

# 实验一：初探Linux与环境配置

## 第一部分：实验目的

学习使用docker;

学习linux(la-xv6)的使用方法

学习并使用若干Linux指令

掌握xv6内核编译方法以及GDB调试方法

## 第二部分：实验环境

### 使用Docker镜像搭建

#### 概念

最重要的两个概念是“容器”和“镜像”。

- 容器：类比于“虚拟机”。（也可类比作“进程”。）
- 镜像：类比于“装某个虚拟机用的 ISO 文件”。（也可类比作“可执行文件”。）

技术上说，容器和虚拟机有不少差别，不过对我们的实验来说了解到这里就足够了。

#### 安装并启动 Docker

Docker 的安装可参考官方文档[Get Docker | Docker Documentation](#)

##### Windows

下载并安装[Docker Desktop on Windows](#)

如果选择 WSL2 作为后端，启动 Docker Desktop 可能会提示缺少 [WSL2](#)，从此链接下载安装即可(也可以按照提示执行wsl update)

## macOS

下载并安装[Docker Desktop on Mac](#)

## Linux(Ubuntu20)

使用包管理器安装Docker

```
1 sudo apt update
2 sudo apt install docker.io
3 sudo usermod -aG docker $USER # 使当前用户不需要 sudo 就可以使用 docker 命令
4 sudo chmod a+rw /var/run/docker.sock
5 sudo systemctl restart docker
```

## 验证是否安装成功

使用 `docker run hello-world` 命令验证Docker是否安装成功，如果出现以下内容则说明安装成功

```
1 Hello from Docker!
2 This message shows that your installation appears to be working correctly.
```

## Docker常用命令

```
1 docker version # 查看 Docker 版本
2 docker run hello-world # 运行 hello-world 容器
3 docker ps -a # 查看所有容器
4 docker images # 查看所有镜像（也可以用 docker image ls）
5 docker pull konc1033/la-xv6 # 拉取 ksqs/la-sel4 镜像
6 docker create --name la-xv6 konc1033/la-xv6 # 创建一个名为 la-axv6 的容器
7 docker start -ai la-xv6 # 启动 la-xv6 容器，并进入交互模式
8 docker rm la-xv6 # 删除 la-xv6 容器
```

## 具体用法

### 下载镜像

```
1 docker pull konc1033/la-xv6
```

下载后，可以用 `docker image ls` 确认

## 创建和查看容器

选择共享文件夹，将下列命令中的“绝对路径”替换为此目录的绝对路径

```
1 docker create --name la-xv6 -v "绝对路径:/labs" -it konc1033/la-xv6
```

上述命令将创建该容器。如果一切顺利，你将可以看到一串十六进制数（容器 ID）打印出来。此时，使用 `docker ps -a` 可以看到出现了一个新的容器。（ps 默认只输出正在运行的容器，因此使用 `-a` 强制输出所有容器。）

参数的含义是：

- `--name` 赋予该容器一个名字。
- `-v $DIR_PATH:/labs` 将创建一个“共享文件夹”，路径在容器的 `/labs`。可以通过这个目录来与宿主机交换文件。
- `-i` 表示创建交互式容器。
- `-t` 表示分配伪 TTY。

## 进入容器环境

```
1 docker start -ai la-xv6
```

你将可以看到一个 `bash` 命令行，表示已经在 Ubuntu 环境中了。退出 `shell` 后，可以再次使用该命令回到容器环境中，并且对环境作出的修改仍将保留着（除非你删除了这个容器，或者不小心进入了另一个容器，或者在创建容器时错误地使用了 `--rm` 参数）。

如果报错 `Error response from daemon: dial unix docker.raw.sock: connect: connection refused`，请检查是否启动了 Docker 引擎。

**参数说明：**

- `la-xv6` 是创建时的 `--name` 参数。
- `-a` 表示 `attach`。
- `-i` 表示 `interactive`。

## 删除容器和镜像

```
1 docker rm la-xv6
2 docker rmi konc1033/la-xv6
```

可以使用 `docker container ls -a` 列出所有容器，用 `docker image ls -a` 列出所有镜像。

当然也可以使用图形化界面直接操作

# Ubuntu下手动安装

通过虚拟机、WSL等方式配置linux环境（建议Ubuntu20.04）。然后通过下面所示方法进行安装

## 1.1 下载并安装交叉编译链

通过 `wget` 指令下载压缩包，并通过 `tar` 指令解压交叉编译链并且复制到 `/opt` 目录下

```
1 wget https://github.com/loongson/build-  
tools/releases/download/2022.05.29/loongarch64-clfs-5.0-cross-tools-gcc-  
full.tar.xz  
2  
3 sudo tar -vxf loongarch64-clfs-5.0-cross-tools-gcc-full.tar.xz -C /opt
```

