实验二 添加Linux系统调用

实验目的

- 学习如何添加Linux系统调用：实现一个简单的ps
- 学习如何使用Linux系统调用：实现一个简单的shell

实验环境

- OS: Ubuntu 18.04
- Linux内核版本: 4.9.263

实验时间安排

> 注：此处为实验发布时的安排计划，请以课程主页和课程群内最新公告为准

- 4.19实验发布
- 4.23晚实验课，讲解实验及检查
- 4.30晚实验课，检查实验
- 5.7晚实验课，实验补检查

  > 补检查分数照常给分，但会有标记记录此次检查未按时完成，标记会在最后综合分数时作为一种参考。

实验报告提交

> 实验检查的目的主要在于确认结果，而实验的过程很难在较短的检查时间内得到验证，所以实验报告在展示结果的同时也是为了体现出**你在做实验过程中的付出**。所以我们建议实验报告至少要有**设计**和**实现**两个部分分别展示对于解决问题的思考以及实际解决的过程，还要有**测试**部分来展示解决问题的程度（在展示可以解决哪些问题的同时，也可以试着去考虑是否有某些场景下当前的解决方案无法生效，认识不足，更好地改进）。最后，也欢迎在实验报告中写出你对于本次实验的思考，为我们未来更好的实验设计提下宝贵的意见和建议。

1. 本实验要求提交实验报告和代码

- 按下面描述的方式组织相关文件

```
- 顶层目录（可自行命名，如EXP2）
  - EXP2.1 子目录1
    - linux （该文件夹下包含你添加系统调用时修改到的文件）
      - syscalls_64.tbl 文件
      - syscalls.h 文件
      - sys.c 文件
    - 调用了添加的系统调用的程序源码(如process_status.c)
  - EXP2.2 子目录2
    - shell程序的源码(如shell.c)
  - 实验报告(学号+姓名+实验二命名，提交PDF版本)
```

- 将上述文件压缩
  - 格式为 .7z/.rar/.zip
  - 命名格式为 学号_姓名_实验2，如果上传后需要修改，请将文件重新命名为学号_姓名_实验2_修改n (n为修改版本)，以最后修改版本为准。
    - 如PB10001000_张三_实验2.zip ，PB10001000_张三_实验2_修改3.zip

2. 提交到bb系统

- bb系统地址：https://www.bb.ustc.edu.cn/
- 上传至作业区: 第二次实验
- 实验提交截止日期：2021-05-09 18:00

3. 实验报告评分

- 满分10分
- 与实验分的10分之间会按一定比例综合得到本次实验的最终分数（实验报告占分不会超过三成）
- 在**不抄袭**和完成两个部分的实验报告的前提下，基本可以拿到满分，扣分只会存在于以下两种情况
  - 报告中各种可以展示出你的实验并非独立完成的现象
    - **抄袭，5分以下没跑了**
    - 实验代码和报告中的实现思路不符
    - 测试结果与实验代码应该呈现出的结果不符
  - 报告中无法展示出你独立完成
    - 设计实现部分的缺失，很贫瘠的报告（比如只复制了一遍源码上去）也会扣分
    - 测试分析部分的缺失，不能证明你"实现了xx功能"的测试结果。

> **友情提示**:
> 本次实验以实验一为基础。一些步骤（如如何启动qemu运行Linux内核）不会在实验说明中详述。如果有不熟悉的步骤，请复习实验一。
> pdf上文本的复制有时候会丢失空格，有时候会增加不必要的空格，有时候还会增加不必要的回车。有些指令一行写不下分成两行了，一些同学就漏了第二行。如果出了bug，建议各位先仔细确认自己输入的指令是否正确。要**逐字符**比对。每次输完指令之后，请观察一下指令的输出，检查一下这个输出是不是报错。**请不要无脑复制。**
> 在本次实验中，如果你写出的代码出现了数组越界，则会出现很多奇怪的错误（如段错误（Segmentation Fault）、输出其他数组的值或一段乱码（注意，烫烫烫是Visual C的特性，Linux下没有烫烫烫）、其他数组的值被改变等）。提问前请先排查是否出现了此类问题。

> 如果同学们遇到了问题，请先查询在线文档。在线文档地址：https://docs.qq.com/sheet/DR1dZTnFRTURHc051

第一部分 编写系统调用实现一个Linux ps （Process Status）

1.1 什么是系统调用 (System call)

