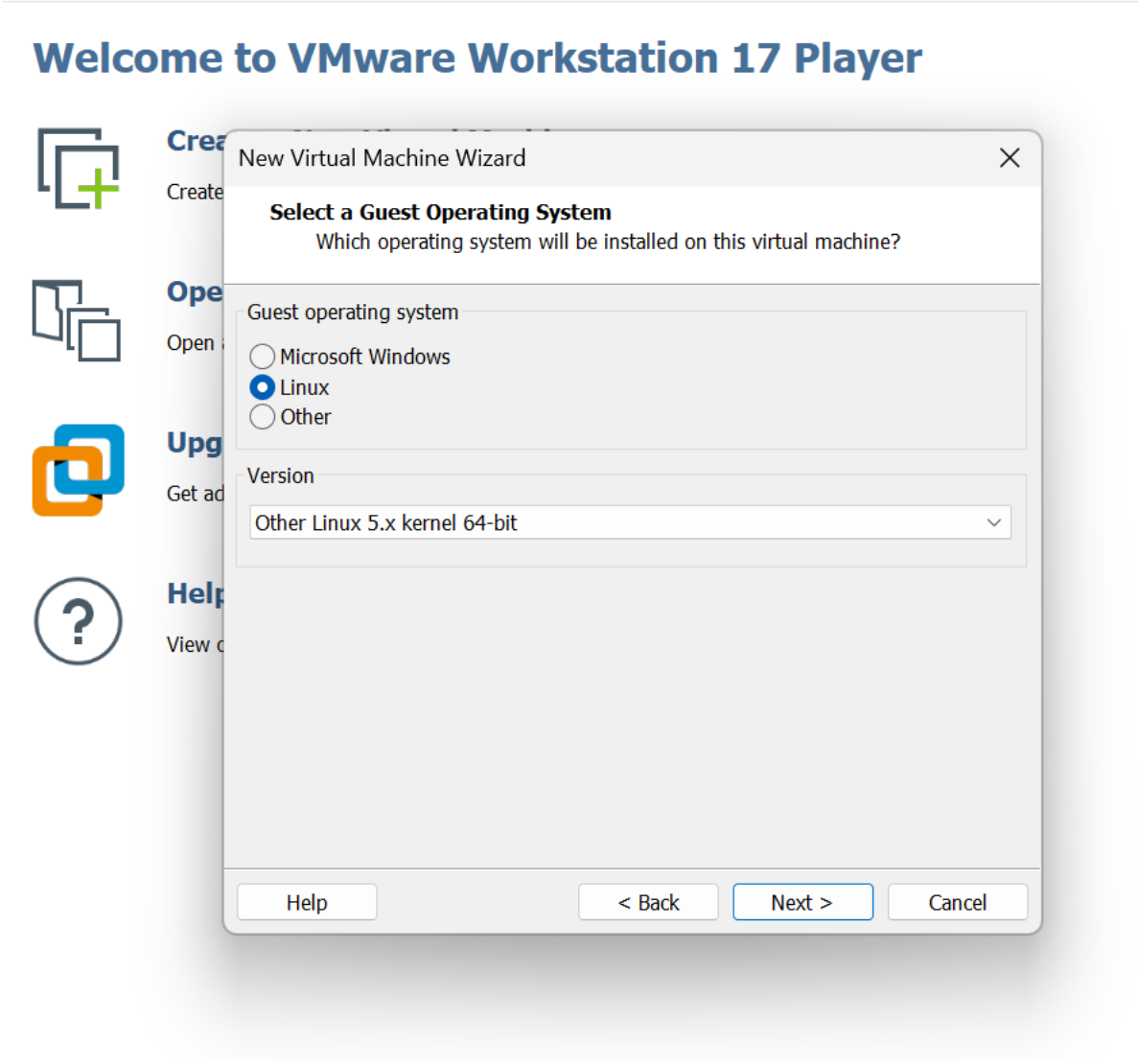


This page gives us two options, and we will use the second option which says, “I will install the operation system later”. We can also choose the first one, which will not make that much of a difference on our work.

4. Set Guest Operating System

• Select “Linux” as the guest OS and choose “Other Linux 5.x or later kernel 64 bit”.

• then Click Next.



On this step, the reason that we choose Linux as a guest OS is that Android is based on the Linux kernel, so technically, it is a Linux operating system under the hood. Although it doesn't behave exactly like a traditional Linux distro (e.g., Ubuntu or Fedora), the kernel and system-level architecture are Linux-based.

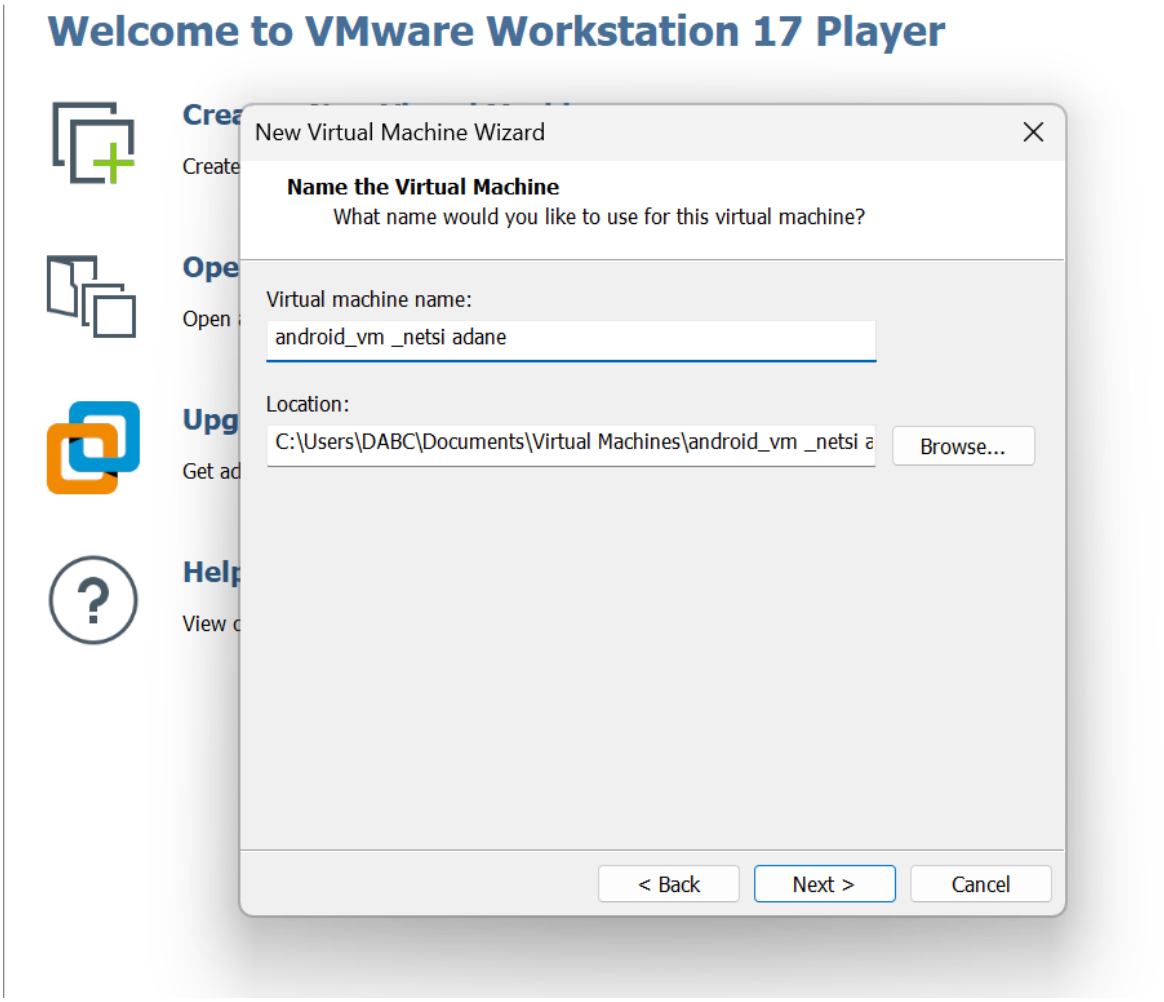
VMware doesn't have a specific preset for "Android," so the closest and most compatible choice is to set it as Linux.

