

# 同济大学操作系统课程设计 - Xv6 - Lab1 - Xv6 and Unix utilities

---

## 【实验目的】

---

### 1. 理解操作系统的基本结构：

- Xv6 是一个简化版的 Unix 操作系统，它保留了 Unix 的核心概念，但代码量较少，便于学习和理解。通过学习 Xv6 可以深入理解操作系统的基本组件，如进程管理、内存管理、文件系统、设备驱动等。

### 2. 掌握 Unix 实用程序的使用和开发：

- Unix Utilities 部分通常包括对常见 Unix 工具的学习，如 `grep`、`ls`、`cat` 等。这不仅涉及如何使用这些工具，还可能涉及如何在系统级别实现类似的功能，从而加深对 Unix 系统的工作原理的理解。

### 3. 学习调试和测试操作系统：

- 在实验过程中，需要使用调试工具（如 `gdb`）来跟踪和修复代码中的问题。这有助于提升调试复杂系统的技能，同时了解如何在低层次上分析和解决系统问题。

### 4. 提高对操作系统原理的理论理解：

- 通过实验，我们能够更好地理解经典操作系统理论，如进程同步、死锁、虚拟内存管理、文件系统设计等，并能够将这些理论应用到实际系统的实现和优化中。

## 【实验环境】

---

- 虚拟机：VMare Workstation 17
- 操作系统：Ubuntu-20.04.6
- 实验系统：xv6-labs-2021

## 【实验内容】

---

### 1 Boot Xv6

启动xv6，创建分支，运行qemu并确定qemu安装是否正确。

#### 1. 获取 xv6 源代码并检出 util 分支：

```
$ git clone git://g.csail.mit.edu/xv6-labs-2021
$ cd xv6-labs-2021
$ git checkout util
Branch 'util' set up to track remote branch 'util' from 'origin'.
Switched to a new branch 'util'
```

```
yzz@ubuntu: ~/Desktop/xv2021/xv6/lab1/xv6-labs-2021
yzz@ubuntu:~/Desktop/xv2021/xv6/lab1$ git clone git://g.csail.mit.edu/xv6-labs-2021
Cloning into 'xv6-labs-2021'...
remote: Enumerating objects: 7051, done.
remote: Counting objects: 100% (7051/7051), done.
remote: Compressing objects: 100% (3423/3423), done.
remote: Total 7051 (delta 3702), reused 6830 (delta 3600), pack-reused 0
Receiving objects: 100% (7051/7051), 17.20 MiB | 2.65 MiB/s, done.
Resolving deltas: 100% (3702/3702), done.
warning: remote HEAD refers to nonexistent ref, unable to checkout.

yzz@ubuntu:~/Desktop/xv2021/xv6/lab1$ cd xv6-labs-2021
yzz@ubuntu:~/Desktop/xv2021/xv6/lab1/xv6-labs-2021$ git checkout util
Branch 'util' set up to track remote branch 'util' from 'origin'.
Switched to a new branch 'util'
```

2. 使用make qemu指令运行xv6:

```
yzz@ubuntu: ~/Desktop/xv2021/xv6/lab1
yzz@ubuntu:~/Desktop/xv2021/xv6/lab1$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ █
```

3. 输入ls能看到输出, 这是根目录下的文件。

```
yzz@ubuntu:~/Desktop/xv2021/xv6/lab1$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ ls
.          1 1 1024
..         1 1 1024
README    2 2 2226
xargstest.sh 2 3 93
cat        2 4 23904
echo       2 5 22736
forktest   2 6 13088
grep       2 7 27256
init       2 8 23832
kill       2 9 22704
ln         2 10 22656
ls         2 11 26136
mkdir      2 12 22808
```

xv6 没有 `ps` 命令, 但是如果输入, 他将打印有关每个进程的信息。要退出 qemu, 键入: `Ctrl-a x`。

## 2 Sleep Xv6

为xv6实现程序sleep, sleep程序应该暂停用户指定的时钟周期数。一个时钟周期的长度是由xv6内核指定的。通过实现sleep程序, 应用sleep系统调用, 并实现错误处理。

1. 查看 `user/` 中的其他一些程序（例如，`user/echo.c`、`user/grep.c`、和 `user/rm.c`）看如何获取传递给程序的命令行参数。

```
1  #include "kernel/types.h"
2  #include "kernel/stat.h"
3  #include "user/user.h"
4
5  int
6  main(int argc, char *argv[])
7  {
8      int i;
9
10     for(i = 1; i < argc; i++){
11         write(1, argv[i], strlen(argv[i]));
12         if(i + 1 < argc){
13             write(1, " ", 1);
14         } else {
15             write(1, "\n", 1);
16         }
17     }
18     exit(0);
19 }
20
```