## 1.2 设置环境变量

在1.1中我们将交叉编译工具放在了 `/opt` 文件夹下，如果我们想在任何文件夹直接使用，可以放在环境变量中

setenv.sh内容

```
1 #!/bin/sh  
2 set -x  
3 CC_PREFIX=/opt/cross-tools  
4  
5 export PATH=$CC_PREFIX/bin:$PATH  
6 export LD_LIBRARY_PATH=$CC_PREFIX/lib:$LD_LIBRARY_PATH  
7 export LD_LIBRARY_PATH=$CC_PREFIX/loongarch64-unknown-linux-  
gnu/lib/:$LD_LIBRARY_PATH  
8  
9 set +x
```

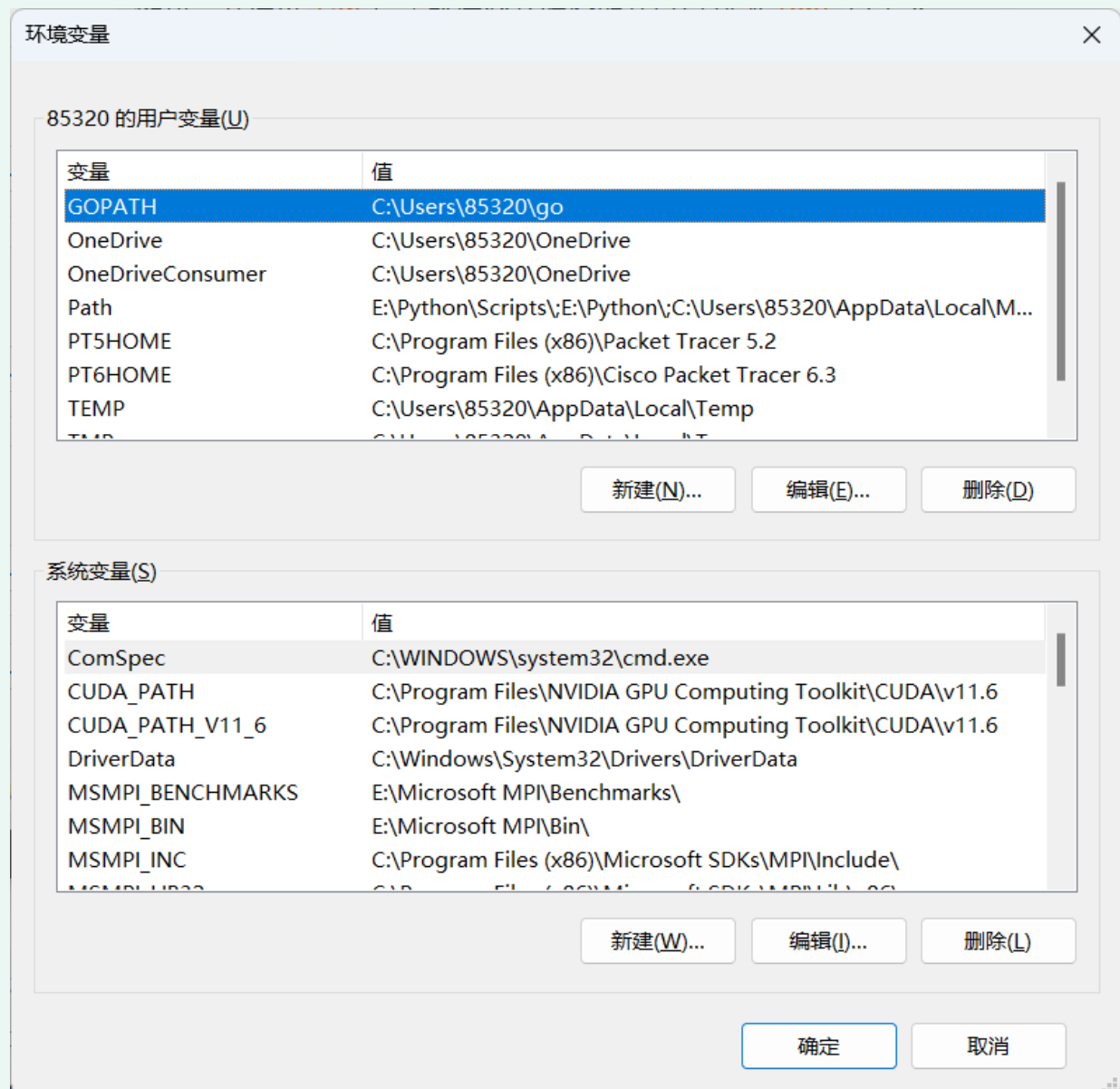
何为环境变量：

环境变量 (**Environment Variable**)

环境变量是在操作系统中一个具有特定名字的对象，它包含了一个或者多个应用程序所将使用到的信息。例如 `PATH`，就是告诉系统，当要求系统运行一个程序而没有告诉他程序所在的完整路径时，系统除了在当前目录下寻找此程序外，还应到那些目录下寻找。

简单来说就是将某些数据，文件或文件夹设置为系统默认值，这样你调用的时候就不用给出完整路径和地址或进行设置，直接用名字就可以了

1.在windows中，环境变量是通过可视化窗口模式展现出来的，环境变量的编辑，查看都可以通过图形化界面进行操作



2.在Linux中

Shell脚本：

上面setenv.sh就是一个shell脚本，下面将讲述shell脚本的作用以及编写方法

### 1. Shell脚本的编写

Shell脚本类似于Windows的.bat批处理文件。一个最简单的Shell脚本长这样：

```
1  #!/bin/bash
2  第一条命令
3  第二条命令
4  第三条命令，以此类推
```

其中，第一行的 `#!/bin/bash` 是一个约定的标记，它告诉系统这个脚本需要什么解释器来执行，即使用哪一种 Shell。Ubuntu下默认的shell是bash。脚本一般命名为 `xxx.sh`。

## 2. Shell脚本的运行

运行Shell脚本有两种方式