In addition to that, the reason we choose the version of Linux as Linux 5.x kernel is that Android-x86 9.0-r2 uses a Linux kernel version around 4.19 or 4.20, depending on the build.

5. Name and Location

• Name the VM (e.g., Android-x86_YourName) and choose a location to store VM files.

• Click Next.

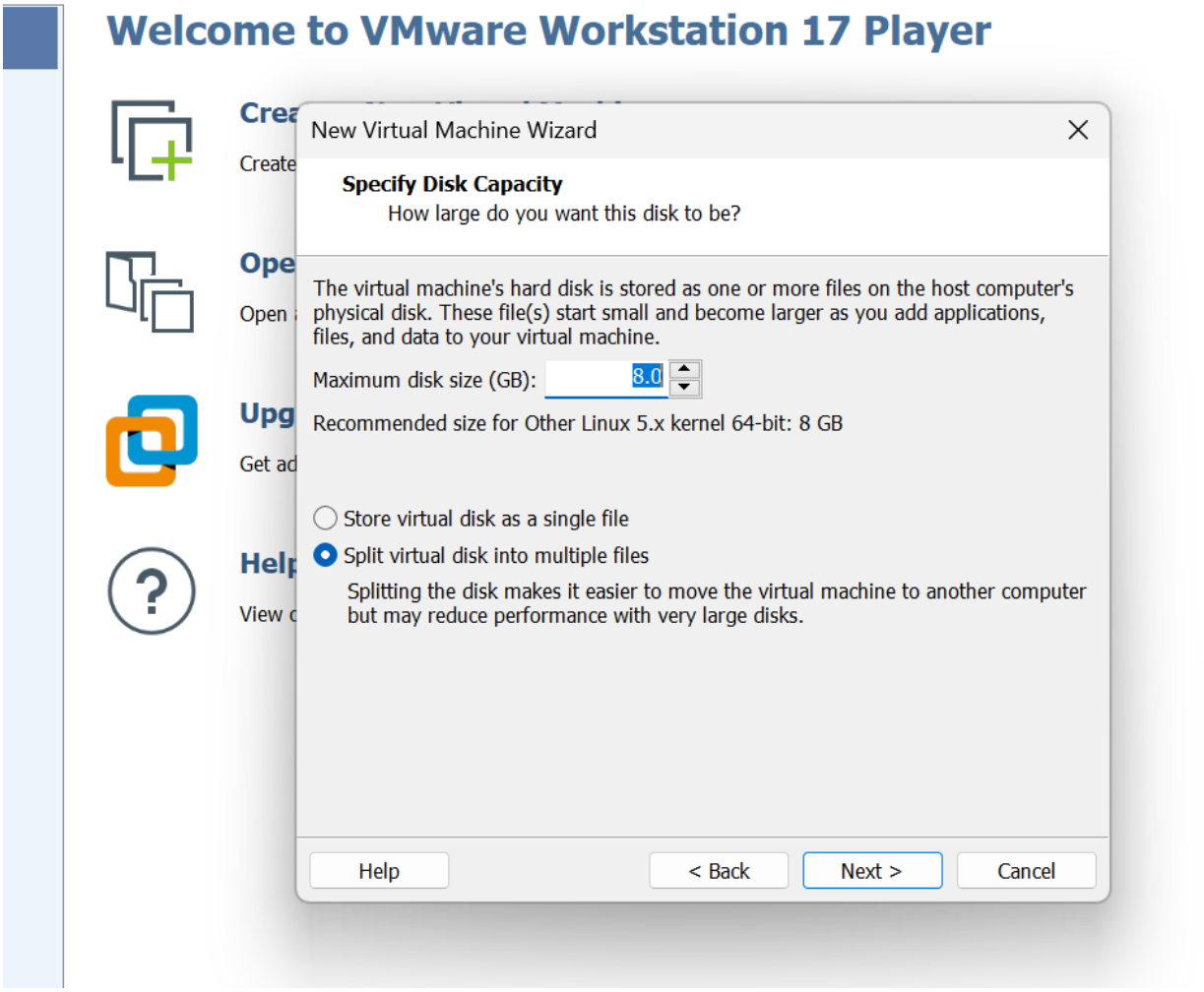


6. Specify Disk Capacity

Allocate at least 8–10 GB for disk size.

Choose “Store virtual disk as a multiple file.

” Click Next.