> 课程中我们学到操作系统中将代码的运行空间分为了**用户空间**和**内核空间**，具体定义和解释请参考课程PPT。

系统调用 (System call, Syscall)是操作系统提供给**用户程序访问内核空间的合法接口。**
系统调用**运行于内核空间，可以被用户空间调用**，是内核空间和用户空间划分的关键所在，它保证了两个空间必要的联系。从用户空间来看，系统调用是一组统一的抽象接口，用户程序**无需考虑接口下面是什么**；从内核空间来看，用户程序不能直接进行敏感操作而是通过这种公共接口间接完成，保证了**内核空间的稳定和安全。**

> 注：我们平时的编程中**为什么没有意识到系统调用的存在**？这是因为应用程序现在一般**通过应用编程接口 (API)来间接使用系统调用**，比如我们如果想要C程序打印内容到终端，只需要使用C的标准库函数API `printf()` 就可以了，而不是使用系统调用 `write()` 将内容写到终端的输出文件（`STDOUT_FILENO`）中。

1.2 系统调用是如何执行的

1. 应用程序调用**库函数（API）**
   比如Linux使用的开源标准C库glibc（GNU libc），它有一个头文件unistd.h中就定义了很多封装好系统调用的函数，如 `read/write`
2. API将**系统调用号存入**EAX，然后**触发软中断**使系统进入内核空间
   具体如何触发软中断可以去参考glibc中函数的实现源码，比如经典的触发中断方式是通过汇编代码

   ```
   int $0x80
   ```

   因为汇编代码的部分都已经实现好内嵌到了glibc的源码中，所以我们对于调用API后从用户空间到内核空间这一过程是毫无察觉的
3. 内核的中断处理函数**根据系统调用号**，调用对应的**内核函数**（系统调用）
   前面的 `read/write`，实际上就是调用了内核函数 `sys_read/sys_write`

   > 注：所以添加系统调用需要**注册中断处理函数和对应的调用号**，内核才能找到这个系统调用和执行对应的内核函数。内核的汇编代码最终会在 `include/generated/asm/syscalls_64.h` 中查找调用号，不过我们并不修改这里，x86平台提供了一个专门用来注册系统调用的文件 `/arch/x86/entry/syscalls/syscall_64.tbl` ，在编译时会运行同目录下的 `syscalltbl.sh` 脚本，将这个文件中登记过的系统调用都生成到前面的 `syscalls_64.h` 文件中。因此我们后面**要添加系统调用就是修改这个tbl文件。**

4. 系统调用完成相应功能，将返回值存入EAX，返回到中断处理函数；

> 注1：系统执行相应功能，调用的是C代码编写的函数，而不是汇编代码，减轻了实现系统调用的负担。系统调用的函数定义在 `include/linux/syscalls.h` 中，因为汇编代码到C代码的参数传递是通过栈实现的，所以可以看到所有系统调用的函数前面都使用了 `asmlinkage` 宏，它意味着编译时限制只使用栈来传递参数。
> 注2：用户空间和内核空间各自的地址是不能直接互相访问的，需要借助函数拷贝来实现数据传递，后面讲解如何添加时会说明。

5. 中断处理函数返回到API中；
6. API将EAX返回给应用程序。

> 注：有一个问题在于，当glibc库没有封装某个系统调用时，我们就**没办法通过使用封装好的API**来调用该系统调用，而**使用 `int $0x80` 触发中断又需要进行汇编代码的编写**，所以这两种方法都不适合我们在添加过新系统调用后再编写测试代码去调用。因此我们后面采用的是第三种方法，使用glibc提供的 `syscall` 库函数，这个库函数只需要传入调用号和对应内核函数要用到的参数，就可以帮助我们完成新系统调用的使用了。

1.3 如何添加一个系统调用

> 注意：
>
> 1. Linux 4.9的文档中有说明如何添加一个系统调用，参考 `linux-4.9.263/Documentation/adding-syscalls.txt` 。
> 2. 本实验文档的编写中假设所有同学使用的平台都是x86，采用的是x86平台适配的系统调用添加方法，其他平台的同学可以参考1中文档给出的其他平台系统调用添加方法。

