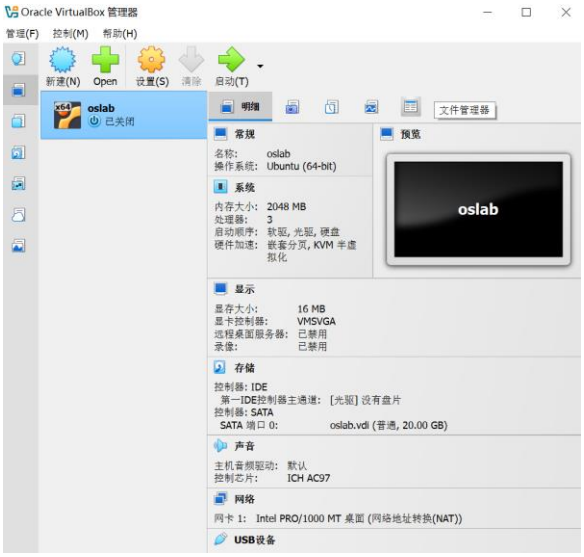
Lab1 实验报告

姓名：李雨朔　学号：20232131090　班级：计科一班　日期：2025.9.16

一．　xv6 的环境搭建过程

1. 安装虚拟机软件：在 Windows10 主机上安装 Oracle VirtualBox。
   （https://www.virtualbox.org/wiki/Downloads）



2. 下载 Ubuntu Server 镜像：从官网下载 Ubuntu Server 24.04 LTS 镜像文件。
   （https://ubuntu.com/download/server）

3. 创建虚拟机：在 VirtualBox 中创建一台新的虚拟机，分配 20GB 硬盘、2GB 内存。

4. 安装 Ubuntu Server 系统。

5. 在 Ubuntu 虚拟机中安装实验所需工具，在虚拟机内部依次执行以下命令：

```
# 1. 更新软件包列表
sudo apt update

# 2. 安装实验所需的工具：编译器、调试器、QEMU 模拟器等
sudo apt install git build-essential gdb-multiarch qemu-system-misc -y
```

```
# 3. 下载 xv6 的源代码
git clone git://g.csail.mit.edu/xv6-labs-2024

# 4. 进入源代码目录
cd xv6-labs-2024

# 5. 切换到第一个实验需要的代码分支（util）
git checkout util

# 6. 编译并启动 xv6 系统
make qemu
```

但是当执行 make qemu 命令时编译失败："Error: Couldn't find a riscv64 version of GCC/binutils." (找不到 RISC-V 64 位的 GCC 编译器和二进制工具集)。出现该错误的原因是：主机（Ubuntu 虚拟机）是 x86_64 架构的 CPU，xv6 是为 riscv64 架构的 CPU 编写的，之前安装的 build-essential 包只包含了为本机（x86_64）编译代码的工具，无法编译 xv6，需要一个在 x86_64 机器上运行、但能生成 riscv64 代码的编译器。

为解决上述问题，需要安装 RISC-V 交叉编译工具链，在终端中输入以下命令安装 gcc-riscv64-unknown-elf（针对 RISC-V 架构的 GCC 编译器）和 binutils-riscv64-unknown-elf（针对 RISC-V 架构的二进制工具）：

```
sudo apt install gcc-riscv64-unknown-elf binutils-riscv64-unknown-elf -y
```

安装完工具链后，清理错误文件并重新编译：

```
# 执行清理命令，删除之前编译产生的所有文件
make clean

# 重新编译并启动 xv6
make qemu
```

xv6 环境搭建完成。

二． 程序

在 xv6 中编辑、编译和运行以下三个程序的总体流程如下：

（1） 使用 nano 创建和修改相关 .c 源文件：

```
# 进入 xv6 的用户程序目录
cd ~/xv6-labs-2024/user/
# 创建并编辑 .c 文件
nano 文件名.c
```

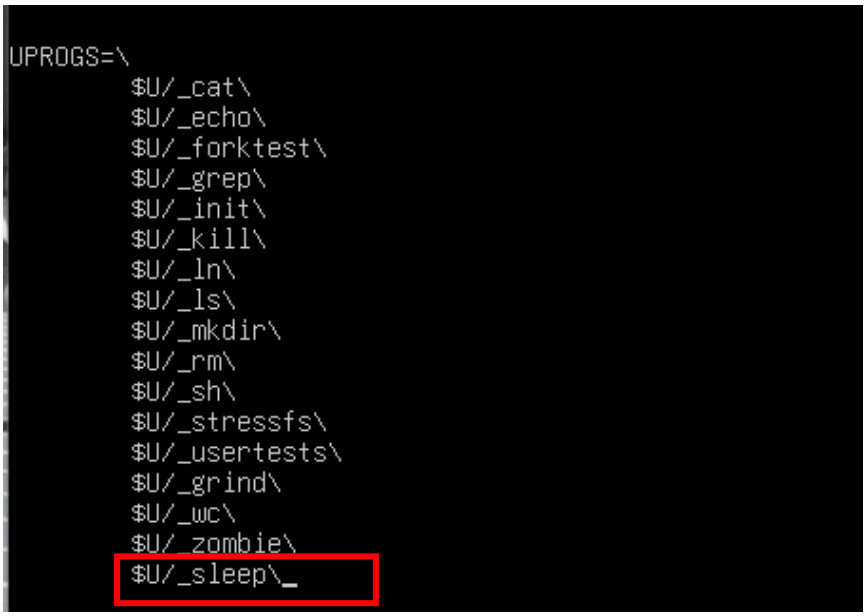
编辑完成后，按 ctrl + o 然后回车保存源文件，然后按 ctrl + x 退出。