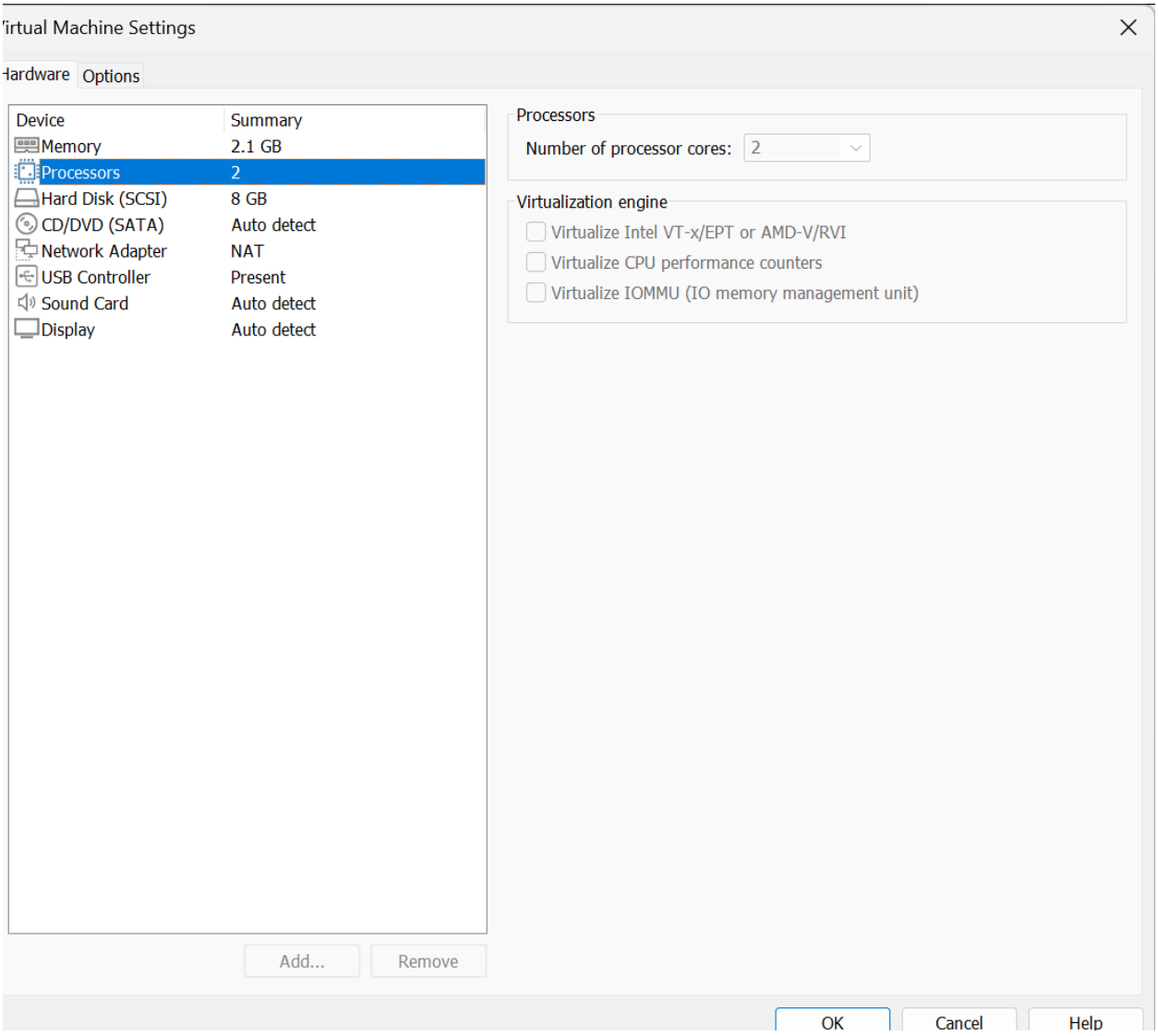
7. Customize Hardware

Click “Customize Hardware.”

Set Memory to at least 2 GB.

Set Processors to 2 cores if possible.

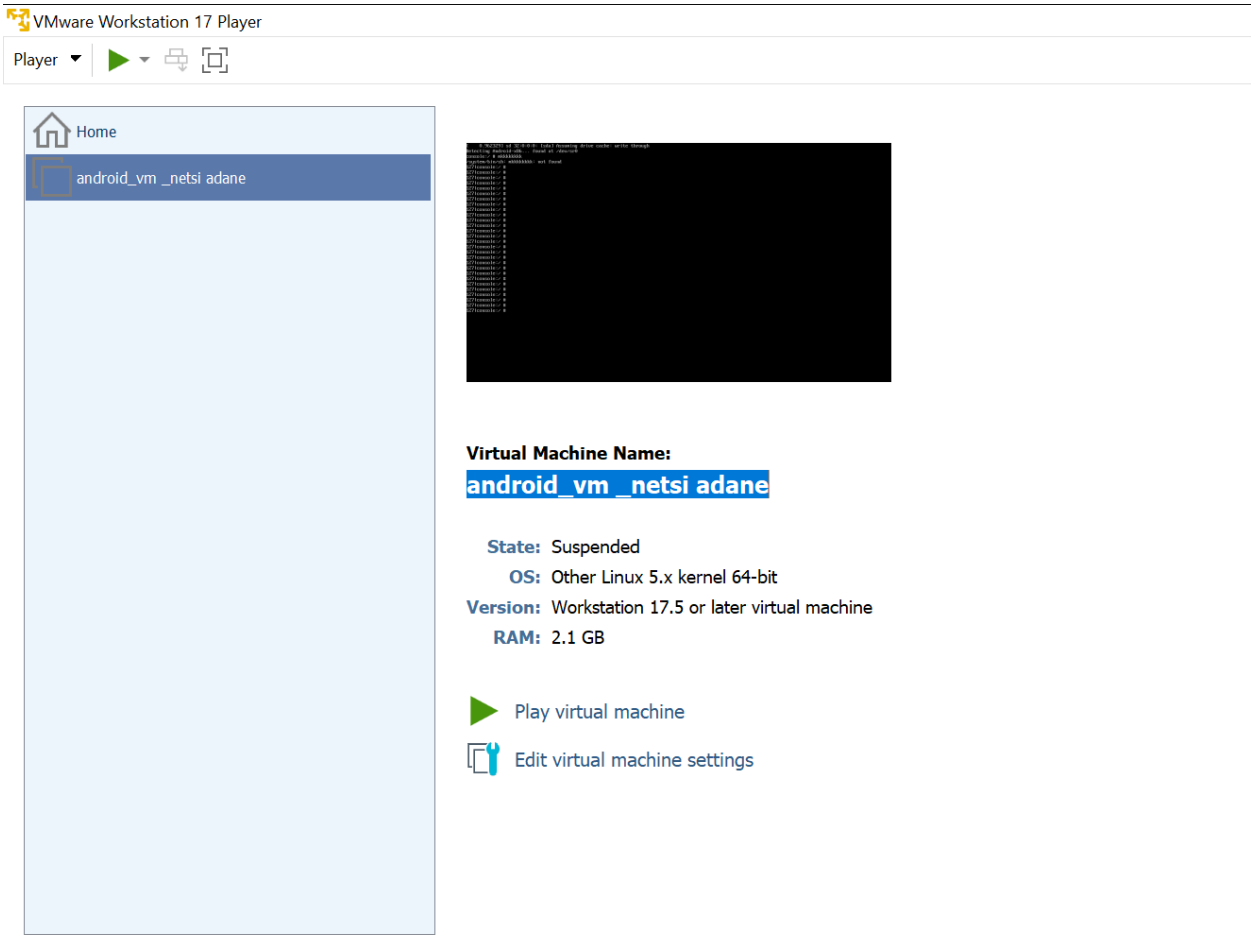
Click close



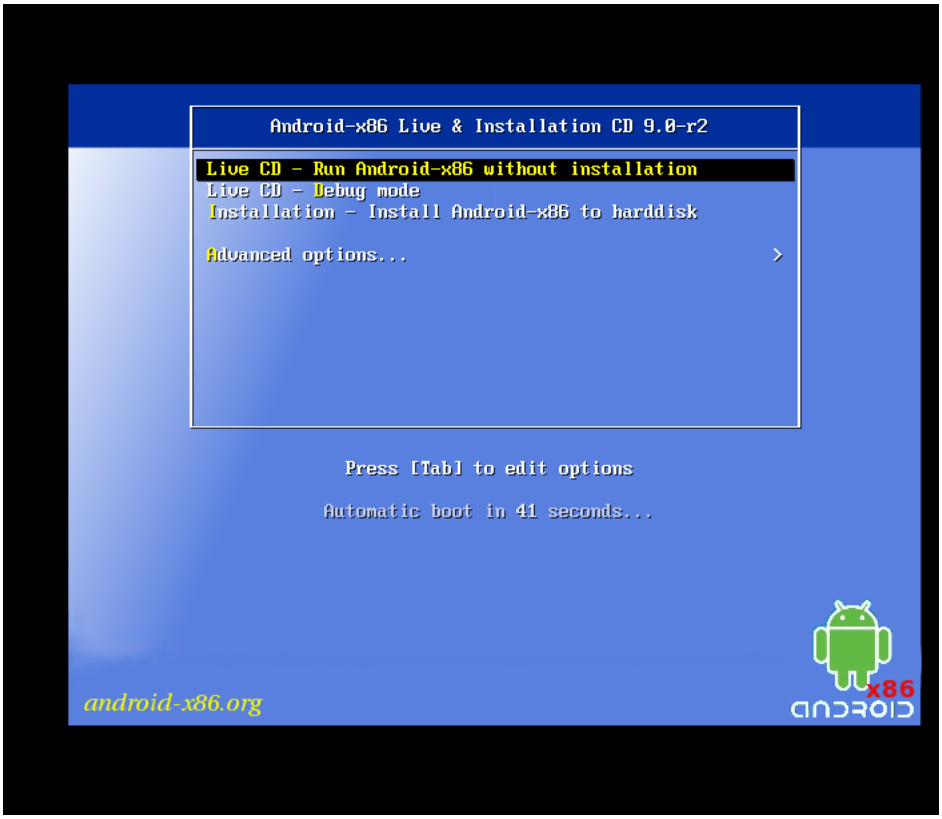
This page gives us two options, and we will use the second option which says, “I will install the operation system later”. We can also choose the first one, which will not make that much of a difference on our work. 4. Set Guest Operating System • Select “Linux”