如果用户忘记传递参数，sleep 应该打印一条错误消息，

2. 仿照以上内容，在 `user` 中编写 `sleep.c` 程序，如下所示：

```
int main(int argc, char *argv[])
{
    // 检查命令行参数的数量。如果不是两个参数（程序名称和时间），则输出错误信息并退出程序。
    if (argc != 2)
    {
        // 使用文件描述符 2（标准错误输出）打印错误信息。
        fprintf(2, "Error! format = 'sleep time'\n");
        // 退出程序，返回状态码 1，表示错误发生。
        exit(1);
    }

    // 将输入的字符串参数转换为整数，并赋值给变量 i。
    int i = atoi(argv[1]);

    // 调用 sleep 函数，使程序暂停执行 i 秒。
    sleep(i);

    // 正常退出程序，返回状态码 0，表示成功执行。
    exit(0);
}
```

3. 将程序添加到 `Makefile` 中：

```
UPROGS=\
    $U/_cat\
    $U/_echo\
    $U/_forktest\
    $U/_grep\
    $U/_init\
    $U/_kill\
    $U/_ln\
    $U/_ls\
    $U/_mkdir\
    $U/_rm\
    $U/_sh\
    $U/_sleep\
    $U/_stressfs\
    $U/_usertests\
    $U/_grind\
    $U/_wc\
    $U/_zombie\
    $U/_pingpong\
    $U/_primes\
    $U/_find\
    $U/_xargs\
```

4. 在命令行中测试 sleep 函数，成功

```
yz@ubuntu: ~/Desktop/xv2021/xv6/lab1
yz@ubuntu:~/Desktop/xv2021/xv6/lab1$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp
3 -nographic -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-devi
ce,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ sleep 3
$ sleep
Error! format = 'sleep time'
$ sleep 2 3
Error! format = 'sleep time'
$
```

### 3 pingpong Xv6

编写一个使用 UNIX 系统调用 "ping-pong" 的程序 一对管道上的两个进程之间的字节，每个管道一个方向。父级应向子级发送一个字节; 孩子应打印“: received ping”，其中 是其进程 ID，将管道上的字节写入父级，并退出; 父级应该从子级读取字节，打印“: received pong”，并退出。解决方案应该在文件 `user/pingpong.c` 中。

- 使用 `pipe` 创建管道。
- 使用 `fork` 创建子项。
- 使用 `read` 从 pipe 读取数据，使用 `write` 写入 pipe 数据。
- 使用 `getpid` 查找调用进程的进程 ID。
- 将程序添加到 `Makefile` 中的 `UPROGS` 中。

因此，对应的程序如下图所示：

```

3
4 int main(int argc, char *argv[])
5 {
6     int parent_to_child_fd[2]; // 父进程到子进程的管道文件描述符数组
7     int child_to_parent_fd[2]; // 子进程到父进程的管道文件描述符数组
8
9     // 创建两个管道，分别用于父进程和子进程之间的数据传输
10    pipe(parent_to_child_fd);
11    pipe(child_to_parent_fd);
12
13    char buffer[8]; // 存储传输的数据
14
15    // 使用 fork() 创建子进程，fork() 的返回值用于判断当前进程是父进程还是子进程
16    if (fork() == 0) {
17        // 子进程
18        read(parent_to_child_fd[0], buffer, 4); // 从父进程读取数据
19        printf("%d: received %s\n", getpid(), buffer); // 打印收到的数据
20        write(child_to_parent_fd[1], "pong", strlen("pong")); // 向父进程发送数据
21    }
22    else {
23        // 父进程
24        write(parent_to_child_fd[1], "ping", strlen("ping")); // 向子进程发送数据
25        wait(0); // 等待子进程完成
26        read(child_to_parent_fd[0], buffer, 4); // 从子进程读取数据
27        printf("%d: received %s\n", getpid(), buffer); // 打印收到的数据
28    }
29
30    exit(0); // 退出程序
31 }

```

结果如下：

```

yzz@ubuntu:~/Desktop/xv2021/xv6/lab1$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp
3 -nographic -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-devi
ce,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ pingpong
4: received ping
3: received pong
$

```

## 4 Primes

目标是使用 `pipe` 和 `fork` 进行设置管道。第一个进程提供数字 2 到 35 进入管道。对于每个素数，安排创建一个通过管道从其左邻居读取数据的进程 并通过另一个管道写入其右侧邻居。由于 xv6 具有文件描述符和进程的数量有限，第一个进程可以在 35 时停止。