（2） 将程序添加到编译系统：

打开 Makefile：

```
cd ~/xv6-labs-2024/
nano Makefile
```

在 Makefile 中找到 UPROGS 然后在其下面列出的多个以“&U”开头的字符串的下方按照相同的格式添加 sleep、pingpong、primes 三个程序，例如 sleep 程序：



保存并退出编辑器。

（3） 编译并运行 xv6：

回到 xv6 源码根目录敲出“make qemu”，让 .c 文件得以编译并在 xv6 内部形成可执行程序：

```
cd ~/xv6-labs-2024/
make qemu
```

看到提示符$，则编译成功，进入 xv6 的终端。

（4） 在 xv6 中运行程序（以 sleep、pingpong、primes 三个程序为例）：
运行 sleep：

```
$ sleep 5
# 程序会暂停约 5 秒，然后返回命令提示符
```

如果只输入 sleep 而不带参数，输出错误信息。

运行 pingpong：

```
$ pingpong
```

运行 primes：

```
$ primes
```

具体源代码如下：

1. sleep
   1) 解题思路：
      (1) sleep 程序需要接收一个参数（休眠的时钟周期数）；
      (2) 若用户未输入参数，打印错误信息；
      (3) 将接收的字符串参数转换为整数后，调用 xv6 提供的 sleep 系统调用；
      (4) 休眠结束后，调用 exit(0)退出程序。

   2) 代码和相关注释：

```
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"
int main(int argc, char *argv[]) {
  //检查参数数量是否正确
```

```
 if (argc != 2) {
      //打印错误信息
      fprintf(2, "Usage: sleep <time>\n");
      //非正常退出


      exit(1);
   }

   //将字符串参数转换为整数
   int sleep_time = atoi(argv[1]);
   // 调用系统调用 sleep
   sleep(sleep_time);

   //  正常退出
   exit(0);
}
```

2. pingpong
   1) 解题思路：
      (1) 创建管道，p[0]：用于从管道读取数据，p[1]：用于向管道写入数据。
      (2) 使用 fork()系统调用创建子进程
      (3) 子进程只读，关闭写端->阻塞等待父进程 ping->回 pong
      (4) 父进程只写，关闭读端->发 ping->等待子进程退出->读 pong

   2) 代码和相关注释：

```
#include "kernel/types.h"
#include "user/user.h"

int main() {
   int p[2];
   char buf[1];          // 读写缓冲区

   // 创建管道
   if (pipe(p) < 0) {
      fprintf(2, "pipe failed\n");
      exit(1);
   }
```

```
    // 创建子进程
    int pid = fork();
    if (pid < 0) {
        fprintf(2, "fork failed\n");
        exit(1);
    }

    if (pid == 0) {        // 子进程
        close(p[1]);       // 关闭写端

        // 从父进程读取数据
        read(p[0], buf, 1);
        printf("%d: received ping\n", getpid());

        // 向父进程发送回应
        write(p[0], "x", 1);

        close(p[0]);
        exit(0);
    } else {               // 父进程
        close(p[0]);       // 关闭读端

        // 向子进程发送数据
        write(p[1], "x", 1);

        // 等待子进程完成
        wait(0);

        // 从子进程读取回应
        read(p[1], buf, 1);
        printf("%d: received pong\n", getpid());

        close(p[1]);
        exit(0);
    }
}
```

运行截图:

```
balloc: first 855 blocks have be
balloc: write bitmap block at se
qemu-system-riscv64 -machine vir
ormat=raw,id=x0 -device virtio-b

xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ pingpong
4: received ping
3: received pong
$ pingpong
6: received ping
5: received pong
$ pingpong
8: received ping
7: received pong
$
```