1. 通过执行 `sh xxx.sh` 来直接调用解释器运行脚本。其中，sh就是我们的Shell。这种方式运行的脚本，不需要在第一行指定解释器信息。
2. 将该脚本视为可执行程序。首先，保存脚本之后，要通过 `chmod +x xxx.sh` 给脚本赋予执行权限，然后直接 `./xxx.sh` 运行脚本。

举例：首先编译abc.cpp并输出一个名为aabbcc的二进制可执行文件，然后执行aabbcc，最后删掉aabbcc，上述操作可以使用如下脚本来实现：

```
1 #!/bin/bash
2 gcc -o aabbcc abc.cpp
3 ./aabbcc
4 rm aabbcc
```

# 第三部分：编译、调试XV6内核

## 3.1 先导知识

### 3.1.1 系统内核启动过程

Linux kernel在自身初始化完成之后，需要能够找到并运行第一个用户程序（此程序通常叫做“init”程序）。用户程序存在于文件系统之中，因此，内核必须找到并挂载一个文件系统才可以成功完成系统的引导过程。

在grub中提供了一个选项“root=”用来指定第一个文件系统，但随着硬件的发展，很多情况下这个文件系统也许是存放在USB设备，SCSI设备等等多种多样的设备之上，如果需要正确引导，USB或者SCSI驱动模块首先需要运行起来，可是不巧的是，这些驱动程序也是存放在文件系统里，因此会形成一个悖论。

- 1 为解决此问题，Linux kernel提出了一个RAM disk的解决方案，把一些启动所必须的用户程序和驱动模块放在RAM disk中，这个RAM disk看上去和普通的disk一样，有文件系统，有cache，内核启动时，首先把RAM disk挂载起来，等到init程序和一些必要模块运行起来之后，再切换到真正的文件系统之中。

但是，这种RAM disk的方案（下称initrd）虽然解决了问题但并不完美。比如，disk有cache机制，对于RAM disk来说，这个cache机制就显得很多余且浪费空间；disk需要文件系统，那文件系统（如ext2等）必须被编译进kernel而不能作为模块来使用。

Linux 2.6 kernel提出了一种新的实现机制，即initramfs。顾名思义，initramfs只是一种RAM filesystem而不是 disk。initramfs实际是一个cpio归档，启动所需的用户程序和驱动模块被归档成一个文件。因此，不需要cache，也不需要文件系统。