程序的实现如下所示：

```

int main(int argc, char const *argv[])
{
    int pipe_fds[2]; // 定义用于父子进程通信的管道
    pipe(pipe_fds); // 创建管道

    if (fork() != 0)
    {
        // 父进程
        close(pipe_fds[0]); // 关闭管道的读端

        for (int i = 2; i <= 35; i++)
        {
            // 将数字 2 到 35 写入管道
            write(pipe_fds[1], &i, sizeof(int));
        }

        close(pipe_fds[1]); // 关闭管道的写端
    }
    else
    {
        // 子进程
        close(pipe_fds[1]); // 关闭管道的写端
        filter_primes(pipe_fds[0]); // 调用 filter_primes 函数进行质数过滤
    }

    wait(0); // 等待子进程结束
    exit(0);
}

```

其中, `filter_primes` 函数如下:

```

void filter_primes(int input_fd)
{
    int prime_number;
    if (read(input_fd, &prime_number, sizeof(int)) == 0)
    {
        // 如果没有读取到任何数字, 说明管道已空, 退出函数
        exit(0);
    }
    printf("prime %d\n", prime_number); // 输出当前的质数
    int new_pipe[2];
    pipe(new_pipe); // 创建一个新的管道, 用于子进程通信
    if (fork() == 0)
    {
        // 子进程
        close(new_pipe[1]); // 关闭新的管道的写端
        filter_primes(new_pipe[0]); // 递归调用 filter_primes 处理新的输入
    }
    else
    {
        // 父进程
        close(new_pipe[0]); // 关闭新的管道的读端
        int number;
        int bytes_read;

        while ((bytes_read = read(input_fd, &number, sizeof(int))) != 0)
        {
            // 如果数字不能被当前的质数整除, 则写入新的管道
            if (number % prime_number != 0)
            {
                write(new_pipe[1], &number, sizeof(int));
            }
        }
    }
}

```

```

    }

    close(new_pipe[1]); // 关闭新的管道的写端
}
wait(0); // 等待子进程结束
exit(0);
}

```

这段代码实现了一个经典的并发编程示例，通常用于展示管道（pipe）和进程间通信的工作原理。代码的核心思想是通过递归创建子进程来筛选出质数。程序通过管道将一系列整数从父进程传递给子进程，然后子进程筛选出这些整数中的质数，并将未筛选的整数递归地传递给下一个子进程。每个子进程负责筛选掉能被自己接收到的第一个数字（即质数）整除的所有数字，并将剩余的数字传递给下一个子进程。每个父进程在完成其工作后，会等待其子进程结束（`wait(0)`），确保所有子进程都执行完毕。

程序首先创建父子进程，并通过管道在它们之间传递数据；父进程生成数字序列并写入管道，子进程读取这些数字并通过递归函数筛选质数。子进程会再创建子进程以处理剩余的数字，直到所有数字都被筛选，最终父进程等待子进程完成后退出。

同样将其添加到Makefile中的UPROGS中，执行结果如下：

```

$ primes
prime 2
prime 3
prime 5
prime 7
prime 11
prime 13
prime 17
prime 19
prime 23
prime 29
prime 31

```

## 5 find

编写 UNIX find 程序的简单版本：查找所有文件 在具有特定名称的目录树中。解决方案 应该在文件 `user/find.c` 中。

- 使用递归允许 find 下降到子目录中。
- 对文件系统的更改在 qemu 运行期间仍然存在;获取 运行干净的文件系统，然后运行 `make clean`, `make qemu`
- 需要使用 C 字符串。
- `==` 不会像在 Python 中那样比较字符串。改用 `strcmp()`。
- 将程序添加到 Makefile 中的 UPROGS 中。

代码如下：

```

#include "kernel/types.h"
#include "user/user.h"
#include "kernel/stat.h"
#include "kernel/fs.h"

void
find(char *dir, char *file)
{
    char buf[512], *p;