3. primes
   1) 解题思路：
      (1) sieve 函数：
      a. 关闭管道写端->从管道读端读取第一个数字，该数字一定是质数->若读取失败，直接退出->打印该质数
      b. 尝试读取下一个数字（若没有则退出）->创建新管道 new_p->调用 fork()创建子进程
         a) 子进程：关闭当前管道的读端 p[0]->递归调用 sieve(new_p)，处理下一阶段
         b) 父进程：关闭新管道的读端 new_p[0]->从当前管道读取剩余数字，去掉当前质数的倍数->不能被当前质数整除的数字，写入新管道的写端 new_p[1]->读完所有数字后，关闭 p[0]和新管道的写端 new_p[1]->调用 wait(0)等待子进程退出
      (2) main 函数：
         (1) 创建管道 p
         (2) 调用 fork()创建子进程
         a. 父进程：关闭管道读端 p[0]->向管道写端 p[1]写入 2~280->关闭写端->调用 wait(0)等待子进程退出
         b. 子进程：调用 sieve(p)函数

   2) 代码和相关注释

```
#include "kernel/types.h"
#include "user/user.h"

void sieve(int p[2]) __attribute__((noreturn));    //函数永不返回，避免尾递归优化警告

void sieve(int p[2]) {
  int prime;
  int n;
```

```
    // 关闭不必要的写端
    close(p[1]);

    // 读取第一个数字，它一定是质数
    if (read(p[0], &prime, sizeof(prime)) != sizeof(prime)) {
        close(p[0]);
        exit(0);
    }

    printf("prime %d\n", prime);

    // 尝试读取下一个数字
    if (read(p[0], &n, sizeof(n)) <= 0) {
        close(p[0]);
        exit(0);
    }

    // 创建新的管道用于下一阶段
    int new_p[2];
    if (pipe(new_p) < 0) {
        fprintf(2, "pipe failed\n");
        exit(1);
    }

    int pid = fork();
    if (pid < 0) {
        fprintf(2, "fork failed\n");
        exit(1);
    }

    if (pid == 0) {
        // 子进程：递归处理下一阶段
        close(p[0]);
        sieve(new_p);
    } else {
        // 父进程：过滤当前质数的倍数
        close(new_p[0]);

        do {
            if (n % prime != 0) {
                write(new_p[1], &n, sizeof(n));
            }
        } while (read(p[0], &n, sizeof(n)) > 0);
```

```
        close(p[0]);
        close(new_p[1]);
        wait(0);
        exit(0);
    }
}

int main() {
    int p[2];
    if (pipe(p) < 0) {
        fprintf(2, "pipe failed\n");
        exit(1);
    }

    int pid = fork();
    if (pid < 0) {
        fprintf(2, "fork failed\n");
        exit(1);
    }

    if (pid == 0) {
        // 子进程：开始筛选过程
        sieve(p);
    } else {
        // 父进程：生成数字 2 到 280
        close(p[0]);

        for (int i = 2; i <= 280; i++) {
            write(p[1], &i, sizeof(i));
        }

        close(p[1]);
        wait(0);
        exit(0);
    }

    return 0;
}
```

运行截图：

三． 实验主观心得

1. 在本次实验中，最大的困难是配置环境，在这个过程中出现的错误次数也最多。但是 AI 帮了我很大的忙：在安装 Ubuntu 时，因为语言无法选择中文，只能使用英文，其中有很多地方对我来说很难理解，通过 AI 翻译，才逐步理解各个配置选项的意思，完成配置。尤其是“English (US, intl., with dead keys)”和标准美式布局的区别，通过 AI 我知道了前者会影响符号输入，不适合编程。首次编译 xv6 时提示缺少 riscv64-unknown-elf-gcc，在 AI 的帮助下我才意识到需要安装交叉编译工具链并通过它提供给我的方法解决了这一错误。在虚拟机中运行 xv6 时，我不知道各种操作需要用什么键，同样也是通过 AI 的帮助学习到的。
2. 但是使用 AI 也要谨慎！在搭建实验环境时，AI 提供给我的步骤中有一项显示需要重启，在开机时进入某个界面的操作，因为我在使用 AI 前搜索过相关环境配置教程，其中并没有这一环节，于是又去问了同学，发现没有这一步骤也是可以的（至于按照 AI 那一步来做会不会出现什么问题，我暂未去实践）。
3. pingpong 程序让我第一次实际使用 fork() 和 pipe()。最初输出会出现混乱（如 3:

rece4: received ipwing)，查了下 AI，才发现是父子进程输出没有同步，后来通过 wait(0) 等待子进程结束再输出，解决了问题。

4. primes 程序是最复杂的，尤其是递归创建进程和管道传递数据的部分。一开始编译器报“infinite recursion”错误，后来我重新仔细阅读文档，发现需要添加 __attribute__((noreturn)) 声明并确保每个分支都有 exit(0) 来消除警告，这才完成了实验。
5. 总的来说，AI 在我完成实验的过程中起到了非常重要的作用，也更加意识到了正确使用 AI 以及“专业知识+AI”的组合的重要性，没有一定的专业知识打底，过度依赖 AI，就无法真正发挥出 AI 的作用。