实验一 编译 Linux 内核

班级： 07112303 学号： 1120231863 姓名： 左逸龙

一、 实验目的

本实验旨在通过一次完整的 Linux 内核编译实践，加深对操作系统的构成与工作机制的理解。具体而言，实验目的涵盖以下几个方面：

1. 熟悉 Linux 内核的模块化设计方法与源代码的组织结构，加深对内核架构的认识。
2. 掌握 Linux 内核从获取源代码、配置编译选项、执行编译到最终部署的全过程，理解其背后的工具链（如 GCC、Make 等）。
3. 深入理解内核配置的原理与方法，学习通过定制内核功能的方式来优化系统性能、减小内核体积或满足特定硬件需求。
4. 了解操作系统的引导流程，学习如何加载新编译的内核，并验证其是否成功运行。
5. 培养在大型系统软件构建过程中分析、定位并解决实际问题的能力。

二、 实验内容

本次实验的核心内容是在 Windows Subsystem for Linux 2 (WSL2) 环境中，对 6.6.87 版本 Linux 内核进行完整的编译、部署与验证。实验涵盖的关键环节如下：

1. 在 Ubuntu 24.04 (WSL2) 操作系统环境中，安装并配置编译 Linux 内核所需的核心工具链及依赖库，包括 GCC、Make、Flex、Bison 等。
2. 从 Linux 内核官方网站 (kernel.org) 获取 6.6.87 版本内核源代码。
3. 基于当前系统运行的内核配置，通过 `make menuconfig` 工具进行个性化定制，主要为修改内核的本地版本字符串 (Local version) 以便后续识别与验证。
4. 利用多核处理器并行编译内核源代码，并最终生成可引导的内核映像文件 (`bzImage`)。
5. 根据 WSL2 的内核加载机制，将新生成的内核映像部署到指定位置，并修改 `.wslconfig` 配置文件以引导新内核。
6. 重启 WSL2 环境后，通过 `uname` 命令验证新内核是否已成功加载并运行。

三、 实验步骤与结果分析

3.1 基本实验环境

本实验在 Windows 10 上通过 WSL2 部署的 Ubuntu 24.04 LTS 环境进行。我们首先通过以下指令查看环境的基本信息:

```
# 查看操作系统版本
cat /etc/os-release

# 查看系统详细信息
uname -a

# 查看CPU信息
lscpu

# 查看内存信息
free -h
```

输出结果如下:

1. `cat /etc/os-release`

```
PRETTY_NAME="Ubuntu 24.04.1 LTS"
NAME="Ubuntu"
VERSION_ID="24.04"
VERSION="24.04.1 LTS (Noble Numbat)"
VERSION_CODENAME=noble
ID=ubuntu
ID_LIKE=debian
HOME_URL="https://www.ubuntu.com/"
SUPPORT_URL="https://help.ubuntu.com/"
BUG_REPORT_URL="https://bugs.launchpad.net/ubuntu/"
PRIVACY_POLICY_URL="https://www.ubuntu.com/legal/terms-and-policies/
privacy-policy"
UBUNTU_CODENAME=noble
LOGO=ubuntu-logo
```

可见操作系统版本为 Ubuntu 24.04.1 LTS。

2. `uname -a`

```
Linux DESKTOP-VARHBN1 6.6.87.2-microsoft-standard-WSL2 #1 SMP
PREEMPT_DYNAMIC Thu Jun  5 18:30:46 UTC 2025 x86_64 x86_64 x86_64 GNU/
Linux
```