在实现系统调用之前，要先考虑好添加系统调用的函数原型，确定函数名称、要传入的参数个数和类型、返回值的意义。在实际设计中，还要考虑加入系统调用的必要性。

> 我们这里以一个统计系统中进程个数的系统调用 `ps_counter(int *num)` 作为演示，num是统计结束的返回值对应的地址。各位在linux环境下可以使用vim, gedit, vscode等完成代码编辑。

下面几步是没有严格顺序的，但在编译前都要完成：

1.3.1 注册系统调用

打开 `linux-4.9.263/arch/x86/entry/syscalls/syscall_64.tbl` 。

> 注意，此处的 `linux-4.9.263/` 是你linux内核源码的存放路径，你在家目录下直接 `gedit linux-4.9.263/arch/x86/entry/syscalls/syscall_64.tbl` 是打不开的，下同。

```
# linux-4.9.263/arch/x86/entry/syscalls/syscall_64.tbl
# ... 前面还有，但没展示
329 common  pkey_mprotect       sys_pkey_mprotect
330 common  pkey_alloc          sys_pkey_alloc
331 common  pkey_free           sys_pkey_free
# 以上是系统自带的，不是我写的，下面这个是
332 common  ps_counter          sys_ps_counter
```

你可以看到这个文件中已经记录了很多数字与函数名的对应，这里就是**x86平台下的系统调用注册表**，在这个文件中添加你的系统调用注册，添加的格式在文件开头有写，也可以参考已有的系统调用。
每一行从左到右各字段的含义是：为你的系统调用找一个没有用过的数字作为调用号（记住，后面要用），然后类型为common（x86_64和x32都适用的意思），然后你的系统调用名xxx，并对应于后面要声明和实现的函数原型sys_xxx。

> 提示：我们看到每一列用制表符排的很整齐，这其实是因为写代码的人有强迫症。经本人测试，无论你是用制表符还是用空格分隔，无论你用多少空格或制表符，只要你把每一列分开了，无论是否整齐，对这一步的完成都没有影响。

1.3.2 定义函数原型

打开`linux-4.9.263/include/linux/syscalls.h`，里面是对于系统调用函数原型的定义，在最后面加上我们要创建的新系统调用函数原型，格式为`asmlinkage long sys_xxx(...)`，注意如果传入了用户空间的地址，需要加入`__user`宏来说明。

```
asmlinkage long sys_mlock2(unsigned long start, size_t len, int flags);
asmlinkage long sys_pkey_mprotect(unsigned long start, size_t len,
                unsigned long prot, int pkey);
asmlinkage long sys_pkey_alloc(unsigned long flags, unsigned long init_val);
asmlinkage long sys_pkey_free(int pkey);
//这里是我们新增的系统调用
asmlinkage long sys_ps_counter(int __user * num);

#endif
```

1.3.3 实现函数

你可以在`linux-4.9.264/kernel/sys.c`代码的最后添加你自己的函数，这个文件中有很多已经实现的系统调用的函数作为参考。我们给出了一段示例代码：

```
SYSCALL_DEFINE1(ps_counter, int __user *, num){
    struct task_struct* task;
    int counter = 0;
    printk("[Syscall] ps_counter\n");
    for_each_process(task){
        counter ++;
    }
    copy_to_user(num, &counter, sizeof(int));
    return 0;
}
```

- 使用宏函数 `SYSCALL_DEFINEx` 来简化实现的过程，其中x代表参数的个数，传入宏的参数为调用名(不带sys_)、以及每个参数的类型、名称（注意这里输入时为两项）。以我们现在的 `sys_ps_counter` 为例，使用了一个参数所以是 `SYSCALL_DEFINE1(ps_counter, int *, num)` 。如果该系统调用有两个参数，那就应该是 `SYSCALL_DEFINE2(ps_counter, 参数1的类型, 参数1的名称, 参数2的类型, 参数2的名称)` 。以此类推。感兴趣的同学可以把这个宏展开研究一下。
- `printk()` 是内核中实现的print函数，和 `printf()` 的使用方式相同， `printk()` 除了在终端输出以外，还会将信息写入到系统记录中。 `printf()` 是使用了C的标准库函数的时候才能使用的，而内核中无法使用标准库函数。你不能 `#include<stdio.h>` ，自然不能用 `printf()` 。
- 为了获取当前的所有进程，我们使用了宏函数 `for_each_process(p)` ，定义在 `include/linux/sched.h` 中，遍历当前所有任务的信息结构体 `task_struct` （同样定义在 `include/linux/sched.h` 中），并**将地址赋值给参数 `p` ，后面实验内容中我们会进一步用到 `task_struct` 中的成员变量来获取更多相关信息**，注意我们暂时不讨论多线程的场景，每一个 `task_struct` 对应的可以认为就是一个进程。
- 前面说到，系统调用是在内核空间中执行，所以我们在内核空间获取的值如果要传给用户空间，**则需要使用函数 `copy_to_user(void __user *to, const void *from, unsigned long n)` 来完成从内核空间到用户空间的复制**， `from` 和 `to` 是复制的来源和目标地址， `n` 是复制的大小，我们这里就是 `sizeof(int)` 。