8. Boot and Install Android-x86

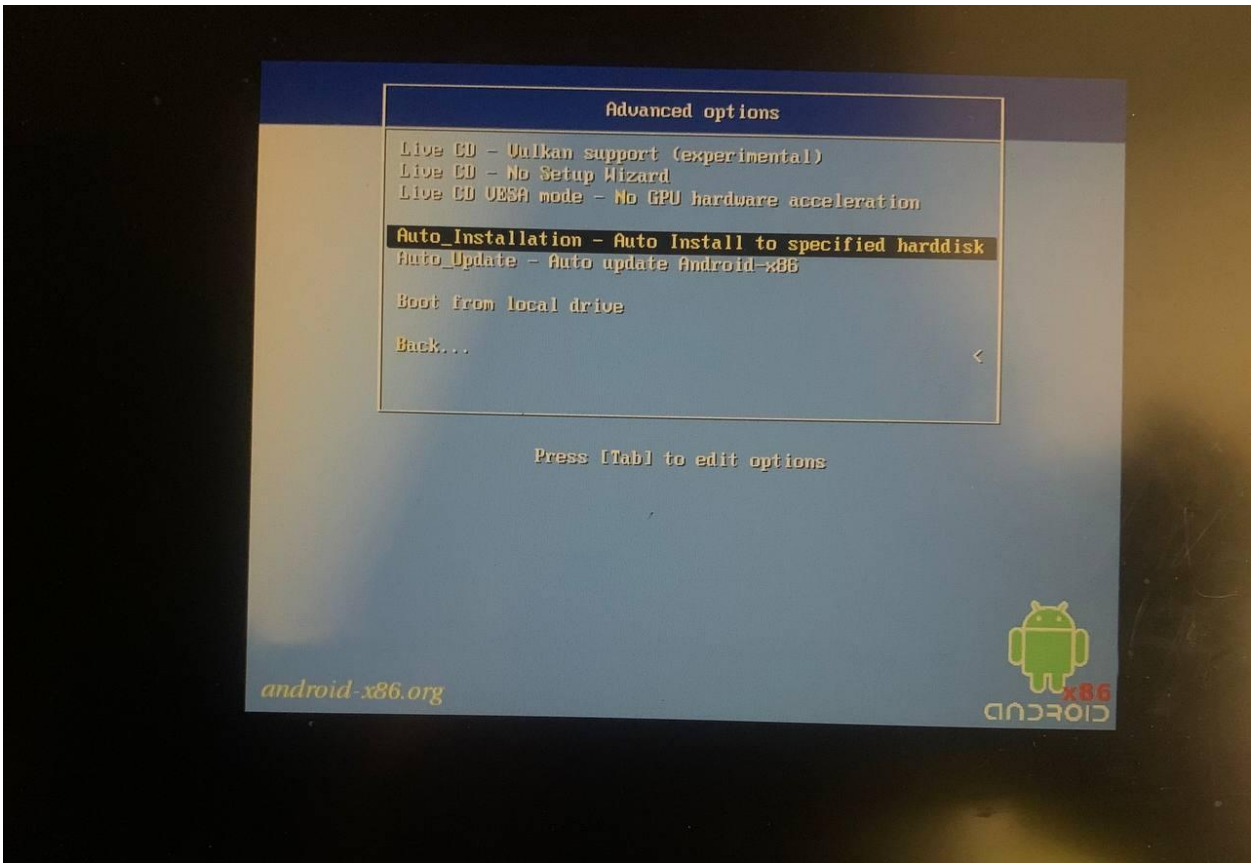
Power on the VM by clicking on the sign that looks like a triangle pointing to the right. Or we can just right click on the newly created workstation and choose power on, aster that our workstation will be powered on.



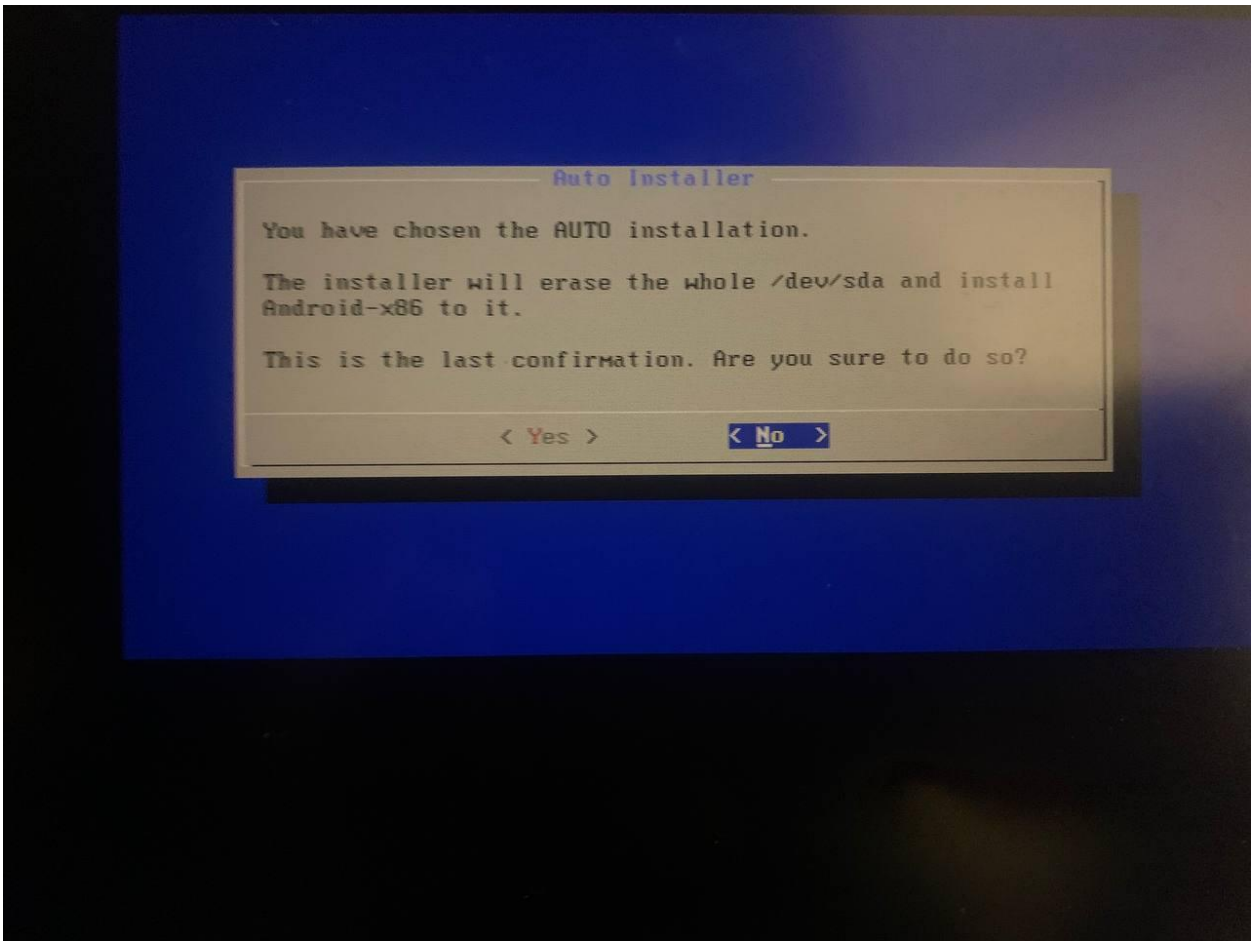
9 . From the boot menu, we choose “Advanced options”.