可见系统架构为 x86_64，且当前内核版本为 6.6.87.2-microsoft-standard-WSL2。

3. `lscpu`

```
Architecture:             x86_64
  CPU op-mode(s):         32-bit, 64-bit
  Address sizes:          39 bits physical, 48 bits virtual
  Byte Order:             Little Endian
CPU(s):                   16
  On-line CPU(s) list:    0-15
Vendor ID:                GenuineIntel
  Model name:             Intel(R) Core(TM) i9-9880H CPU @ 2.30GHz
    CPU family:           6
    Model:                158
    Thread(s) per core:   2
    Core(s) per socket:   8
    Socket(s):            1
    Stepping:             13
    BogoMIPS:             4608.00
    Flags:                fpu vme de pse tsc msr pae mce cx8 apic sep
mtrr pge mca cmov pat pse36 clflush mmx fxsr sse s
se2 ss ht syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon
rep_good nopl xtopology cpuid                           tsc_known_freq
pni pclmulqdq ssse3 fma cx16 pdcm pcid sse4_1 sse4_2 movbe popcnt aes
xsave av                          x f16c rdrand hypervisor lahf_lm abm
3dnowprefetch ssbd ibrs ibpb stibp ibrs_enhanced fsgsbase
bmi1 avx2 smep bmi2 erms invpcid rdseed adx smap clflushopt xsaveopt
xsavec xgetbv1 xsaves md                          _clear flush_l1d
arch_capabilities
Virtualization features:
  Hypervisor vendor:      Microsoft
  Virtualization type:    full
Caches (sum of all):
  L1d:                    256 KiB (8 instances)
  L1i:                    256 KiB (8 instances)
  L2:                     2 MiB (8 instances)
  L3:                     16 MiB (1 instance)
NUMA:
  NUMA node(s):           1
  NUMA node0 CPU(s):      0-15
Vulnerabilities:
  Gather data sampling:   Unknown: Dependent on hypervisor status
  Itlb multihit:          KVM: Mitigation: VMX unsupported
  L1tf:                   Not affected
  Mds:                    Not affected
  Meltdown:               Not affected
  Mmio stale data:        Vulnerable: Clear CPU buffers attempted, no
microcode; SMT Host state unknown
  Reg file data sampling: Not affected
  Retbleed:               Mitigation; Enhanced IBRS
  Spec rstack overflow:   Not affected
  Spec store bypass:      Mitigation; Speculative Store Bypass disabled
via prctl
  Spectre v1:             Mitigation; usercopy/swapgs barriers and
__user pointer sanitization
  Spectre v2:             Mitigation; Enhanced / Automatic IBRS; IBPB
conditional; RSB filling; PBRSB-eIBRS SW sequence;
```

```
BHI SW loop, KVM SW loop
  Srbds:                   Unknown: Dependent on hypervisor status
  Tsx async abort:         Mitigation; TSX disabled
```

可见 CPU 为 Intel(R) Core(TM) i9-9880H CPU @ 2.30GHz，且有 16 个核心。

4. `free -h`

```
               total        used        free      shared  buff/cache
available
Mem:            31Gi       571Mi        30Gi       2.9Mi        53Mi
30Gi
Swap:          8.0Gi          0B       8.0Gi
```

可见内存为 31G，且有 8G 的 Swap 空间。

3.2 环境准备与依赖安装

编译 Linux 内核首先需要一个完整的构建工具链，包括编译器、构建系统以及内核配置所需的特定库。我们首先通过 `apt` 包管理器安装所有必需的依赖项：

```
sudo apt update && sudo apt install build-essential flex bison
libncurses-dev libssl-dev libelf-dev dwarves
```

`build-essential` 提供了 `gcc` 编译器和 `make` 工具；`flex` 和 `bison` 是词法分析器和语法分析器，用于处理内核配置脚本；`libncurses-dev` 为 `make menuconfig` 提供了文本图形界面（TUI）支持；而 `libssl-dev`、`libelf-dev` 与 `dwarves` 则是新版内核编译过程中处理加密、ELF 文件格式和调试信息所必需的依赖库。

按照指引安装完成后，让我们检查编译链是否可用：

```
# 查看GCC版本
gcc --version

# 查看Make版本
make --version

# 查看flex版本
flex --version

# 查看bison版本
bison --version

# 其他依赖可以类似检查
```

输出结果如下：

1. `gcc --version`

```
gcc (Ubuntu 13.3.0-6ubuntu2~24.04) 13.3.0
Copyright (C) 2023 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is
NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR
PURPOSE.
```

2. `make --version`

```
GNU Make 4.3
Built for x86_64-pc-linux-gnu
Copyright (C) 1988-2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.
html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
```

3. `flex --version`

```
flex 2.6.4
```

4. `bison --version`

```
bison (GNU Bison) 3.8.2
Written by Robert Corbett and Richard Stallman.

Copyright (C) 2021 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is
NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR
PURPOSE.
```

经过检查，所有依赖项均已安装并配置完成。

3.3 内核源代码获取与解压

本次实验选用 Linux 内核 6.6.87 版本，该版本与当前系统运行的内核版本一致，确保切换内核后，WSL2 能够正常运行。我们使用 `wget` 工具从官网下载源代码压缩包，并在本地工作目录中进行解压。

```
# 创建并进入工作目录
mkdir ~/kernel-dev && cd ~/kernel-dev

# 从 kernel.org 官方镜像下载源代码
wget https://cdn.kernel.org/pub/linux/kernel/v6.x/linux-6.6.87.tar.xz

# 解压源代码包
tar -xvf linux-6.6.87.tar.xz
```

```
# 进入源代码根目录
cd linux-6.6.87
```

解压完成后，我们使用 `ls -F` 指令查看源代码目录结构：

```
COPYING  Documentation/  Kconfig     MAINTAINERS  README  block/  crypto/
fs/      init/           ipc/        lib/         net/    samples/  security/  tools/
virt/
CREDITS  Kbuild          LICENSES/   Makefile     arch/   certs/
drivers/  include/  io_uring/  kernel/  mm/   rust/  scripts/  sound/
usr/
```