1.4 测试

1.4.1 编写测试代码

首先，在你的Linux环境下（不是打开qemu后弹出的那个）编写测试代码。我们在这里命名其为 `get_ps_num.c` 。新增的系统调用可以使用 `long int syscall (long int sysno, ...)` 来触发， `sysno` 就是我们前面添加的调用号，后面跟着要传入的参数，我们这里给一个简单的测试代码样例 `get_ps_num.c` ：

```
#include<linux/unistd.h>
#include<sys/syscall.h>
int main(void)
{
    int result;
    syscall(332, &result);
    printf("process number is %d\n",result);
    return 0;
}
```

> 提示：
>
> - 这里的测试代码是用户态代码，不是内核态代码，所以可以使用`printf`。
> - 如果你想通过`syscall`函数来完成系统调用号调用系统调用的过程，请`#include<sys/syscall.h>`。

1.4.2 编译测试代码

然后使用gcc编译器编译`get_ps_num.c`。本次要用到的gcc指令原型是：`gcc [-static] [-o outfile] infile`。下面是指令的各个参数：

参数	含义
-static	静态编译，把所有用到的库函数全都打包到可执行文件里。如果不这么做的话，Linux在运行编译出来的可执行文件的时候会从系统里找库。然而我们不能保证你的qemu下的Linux 4.9.263能找到这些库，所以要静态编译。
-o outfile	指定输出的可执行文件的文件名为outfile。你可以随意制定输出的可执行文件名。我们在这里命名为get_ps_num。
infile	要编译的gcc文件名。注意，这里的文件名是gcc在终端当前路径下的相对路径。

所以，构造出的编译指令是`gcc -static -o get_ps_num get_ps_num.c`。

1.4.3 运行测试程序

然后我们需要将编译出的可执行文件`get_ps_num`放到`busybox-1.32.1/_install`下面，重新制作initramfs文件（参考实验一，只需在_install目录下执行那句find操作即可），这样我们才能在qemu中看见编译好的`get_ps_num`可执行文件。

> 注意：因为_install文件夹只有root用户才有修改权限，所以复制文件应该在命令行下使用`sudo mv`的操作进行。`mv`的使用方法可以`man mv`或百度。

编译linux4.9（直接在`linux-4.9.263`目录下运行`make -j $((`nproc`-1))`即可，不用重新make menuconfig），并运行qemu（参考实验一的1.4，我们在这里不要求使用gdb调试），在qemu中执行`./get_ps_num`，得到当前的进程数量（这个进程数量是包括`get_ps_num`程序本身的），你同样可以使用`ps -e | wc -l`获取当前进程数，`ps -e`命令输出当前运行进程的信息，一个进程一行。中间的`|`是管道操作符，在后面会讲到。`wc -l`命令则统计前面`ps -e`命令输出了多少行。这个输出的结果在考虑进去统计进程的`ps`命令对应的进程以外还会多算两行（多算了哪些？）。如果不考虑拿来统计进程数的进程本身，我们的程序结果是49，下面的命令结果也是49，可以验证我们的结果与实际的进程数是一样的（为什么是49请仔细阅读和思考本段）。

```
/ # ./get_ps_num
[    8.866422] [Syscall] ps_counter
process number is 50

/ # ps -e | wc -l
52
```