### 3.1.2 什么是initramfs

initramfs 是一种以 cpio 格式压缩后的 rootfs 文件系统，它通常和 Linux 内核文件一起被打包成boot.img 作为启动镜像。

BootLoader 加载 boot.img，并启动内核之后，内核接着就对 cpio 格式的 initramfs 进行解压，并将解压后得到的 rootfs 加载进内存，最后内核会检查 rootfs 中是否存在 init 可执行文件（该init 文件本质上是一个执行的 shell 脚本），如果存在，就开始执行 init 程序并创建 Linux 系统用户空间 PID 为 1 的进程，然后将磁盘中存放根目录内容的分区真正地挂载到 / 根目录上，最后通过 `exec chroot . /sbin/init` 命令来将 rootfs 中的根目录切换到挂载了实际磁盘分区文件系统中，并执行 /sbin/init 程序来启动系统中的其他进程和服务。

基于ramfs开发的initramfs取代了initrd。

### 3.1.3什么是initrd

initrd代指内核启动过程中的一个阶段：临时挂载文件系统，加载硬盘的基础驱动，进而过渡到最终的根文件系统。

initrd也是早期基于ramdisk生成的临时根文件系统的名称。现阶段虽然基于initramfs，但是临时根文件系统也 依然存在某些发行版称其为initrd。例如，CentOS 临时根文件系统命名为 initramfs-`uname -r`.img，Ubuntu 临时根 文件系统命名为 initrd-`uname -r`.img（uname -r 是系统内核版本）。

### 3.1.4 QEMU

QEMU是一个开源虚拟机。可以在里面运行Linux甚至Windows等操作系统。

## 3.2 下载、安装

### 3.2.1本机下载内核并编译

#### 1. 安装并且初始化Git

Docker 中已经安装了 Git，如果打算在 Docker 中使用，可跳过安装过程。如果想在宿主机上使用 Git，可以参考 Git 官方文档 安装 Git，或者安装 GUI 界面的 Git 客户端，如 GitHub Desktop。使用 Git 首先需要设置用户名和邮箱，这些信息会被记录在每次提交的 commit 中，用于标识提交者。

```
1 git config --global user.name "Your Name"
2 git config --global user.email "Your Email"
```

#### 2. 常用的Git命令



```
1 git status # 查看当前仓库的状态
2 git add . # 将所有修改添加到暂存区
3 git commit -m "commit message" # 将暂存区的内容提交到本地仓库
4 git push # 将本地仓库的内容推送到远程仓库
5 git pull # 将远程仓库的内容拉取到本地仓库
6 git log # 查看当前仓库的提交历史
7 git reset --hard HEAD~ # 回退到上一个版本
8 git checkout abcd1234 # 切换到某个 commit 或 branch
```

### 3. 将实验仓库克隆到本地

```
1 git clone https://gitlab.eduxiji.net/202310358111361/project1466467-176097.git
```

4. 在根目录下 `make all` 将编译出相应的文件镜像 `fs.img` 以及在 `/kernel` 文件夹下出现二进制文件 `kernel`，而后进入到 `\qemu-loongarch-runenv` 文件夹下执行以下命令运行停止后将会出现 `$` 符号表示内核已经启动

```
1 ./run_loongarch.sh -k ../kernel/kernel
```

## 3.2.2使用docker容器

1. 将文件clone到本地放入共享文件夹中
2. 其余步骤同上

## 3.3 GDB调试xv6内核

gdb是一款命令行下常用的调试工具，可以用来打断点、显示变量的内存地址，以及内存地址中的数据等。使用方法是 `gdb 可执行文件名`，即可在gdb下调试某二进制文件。

一般在使用gcc等编译器编译程序的时候，编译器不会把调试信息放进可执行文件里，进而导致gdb知道某段内存里有内容，但并不知道这些内容是变量a还是变量b。或者，gdb知道运行了若干机器指令，但不知道这些机器指令对应哪些C语言代码。所以，在使用gdb时需要在编译时加入 `-g` 选项，如：`gcc -g -o test test.c` 来将调试信息加入可执行文件。而Linux内核采取了另一种方式：它把符号表独立成了另一个文件，在调试的时候载入符号表文件就可以达到相同的效果。