可以看到包括 `arch/`、`kernel/`、`drivers/`、`fs/` 在内的多个核心目录，表明源代码已完整获取。

3.4 内核编译选项配置

为确保编译的内核与 WSL2 虚拟环境完全兼容，我们必须采用 Microsoft 官方提供的内核配置文件作为编译蓝本。该配置确保所有特定的驱动模块被正确启用，从而避免潜在的引导失败问题。

我们可以在 Microsoft WSL2 的 GitHub 仓库 中找到官方配置文件，并可直接复制该文件至 `config` 文件中。

在官方配置的基础上，我们再进行个性化定制。我们可以通过 `make menuconfig` 工具，为新内核添加可供识别的自定义标识。

```
# 启动文本菜单配置界面
make menuconfig
```

`make menuconfig` 指令执行后，系统展示了内核配置的文本图形化界面。

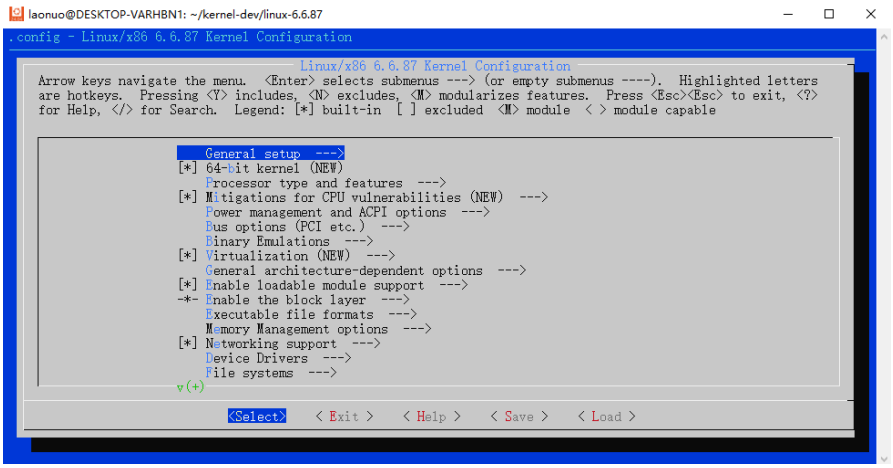

图 1：内核配置主界面（menuconfig）

为了在编译后能够清晰地识别新内核，本次实验对内核的本地版本字符串进行了修改。在配置界面中，导航至 `General setup --->`，找到并修改 `Local`

`version - append to kernel release` 选项，为其追加 `-bit-zyl` 作为自定义标识。

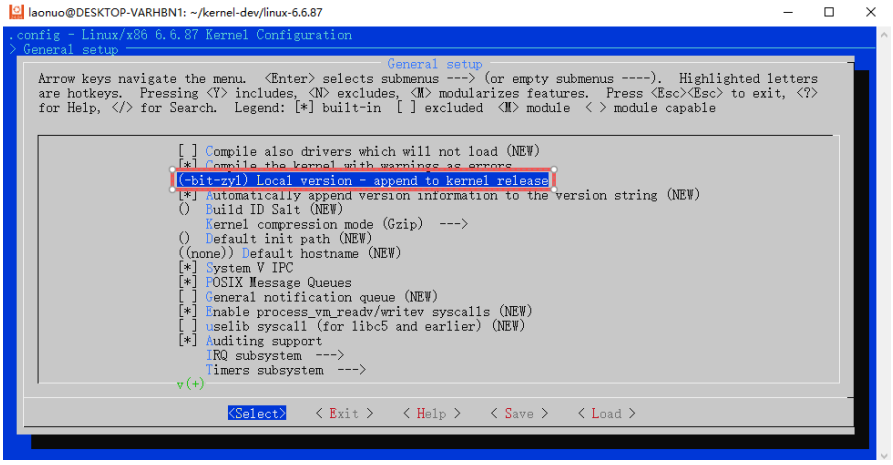

图 2：修改内核本地版本字符串

完成修改后，保存配置并退出即可。

3.5 内核编译

内核编译是一个计算密集型过程。我们可以利用 `make` 工具的并行编译功能，并结合 `nproc` 命令动态获取当前系统的 CPU 核心数，从而最大化利用 CPU 多核心资源，显著缩短编译时间。

```
# 使用所有 CPU 核心并行编译内核
make -j$(nproc)
```

编译过程耗时约 7 分钟。终端输出大量编译日志，最终无错误信息并停止。这表明编译过程顺利完成。编译的核心产物是压缩后的可引导内核映像文件 `bzImage`。我们可以通过以下指令检查该文件：

```
# 查看生成的内核映像文件信息
ls -lh arch/x86/boot/bzImage
```