10. After that we choose “Auto installation...”.



11. On this step, it will tell us that the installer will erase the whole dev/sda and ask us if we are sure to do that and we will choose yes



12. After the installer erases dev/sda, it will install Android on it. Then it asks us if we want to run android or reboot it. We choose to reboot it.

• This option restarts the virtual machine.

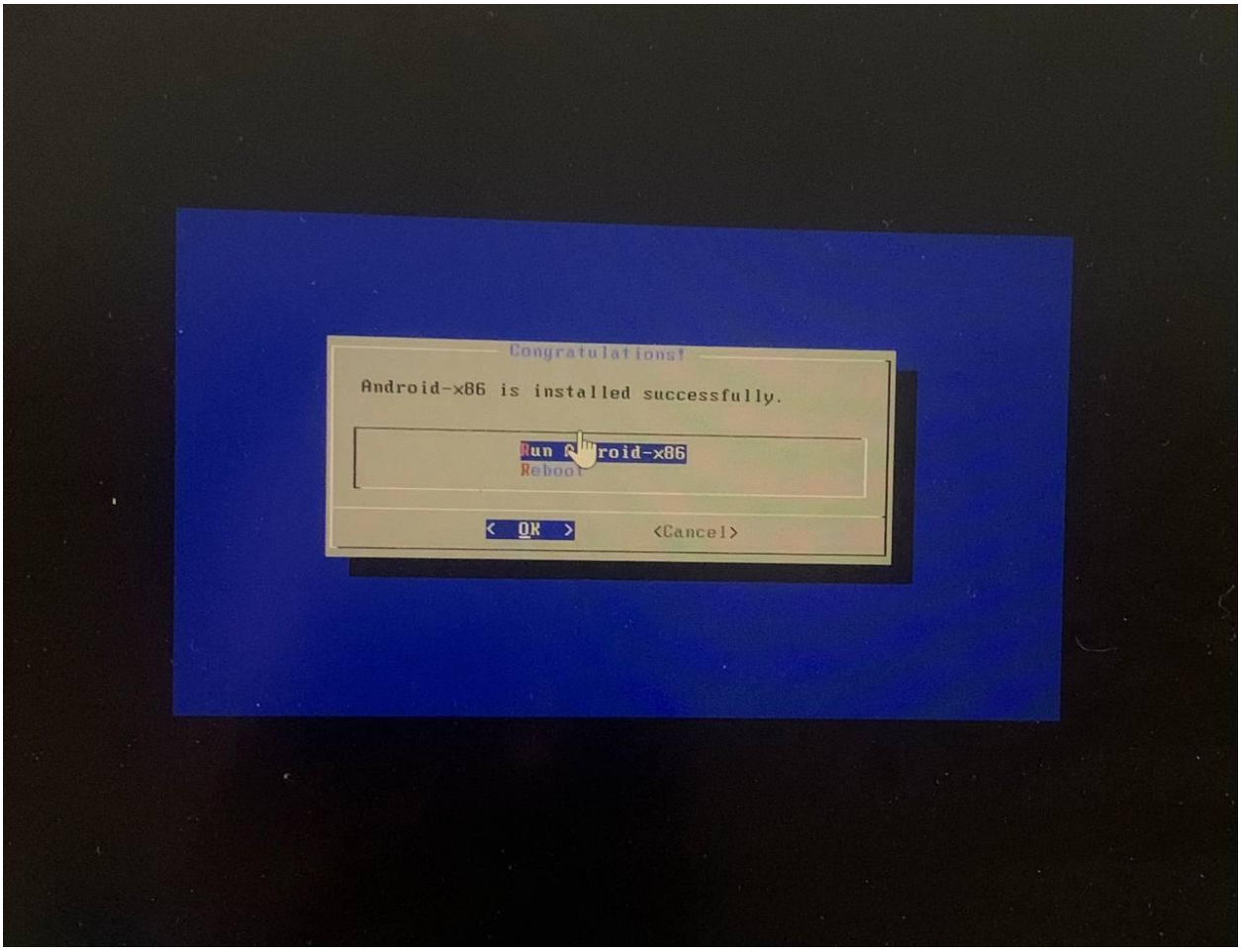
• After reboot, the system loads Android from the virtual hard disk (e.g., /dev/sda), not from the installer ISO.

• This is how you permanently start using Android

But if we were to choose the first option, boots directly into the Android system you just installed, without restarting the virtual machine.

• It lets you test the installation immediately and explore Android.

• But: This runs in a temporary live session, meaning any changes you make (like downloads or settings) might not persist if you reboot later.



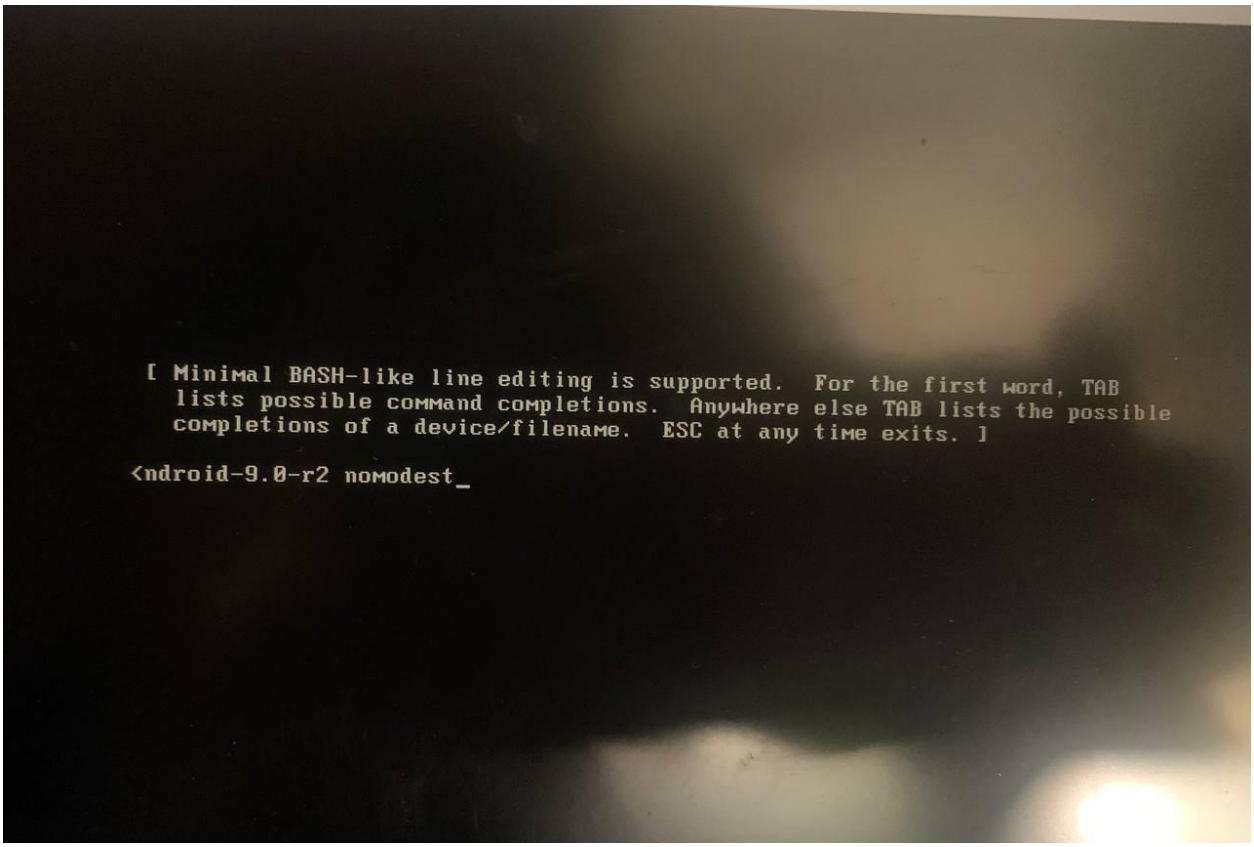
13. After rebooting it, it will restart the workstation and will directly take us to a grub menu which has around 4 options. We choose the first one which is the standard option

. It’s the default and is meant for regular use. It:

- Loads Android normally
- Uses default hardware drivers (graphics, input, etc.)
- Boots into the graphical user interface (GUI)

It’s what you want when everything is working as expected.