- gdb常用命令



```

1 r/run # 开始执行程序
2 b/break <location> # 在location处添加断点，location可以是代码行数或函数名
3 b/break <location> if <condition> # 在location处添加断点，仅当condition条件满足才中断运行
4 c/continue # 继续执行到下一个断点或程序结束
5 n/next # 运行下一行代码，如果遇到函数调用直接跳到调用结束
6 s/step # 运行下一行代码，如果遇到函数调用则进入函数内部逐行执行
7 ni/nexti # 类似next，运行下一行汇编代码（一行c代码可能对应多行汇编代码）
8 si/stepi # 类似step，运行下一行汇编代码
9 list # 显示当前行代码
10 p/print <expression> # 查看表达式expression的值
11 q # 退出gdb

```

### 3.3.1 安装gdb

在交叉编译链中已经安装了龙芯版本的gdb，具体目录为 `/opt/gdb/bin/loongarch64-unknown-linux-gnu-gdb`

### 3.3.2 启动gdb server

使用3.2.1所述指令运行qemu。但需要加上-d -D 两个参数。这两个参数的含义如下：

参数	含义
-d	启动gdb server调试内核，server端口是1234。若不想使用1234端口，则可以使用 -gdb tcp:xxxx 来取代此选项。
-D	若使用本参数，在qemu刚运行时，CPU是停止的。你需要在gdb里面使用c来手动开始运行内核。

### 3.3.3 建立连接

**另开**一个终端，运行gdb(执行指令 `/opt/gdb/bin/loongarch64-unknown-linux-gnu-gdb`)，然后在gdb界面运行如下命令：

```

1 target remote:1234 # 建立gdb与gdb server间的连接。这时候我们看到输出带有??，还报了一条warning
2 # 这是因为没有加载符号表，gdb不知道运行的是什么代码。
3 c # 手动开始运行内核。执行完这句后，你应该能发现旁边的qemu开始运行了。
4 q # 退出gdb。

```

### 3.3.4加载符号表

1. 重新执行3.3.2
2. 另开一个终端，运行gdb，然后再gdb界面运行如下命令：

```
1 target remote:1234 # 建立gdb与gdb server间的连接。这时候我们看到输出带有??，
2 # 这是因为没有加载符号表，gdb不知道运行的是什么代码。
3 file ../kernel/kernel # 加载符号表。
4 n # 单步运行。此时可以看到右边不是??，而是具体的函数名了。
5 break main # 设置断点在start_kernel函数。
6 c # 运行到断点。
7 l # 查看断点代码。
```

也可以直接在启动gdb时加载符号表 `/opt/gdb/bin/loongarch64-unknown-linux-gnu-gdb ../kernel/kernel`

## 第四部分：实验任务

### 4.1 安装Docker/WSL/虚拟机/Linux系统

四种安装方式选一即可

### 4.2 了解Linux相关命令

1. 我们现场随便给出一条本实验文档未涉及的指令，你需要自己使用 man 指令阅读其手册简要解释该指令的含义，并简要介绍任意一个参数的含义。本部分共1分。若遇到不认识的英文单词，可以现场搜索。
2. 在助教的电脑上或机房的电脑上，在助教的监督下，使用实验提供的测试程序进行现场测试，每人最多尝试 2次，取最高分。答题时不得打小抄、查阅实验文档。从题库中抽4题，每题抽4个选项，选项顺序会随机打乱。答题总时间120秒。每题1分。满分4分。为防止同一学生不同助教处刷次数，每一次尝试都记录成绩。

题库见提供的csv文件，三个csv都是题库。csv文件里，第一列是题目，第二列是正确答案，其余是干扰项。公布的题库、自测程序和考察时使用的完全相同。csv文件是UTF-8 with BOM格式，可以直接用Excel打开。但 Excel可能会将部分选项解释异常，因此建议使用文本编辑器打开。自测程序由Python语言编写，在Linux下，在自测程序目录下执行 `python3 test.py` 即可运行测试程序。Ubuntu自带Python3。若未安装Python3，可用 apt自行安装。强烈建议大家在检查之前自测几次，熟悉程序的工作流程。为防止替考及背完就忘的情况，在后续实验中，如果发现有同学忘了指令含义，我们可能会考虑让其重做一遍测试题，并按照错误个数扣实验分。

## 4.3启动xv6内核并启动调试

1. 现场检查能否启动虚拟机并启动gdb调试，即现场执行3.2。本部分共5分。本部分检查中，允许学生对照实验 文档操作。
2. 编写shell脚本自动化编译内核并且启动qemu