```

```

int fd;
struct dirent de;
struct stat st;
if ((fd = open(dir, 0)) < 0)
{
    fprintf(2, "Error! find: cannot open %s\n", dir);
    return;
}
if (fstat(fd, &st) < 0)
{
    fprintf(2, "Error! find: cannot stat %s\n", dir);
    close(fd);
    return;
}
if (st.type != T_DIR)
{
    fprintf(2, "Error! find: %s is not a directory\n", dir);
    close(fd);
    return;
}
if(strlen(dir) + 1 + DIRSIZ + 1 > sizeof buf)
{
    fprintf(2, "Error! find: directory too long\n");
    close(fd);
    return;
}
strcpy(buf, dir);
p = buf + strlen(buf);
*p++ = '/';
while (read(fd, &de, sizeof(de)) == sizeof(de))
{
    if(de.inum == 0)
        continue;
    if (!strcmp(de.name, ".") || !strcmp(de.name, ".."))
        continue;
    memmove(p, de.name, DIRSIZ);
    if(stat(buf, &st) < 0)
    {
        fprintf(2, "Error! find: cannot stat %s\n", buf);
        continue;
    }
    if (st.type == T_DIR)
    {
        find(buf, file);
    }
    else if (st.type == T_FILE && !strcmp(de.name, file))
    {
        printf("%s\n", buf);
    }
}
}

int
main(int argc, char *argv[])
{

```



```
if (argc != 3)
{
    fprintf(2, "Error! format = 'find dirName fileName'\n");
    exit(1);
}
find(argv[1], argv[2]);
exit(0);
}
```

这个代码实现了一个简单的文件查找程序，在指定的目录及其子目录中递归查找匹配的文件名，并输出其完整路径。代码的结构和逻辑如下：

1. main 函数：

- **参数验证**：程序启动时接收两个参数，一个是目录名 `dirName`，另一个是文件名 `fileName`。如果参数数量不为 3（包括程序本身），则输出用法提示并退出。
- **查找调用**：通过 `find` 函数在指定目录中查找文件。

2. find 函数：

- **打开目录**：使用 `open` 函数尝试打开给定的目录。如果打开失败，打印错误信息并返回。
- **获取目录状态**：使用 `fstat` 函数获取目录的状态信息，确认它是否为一个目录。如果不是目录，打印错误信息并返回。
- **路径长度检查**：检查路径长度是否超出缓冲区大小。如果路径太长，打印错误信息并返回。
- **读取目录内容**：
  - **跳过特殊目录**：忽略当前目录 `.` 和父目录 `..`，避免无限递归。
  - **递归查找**：如果读取的项是子目录，则递归调用 `find` 函数，在该子目录中继续查找。
  - **文件匹配**：如果读取的项是文件且名称匹配给定的文件名，打印该文件的完整路径。

3. 关闭文件描述符：每次打开的目录在处理完成后，都会关闭相应的文件描述符以释放资源。

利用递归函数遍历目录树，逐层深入查找目标文件。通过 `open`、`read`、`fstat` 等系统调用与文件系统交互，获取目录和文件的相关信息。拼接路径和文件名，进行字符串比较，确保查找的正确性

例如，输入以下输入：

```
echo > b
mkdir a
echo > a/b
find . b

find a b
```

显示结果如下：

```
xv6 kernel is booting
```

```
hart 2 starting
```

```
hart 1 starting
```

```
init: starting sh
```

```
$ echo > b
```

```
$ mkdir a
```

```
$ echo > a/b
```

```
$ find . b
```

```
./b
```

```
./a/b
```

```
$
```

即为试验成功。

## 6 xargs

编写 UNIX xargs 程序的简单版本：从 标准输入 并为每行运行一个命令，将行提供为 参数 添加到命令中。解决方案 应位于文件 `user/xargs.c` 中。

1. 理解xarg的工作原理：

```
yzz@ubuntu:~/Desktop/xv2021/xv6/lab1$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp
3 -nographic -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-devi
ce,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ echo hello too | xargs echo bye
bye hello too
$
```

2. 编写 `xargs.c` 的代码程序，如下：

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

int main(int argc, char *argv[]) {
    char inputBuf[32]; // 缓冲区，用于存储从标准输入读取的输入
    char charBuf[320]; // 存储命令行参数的缓冲区
    char* charBufPointer = charBuf; // 指向当前缓冲区位置的指针
    int charBufSize = 0; // 当前缓冲区中存储的字符数量

    char *commandToken[32]; // 存储分隔后的命令行参数
    int tokenSize = argc - 1; // 初始命令行参数个数（不包括程序本身）
    int inputSize = -1;

    // 初始化命令行参数，将初始参数复制到commandToken
    for(int tokenIdx = 0; tokenIdx < tokenSize; tokenIdx++)
        commandToken[tokenIdx] = argv[tokenIdx+1];

    while((inputSize = read(0, inputBuf, sizeof(inputBuf))) > 0) {
        for(int i = 0; i < inputSize; i++) {
            char curChar = inputBuf[i];
            if(curChar == '\n') { // 当读取到换行符时，执行命令
                charBuf[charBufSize] = 0; // 在字符串末尾添加终止符
                commandToken[tokenSize++] = charBufPointer; // 将当前参数加入到命令参数数组中
            }
            charBufPointer++;
            charBufSize++;
        }
    }
}
```