The other options are troubleshooting (debug) modes, used when the normal boot has problems:



Debug mode

• Boots Android in text/command-line mode with logs showing system activity.

• Helpful for advanced users to see where boot is failing.

• No graphical interface.

Debug nomodeset

• Prevents Android from loading video drivers (disables hardware graphics acceleration).

• Useful if you're stuck on a black screen or your display is glitching.

Debug – VESA mode

• Forces Android to use VESA-compatible graphics mode (a basic, universal driver).

• Useful if your GPU isn't supported and normal boot doesn't work.

14. The selected option will take us to another sub grub menu which has two options and we choose the first one by pressing the letter 'e' which stands for editing.

15. Then it will take us to a page where we can write command and by clicking space we write the command "nomodeset"which is a Linux kernel boot parameter that we can type in at the boot command line (that black screen with a place to enter text before the OS loads).

Normally, during boot, the Linux kernel hands over control of the display (graphics) to your GPU's driver—this is called Kernel Mode Setting (KMS).

But sometimes:

• The GPU driver crashes

• The screen goes black or glitches

• The system hangs

"nomodeset" tells the kernel: "Don't try to use the graphics driver. Just keep using basic VGA graphics instead.

" Android-x86 inherits its kernel behavior from Linux. If we're stuck on:

• A black screen after GRUB

• A frozen boot logo

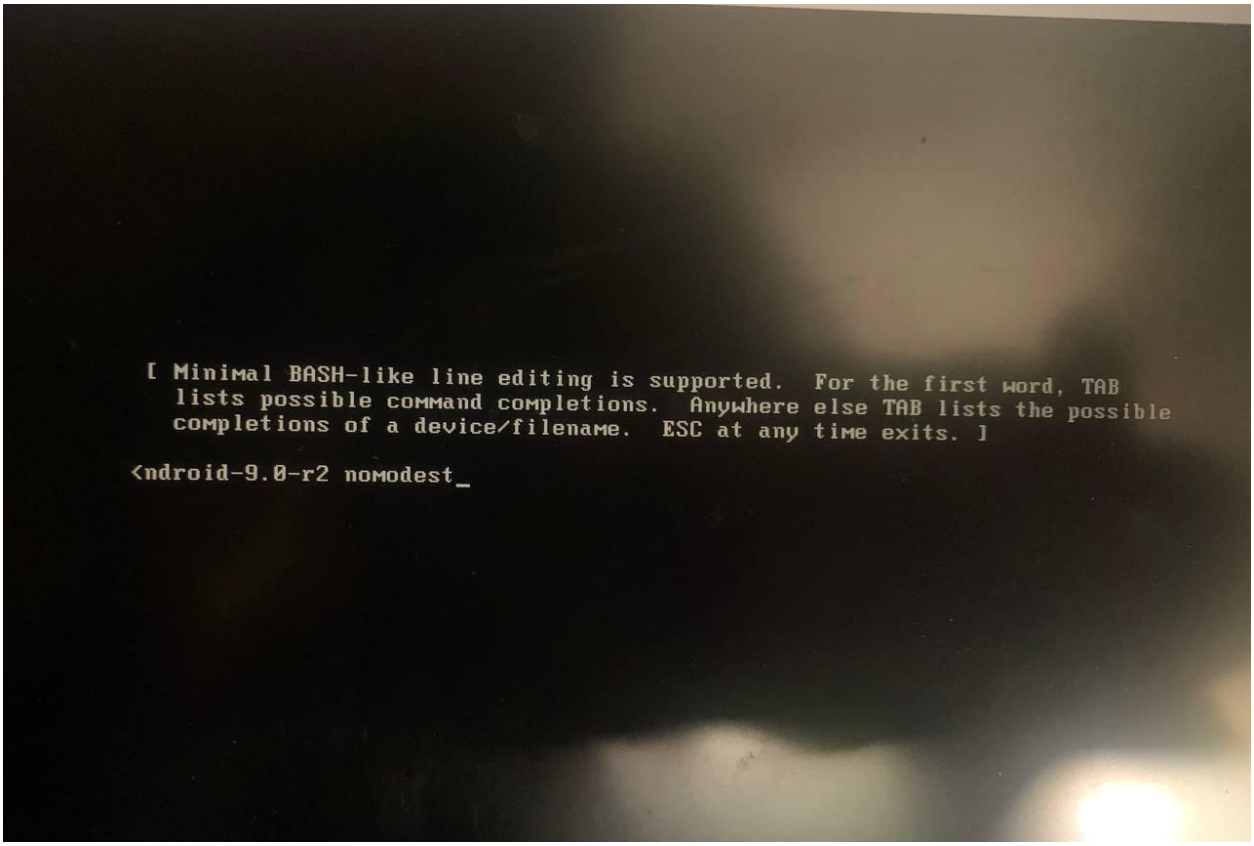
• Random graphical glitches

...we can press e in GRUB to edit the boot command line and add “nomodeset” at the end of the line that starts with Linux. This disables problematic GPU drivers and boots Android with basic video settings, which helps:

• Older machines

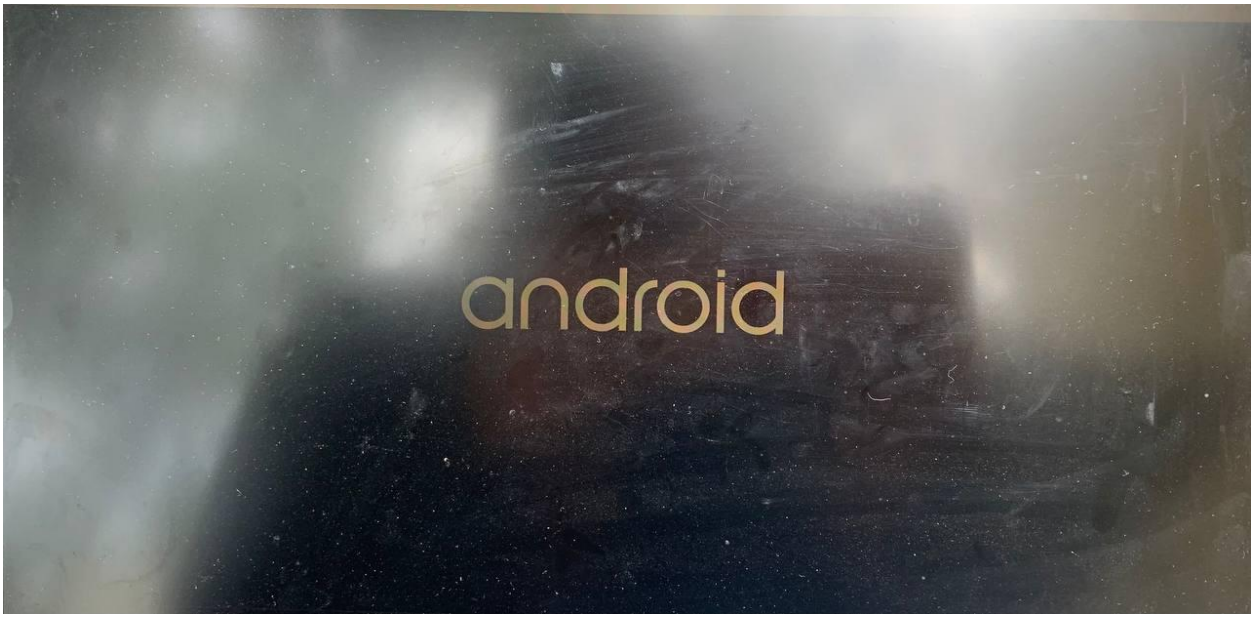
• Some virtual machines

• PCs with unusual or unsupported graphics cards



16. Then we press Esc or enter then it will take us back to the sub grub menu and while we are on the first option we press ‘b’ which stands for boot. By pressing b, we're telling GRUB: “I’m done editing—boot now using this custom setup.”

17. Finally, it will open our operating system, and we are done installing it, which the next step will be creating an account, password and the like.



18. The next step is to create a user account. (I do on a video for this)

Unlike Windows, Android doesn't force a login screen because:

• It uses screen locks (PIN, pattern, fingerprint, face unlock) to secure your data.

• Once unlocked, it assumes you're the right person.

Android Is Designed Primarily for Single Users

• Most Android devices (phones, tablets, etc.) are meant to be used by one person at a time.

• Unlike a shared desktop in a home or office, phones are personal devices with private messages, photos, etc.

• So, multi-user support wasn't a core part of Android's design originally. So that, unlike other OS like Windows, when we resart or power on the Android OS, it doesn't make us choose between multiple

users, unlike it will chane to guest mode or another user or even to the owner, after the OS has started its work. So when we say creating user account, it is like creating a new Gmail account and use it as a user account on the OS. The creation process is described below.

i. Creating this account will let us use other google apps like play Store and the like. On this step, we click on create account and it will take us to the next step.

On the next step, it asks us to provide our first name and our last name, but the surname is optional, and we can stop just after writing our first given name and then click next.

Next it asks us to choose between the emails it will create us by default, or we can create our own. We click on create your own mail address, then it gives us a box where we can put our new mail address, which we created on our own. 17

After we created our own email address, we will set our password. This should be a mix of special characters, numbers, uppercase letters, lowercase letters in order to make it strong. After we complete this, the last draft looks like this which ensures us the creation of our email/user account.

Our user account looks like this. This contains our newly created email address, and our name.

Problems faced

While attempting to install Android x86 on a virtual machine for a system programming assignment, I faced a series of frustrating and time-consuming problems.

First, when creating a new virtual machine, I was unsure about the correct type and version to select. The default settings pointed to “Oracle Linux 64-bit,” which led to uncertainty about compatibility. After

continuing with the setup, I found that the ISO image was not automatically attached, and the virtual machine prompted me to manually install the 19 operating system. Even after attaching the ISO file, the live session failed to boot successfully.

In VirtualBox, I encountered a persistent issue where the system either got stuck in the GRUB menu or kept cycling back to it without loading Android. Pressing keys like e or Tab to edit boot options didn't respond as expected. This made it difficult to apply recommended tweaks like modifying boot parameters to enable a graphical interface or bypass certain boot checks.

I experimented with various settings: increasing base memory to 4 GB, assigning 2 processor cores, disabling/enabling EFI, adjusting video memory, and toggling 3D acceleration. None of these resolved the problem. I also verified my system's virtualization support (VT-x/AMD V and nested paging), which was enabled, yet Android still wouldn't boot properly. Even the live session — which usually loads without installation — failed to work, suggesting deeper compatibility or configuration issues.

I eventually switched to VMware, hoping for better results. While I managed to complete the installation process there, rebooting the virtual machine brought me right back to the GRUB menu instead of launching Android. Attempts to edit the GRUB configuration from within Android were also problematic. The system didn't recognize standard utilities like "adb" or "gcc", making development almost impossible from the default terminal.

To work around this, I installed Termux inside Android to get access to a usable terminal and compiler tools. However, even in Termux, I faced issues such as file permission errors, missing compiler binaries, and compatibility warnings during code compilation. Simple actions like writing or reopening code files required additional steps due to Android's unconventional file system and permission model.

All these obstacles — from bootloader limitations to missing development tools — made the process unexpectedly complex and time-intensive, highlighting several systemic flaws in running Android x86 in virtualized environments.

Solution for the faced problems

To address the various issues encountered during the Android x86 installation and setup, I took a series of targeted troubleshooting steps and workarounds:

1. **Reconfigured Virtual Machine Settings:**
   I adjusted the base memory to 4 GB and allocated 2 processor cores. I also experimented with toggling EFI, enabling/disabling 3D acceleration, and ensuring 20 that nested paging and virtualization (VT-x/AMD-V) were enabled in the BIOS. These settings were critical to improve compatibility with Android x86.

2. ISO File Management:

I made sure the Android x86 ISO image was properly attached in the virtual machine's storage settings. When the ISO wasn't initially recognized, I reattached it manually using the virtual machine's storage menu to ensure the VM could boot from it.

3. Switched from VirtualBox to VMware:

After repeated failures with VirtualBox, I switched to VMware, where the installation proceeded more smoothly. VMware provided better hardware emulation for this setup, which allowed the Android system to boot and install successfully.

4. Workaround for GRUB Boot Issues

: Although the GRUB menu initially blocked direct booting into Android, I eventually navigated the boot options using Tab instead of e, which allowed me to tweak the boot commands when needed. This workaround helped bypass the graphical boot issues temporarily.

5. Installed Termux for Development Tools:

Since Android's default terminal lacked gcc, adb, and other essential system programming tools, I installed Termux, a powerful terminal emulator for Android. Within Termux, I installed required packages like clang, make, and other Linux development tools that enabled me to compile and run C programs using system calls like clock_settime().

6. Handled File Permissions and Compilation Warnings: (this will be seen in the system call implementation)

I resolved file permission issues by using appropriate directories like Termux's home folder, and carefully managed file access with chmod and mv commands. I also adjusted the code to resolve compiler warnings — for instance, by explicitly casting time_t to long long in printf() statements to match the expected format.

7. Developed in Nano Editor inside Termux

: I used the nano text editor in Termux to write and modify C programs. This made it easy to return to files like mytime.c and make changes incrementally, which was especially helpful when debugging. 21

Which Filesystem Did I Use?

During the installation process of Android x86, I selected the "ext4" filesystem for formatting the virtual hard drive. This choice was based on its native compatibility with Linux-based systems like Android, and its proven stability and performance.

Why ext4?

• It is the default and most stable Linux filesystem supported by Android x86.

• It offers journaling, better performance, and fewer compatibility issues.

• During installation, the Android x86 installer likely asked if I wanted to format the virtual hard drive. Choosing ext4 was either the default or the most recommended option presented.

Advantages of Android-x86

Android-x86 is a port of the Android operating system designed to run natively on x86-based hardware like desktops, laptops, and tablets, instead of ARM architecture used in most mobile devices. It offers several advantages:

1. Compatibility with x86 Hardware:

a. Designed to work efficiently on computers with x86 architecture (like Intel and AMD processors). This allows us to run Android apps on desktops and laptops seamlessly.

2. Desktop Experience: a. Features a desktop-like user interface tailored for larger screens and peripherals (mouse, keyboard), making it ideal for productivity or gaming.