输出结果如下：

```
-rw-r--r-- 1 laonuo laonuo 13M Oct 15 20:43 arch/x86/boot/bzImage
```

结果表明 `bzImage` 文件已在预期的 `arch/x86/boot/` 目录下生成，文件大小约为 13M。这是部署新内核所需的核心文件。

3.6 内核部署与引导配置（WSL2）

WSL2 的内核引导机制与《操作系统实验教程》当中介绍的传统 GRUB 等引导加载器不同，它通过 Windows 用户目录下的 `.wslconfig` 文件来加载指定的

内核。因此，部署过程分为两步：将编译好的 `bzImage` 文件复制到 Windows 文件系统中，然后修改 `.wslconfig` 指向该文件。

首先，将内核映像文件复制到 Windows 用户目录下一个专用于存放 WSL 内核的文件夹：

```
# 在用户目录下创建文件夹 wsl-kernels 并复制文件
mkdir -p /mnt/c/Users/25912/wsl-kernels
cp arch/x86/boot/bzImage /mnt/c/Users/25912/wsl-kernels/v6.6.87-bit-zyl
```

接着，在 Windows 资源管理器中，导航到 `C:\Users\25912\` 目录，创建并编辑 `.wslconfig` 文件，写入以下内容：

```
[wsl2]
kernel=C:\\Users\\25912\\wsl-kernels\\v6.6.87-bit-zyl
```

`.wslconfig` 文件为 WSL 提供了全局配置，其中 `kernel` 键指定了自定义内核映像的绝对路径。需要特别注意，此处的路径格式需要遵循 Windows 规范，且路径分隔符 `\` 必须进行转义，即写为 `\\`。

3.7 重启并验证新内核

配置完成后，需要彻底关闭所有 WSL 实例，以确保下次启动时能加载新的 `.wslconfig` 配置。在 Windows PowerShell 中执行 `wsl --shutdown` 命令：

```
wsl --shutdown
```

关闭后，重新打开 Ubuntu 24.04 终端。WSL2 会自动使用 `.wslconfig` 文件中指定的新内核进行启动。进入系统后，使用 `uname -r` 命令检查当前运行的内核版本：

```
uname -r
```

终端返回的输出结果为：

```
6.6.87-bit-zyl
```

可见自定义编译的 Linux 内核已被 WSL2 成功加载并正在作为当前操作系统核心运行。本次实验到这里便圆满完成了。

四、 实验收获与体会

通过本次 Linux 内核编译的实践，我对操作系统有了更为具体和深入的理解，将课堂上的理论知识与复杂的工程实践有效地结合起来。实验的主要收获与体会可以归纳为以下几点：

1. 本次实验清晰地揭示了理论与实践之间的关键差异，尤其是在面对特定技术环境的时候。教程中的 GRUB 引导加载器配置方法，在现代的 WSL2 虚拟化环境中已不再适用。WSL2 采用 `.wslconfig` 文件的配置机制，这体现了技术的演进以及不同平台间引导流程的差异性。这使我认识到，在工程实践中，必须将基本原理与特定平台的实现细节相结合，不能机械地套用既有流程。

2. 实验过程中所遇到的 `WSAENOTCONN` 引导失败问题，是本次实践当中最有价值的收获之一。最初，我采用了通用的内核配置作为蓝本来进行编译，但是在引导过程当中遇到了失败的问题，这暴露了通用内核配置在特定虚拟化平台上的局限性。问题分析过程让我深刻理解到，WSL2 作为一个高度集成的虚拟化环境，其稳定运行依赖于一系列特定的 Hyper-V 驱动模块，如 `hv_sock`。标准的 Linux 配置默认并未启用这些模块，从而导致主机与虚拟机之间的通信链路在引导阶段无法建立。这一经历充分证明了内核配置的精确性与平台适配性的重要性，也锻炼了通过错误现象反推技术根源的故障排查能力。

3. 实验过程加深了我对内核构建系统（`Kbuild`）智能性的理解。在解决配置问题并重新编译时，我观察到了“瞬间完成”的现象，这直观地展示了 `make` 工具的增量编译机制。它通过比对文件时间戳，仅对已修改的源文件进行重新编译，极大地提高了开发和调试效率。这让我对大型项目的高效构建策略有了具象化的认识。

总而言之，本次实验不仅让我掌握了编译 Linux 内核的全过程，更重要的是，它提供了一次宝贵的、在真实环境中分析并解决复杂问题的机会。从环境配置、依赖管理，到核心的内核配置与故障排查，每一步都加深了我对操作系统底层工作原理的理解。这次实践显著提升了我驾驭大型软件系统的信心与能力。