```

commandToken[tokenSize] = 0; // 在命令参数数组末尾添加空指针

if(fork() == 0) { // 创建子进程以执行命令
    exec(argv[1], commandToken);
}
wait(0); // 等待子进程完成
tokenSize = argc - 1; // 重置参数数量
charBufSize = 0; // 重置缓冲区
charBufPointer = charBuf; // 重置指针
}
else if(curChar == ' ') { // 遇到空格时，分隔命令参数
    charBuf[charBufSize++] = 0; // 标记当前字符串结束
    commandToken[tokenSize++] = charBufPointer; // 将分隔的参数加入数组
    charBufPointer = charBuf + charBufSize; // 更新指针，指向新参数的起始位置
}
else {
    charBuf[charBufSize++] = curChar; // 将当前字符添加到缓冲区
}
}
}
exit(0);
}

```

这个算法实现了一个简单的 `xargs` 工具功能，它从标准输入读取字符串（通常是命令行参数），然后将这些参数与命令行初始参数组合，逐行执行命令。程序通过解析输入中的空格和换行符来分隔不同的参数，在读取到完整的一行后，创建一个子进程执行组合后的命令，等待子进程执行完毕后继续处理下一行输入。

3. 在命令行中输入以下命令：`xargstest.sh`，显示如下：

```

xv6 kernel is booting
hart 1 starting
hart 2 starting
init: starting sh
$ sh < xargstest.sh
$ $ $ $ $ hello
hello
hello
$ $

```

## 【分析讨论】

通过这个实验，我初步了解了 xv6 操作系统内核的基本结构和 qemu 模拟器的使用方法。通过 `make qemu` 命令，可以编译并在 qemu 中运行 xv6。如果一切正常，`make` 会执行一系列编译和链接操作，输出大量日志，并最终在 qemu 中启动 xv6 系统。系统启动后，`init` 进程会启动一个 shell 等待用户输入命令。在这个简易的 shell 中，可以使用 `ls` 指令列出根目录下的文件。

实验中，通过实现一个简单的质数筛选器，掌握了如何使用管道和递归调用来实现进程间的数据传递。每个子进程负责筛选下一个质数，并将剩余的数字传递给下一个子进程。这一过程加深了我对 `fork` 系统调用的理解：父进程和子进程在 `fork()` 调用后的代码是独立执行的，拥有各自的地址空间，因此需要在 `fork()` 后进行合适的逻辑分支，确保父子进程各自执行不同的任务，并实现数据传递。

此外，通过这个实验，我深入了解了管道的概念和使用方法，以及父子进程间的通信机制。通过使用 `pipe()` 函数创建管道，`fork()` 函数创建子进程，并利用文件描述符进行进程间的读写操作，实现了父子进程间的有效通信。在通信过程中，确保正确打开和关闭管道非常重要，以避免进程阻塞或死锁。同时，通过适当的管道读写操作和进程等待机制（如使用 `wait()` 函数），实现了父子进程的同步，确

保了数据的正确交换和打印顺序。

通过本次实验，我不仅学到了管道通信和父子进程间的基本通信机制，还体会到这些技术在实际应用中的重要性和广泛应用场景。这个实验提供了一个切实的机会，让我深入理解了进程同步、资源共享、以及系统调用在操作系统中的关键作用。

## 【实验验证】

---

新建 `time.txt`，输入自己做实验的用时，运行 `make grade` 进行评分：

```
== Test sleep, no arguments ==
$ make qemu-gdb
sleep, no arguments: OK (3.5s)
== Test sleep, returns ==
$ make qemu-gdb
sleep, returns: OK (0.7s)
== Test sleep, makes syscall ==
$ make qemu-gdb
sleep, makes syscall: OK (0.7s)
== Test pingpong ==
$ make qemu-gdb
pingpong: OK (1.0s)
== Test primes ==
$ make qemu-gdb
primes: OK (0.8s)
== Test find, in current directory ==
$ make qemu-gdb
find, in current directory: OK (1.4s)
== Test find, recursive ==
$ make qemu-gdb
find, recursive: OK (1.5s)
== Test xargs ==
$ make qemu-gdb
xargs: OK (1.5s)
== Test time ==
time: OK
Score: 100/100
yzz@ubuntu:~/Desktop/xv2021/xv6/lab1$
```