3. Access to Android Apps: a. Enjoy access to millions of Android apps on the Google Play Store. This provides versatility for gaming, media consumption, or productivity tasks.

4. Multi-Tasking Features: a. Offers multi-window functionality, allowing you to use multiple apps at the same time. This is a feature often missing from mobile devices. 5. Performance Optimization: a. Unlike emulators (e.g., Bluestacks or Genymotion), Android-x86 runs directly on the hardware, resulting in better performance and lower resource usage. 22

6. Open-Source Flexibility: a. Being an open-source project, it allows developers and enthusiasts to modify and customize the OS according to their needs, such as tweaking settings for unique hardware configurations.

7. Cost-Effective Solution: a. You can repurpose older computers or laptops that may no longer support modern Windows or Linux operating systems by installing Android-x86, breathing new life into the hardware.

8. Variety of Use Cases: a. Android-x86 supports touch-enabled devices, making it great for tablets or convertible laptops. It’s also useful for app testing, gaming, or creating an Android-specific development environment. Disadvantages of Android-x86 Android-x86 is an innovative solution for running Android on x86 devices, but it's not without its drawbacks. Here are some notable disadvantages:

1. Compatibility Issues • Not all x86 devices are fully supported, which means some hardware (e.g., graphics cards, sound cards, or network adapters) may not work properly. • Limited drivers can cause problems such as missing Wi-Fi connectivity or poor GPU performance. 2. Not Optimized for Desktop Usage • While Android-x86 adapts Android to x86 hardware, the user interface is still primarily designed

for touch-based mobile devices. This can feel awkward when using a keyboard and mouse. • Android apps may not perform or display correctly on larger screens. 23 3. Performance Limitations • Although it runs directly on hardware, Android-x86 may face performance bottlenecks, especially on older machines or when handling resource-intensive apps like games. • Animations and transitions can feel sluggish compared to mobile devices optimized for Android. 4. Limited Software Support • Some apps are designed exclusively for ARM architecture and may not work on Android-x86 without modifications or emulation. • This limits access to certain apps, particularly those relying on native ARM libraries. 5. Stability Concerns • Android-x86 may experience occasional crashes or bugs, especially in complex applications or multitasking scenarios. • This makes it less reliable compared to Android running on native devices. 6. Security Risks • Running Android-x86 on a desktop system may expose it to security vulnerabilities due to outdated Android versions or lack of regular security patches. • It's harder to lock down Android-x86 compared to a mobile device because desktops don't have secure boot and other built-in protections. 7. Not Ideal for Primary OS • Android-x86 may be great for testing or secondary use, but it’s rarely suitable as a primary operating system for traditional computing tasks like file management, coding, or office productivity. Conclusion The process of installing and configuring Android-x86 on a virtual machine was a mix of challenges and learning opportunities. While I initially faced several issues such as booting 24 errors, difficulties with GRUB, and compatibility problems, the experience helped me better understand the intricacies of operating systems, especially when it comes to virtual environments and cross-platform compatibility. By fine-tuning the settings, such as adjusting the memory allocation, enabling virtualization features in the BIOS, and ensuring proper disk image mounting, I was able to overcome many of the obstacles. I opted for the ext4 filesystem, which is widely supported in Linux based environments like Android, and it worked well for the installation. Ultimately, while Android-x86 is not without its drawbacks—such as incomplete hardware support and limitations in user experience—it remains an excellent tool for testing and development. It provided a stable, lightweight alternative to using physical Android devices and allowed me to explore Android's capabilities on a desktop-like system. Though there were some hiccups, the process was valuable for deepening my understanding of system configuration and virtual environments. Future Outlook / Recommendation Although Android-x86 serves as a functional adaptation of the Android operating system for x86-based PCs, there are noticeable areas where the experience feels out of place. 1. Optimizing Keyboard and Mouse Usability: • Android-x86 retains its mobile-first design, which leads to awkward experiences when using traditional input devices like keyboards and mice. This disconnect makes desktop usage feel strange and inefficient. • Recommendation: Leverage Android-x86's open-source nature to introduce desktop-oriented enhancements, such as: o Adaptive UI scaling for larger screens. o Improved input handling that prioritizes keyboard shortcuts and mouse functionality over touch gestures. • These changes would provide a smoother and more intuitive experience for desktop users. • Streamlining Account Switching: • Current Android-x86 systems only allow switching between accounts after the device has powered on, which adds unnecessary steps and consumes time during startup. • Recommendation: Implement a login screen at the startup phase, similar to traditional desktop operating systems like Windows or Linux. This would: o Allow users to choose an account during the boot process. 25 o Enhance convenience and align Android-x86 with the expectations of desktop users. • This feature would make account management faster and more seamless, improving user experience in multi-account scenarios. What is Virtualization? Virtualization is a technology that

allows the creation of virtual versions of physical resources, such as operating systems, servers, storage, or networks. In a virtualized environment, multiple virtual instances can run on a single physical machine, effectively separating hardware from software. Why Virtualization? 1. Resource Optimization: a. Virtualization enables efficient utilization of hardware resources by running multiple virtual machines (VMs) on a single physical machine. b. It reduces the need for dedicated hardware for every system or application. 2. Isolation: a. Virtual machines operate independently of one another, providing security and fault isolation. If one VM crashes or is compromised, it doesn't affect others. 3. Flexibility: a. Virtualization simplifies testing, deployment, and system management. Developers can create isolated environments for app development and testing without affecting the host system. 4. Cost Efficiency: a. Reduces hardware costs by consolidating workloads onto fewer machines. b. Lowers operational expenses by streamlining system management and scalability. How Virtualization Works Virtualization works through a software layer called a hypervisor. The hypervisor creates, manages, and isolates virtual machines by allocating system resources (CPU, memory, disk) to each VM while ensuring they operate independently. 26 Android-x86 often operates in virtualized environments as part of app development, testing, or emulation. Here's how virtualization ties into Android-x86: 1. Running Android on Non-Mobile Devices: a. Android-x86 is commonly run inside virtual machines (e.g., VMware or VirtualBox). Virtualization allows users to test Android apps on desktops/laptops without relying on physical Android devices. 2. App Development and Testing: a. Developers use virtualized instances of Android-x86 to create isolated environments for app testing. They can simulate different hardware configurations and Android versions within VMs. 3. Resource Efficiency: a. Virtualizing Android-x86 enables developers to run multiple Android environments on a single machine. This is particularly useful for testing apps on various configurations simultaneously. 4. Isolation for Security: a. Using Android-x86 in a VM ensures that any crashes or malware remain confined to the virtual machine, preventing harm to the host operating system. Implementation of clock_settime system call on Android-x86 Android's native environment does not provide direct support for running C language code or utilizing essential tools like the GCC (GNU Compiler Collection) library out of the box. This limitation arises because Android focuses primarily on Java or Kotlin-based applications for app development, leaving system-level programming tasks less accessible to regular users. As a result, the default terminal emulator on Android lacks the necessary compilation tools for C programs. To overcome this limitation, developers and enthusiasts turn to Termux, a powerful terminal emulator application available as an APK for Android devices. Termux bridges the gap by offering a Linux-like environment with access to essential development tools. With Termux installed, users can set up a fully functional C development environment by downloading libraries like GCC, Clang, and other utilities directly within the app. This enables the compilation, testing, and execution of C code seamlessly on Android devices. Using Termux for system programming provides flexibility and convenience, allowing users to explore low-level programming tasks without the need for external hardware or additional 27 software. Its versatility makes it a vital tool for developers looking to experiment with system calls like clock_settime() in a simplified, portable environment.