1.5 任务目标

- 实现一个linux4.9下的进程状态信息统计程序，参考命令`ps -e`：

```
/ # ps -e
PID     USER        TIME    COMMAND
1       0           0:00    /bin/sh
2       0           0:00    [kthreadd]
3       0           0:00    [ksoftirqd/0]
4       0           0:00    [kworker/0:0]
5       0           0:00    [kworker/0:0H]
6       0           0:00    [kworker/u2:0]
7       0           0:00    [rcu_sched]
```

- 输出的信息可以为：
  - 进程个数
  - 进程的PID
  - 进程的COMMAND（进程名）
  - 进程的TIME（运行时间）

  在linux4.9下创建适当的系统调用，要求添加的系统调用都会在被调用时打印出自己的系统调用名（如`ps_counter`），运行使用了新系统调用的测试程序，获取并输出以上四个信息中的三个即可拿到本部分实验的全部分数，具体评分规则见1.7。
- 示例输出：

```
/ # ./process_status
[    8.866422] [Syscall] ps_info
process num is 50
PID     TIME/ms COMMAND
1       1745    sh
2       13      kthreadd
3       2       ksoftirqd/0
4       2       kworker/0:0
5       3       kworker/0:0H
6       6       kworker/u2:0
7       0       rcu_sched
8       1       rcu_bh
```

1.6 任务提示

- 前面示例中`for_each_process`获取到的`task_struct`结构体保存有对应进程的很多信息，如PID、启动时间、运行时间、进程名等。
  - 对应的结构体定义在`include/linux/sched.h`中，为了使用它的成员变量，你可能要去参考一下。
  - 举个例子，结构体中：
    - `char comm[TASK_COMM_LEN]`成员变量为进程名
    - `pid_t pid`成员变量为进程的PID
    - `cputime stime,utime`定义进程的运行时间（具体含义自行了解）

    - 等等
- `task_struct` 中的成员变量要借助 `copy_to_user(void __user *to, const void *from, unsigned long n)` 函数复制到用户空间的变量中，才可以在用户空间访问内容
- 如果需要将用户空间的变量复制到内核空间，用 `copy_from_user(void *to, const void __user *from, unsigned long n)`
- **请注意！如果你的数组越了界，则会出现很多奇怪的错误（如段错误、输出其他数组的值或一段乱码（注意，烫烫烫是Visual C的特性，Linux下没有烫烫烫）、其他数组的值被改变等）。提问前请先排查是否出现了此类问题。**

1.7 任务评分规则

满分5分，包括：

- 用 `printk` 打印系统调用名并完成任意一个信息的输出，可以得3分。
- 每多输出一种信息，得1分，上限2分。

> 注：实验检查的流程为先确认运行结果，再简单解释一下实现思路。

第二部分 实现一个Linux Shell

> 注意：
>
> - 本部分的主要目的是锻炼大家如何**使用** Linux系统调用编写程序，本部分不会涉及编写一个系统调用。
> - 本部分主要是代码填空，费力的部分已由助教完成，代码量不大，大家不必恐慌。**需要大家完成的代码已经用注释 `TODO:` 标记，可以通过搜索轻松找到**，使用支持TODO高亮编辑器（如vscode装TODO highlight插件）的同学也可以通过高亮找到要添加内容的地方。

2.1 有关shell的背景知识

> 建议大家在自己的Linux系统（不是qemu）中尝试运行以下命令。busybox对一些应用的实现（如ps）有差异，所以qemu中的输出可能不同。但这不影响本次实验。

- 子命令：每行命令可能由若干个子命令组成，各子命令由";"分隔，需要按照顺序依次执行这些子命令。如：
  `ps -a; pwd; ls -a` 表示：先打印当前用户运行的进程，然后打印当前shell所在的目录，最后显示当前目录下所有文件（ls的-a参数表示显示隐藏文件）。
- '|'管道符：它可以将前面一个命令的输出传递给下一个命令，作为它的**标准输入**（对应C语言的stdin）。它的实现方法在下一节会讲。举例：
    - `grep` 命令可以筛选并高亮指定字符串。尝试在terminal下运行 `grep a`，它会等待用户输入。如果你输入一行不带a的字符串并按回车，它什么都不会输出。如果你输入一行带a的字符串，它会把该行的a高亮后输出。
    - `ps` 命令会显示当前用户运行的进程。为它添加 `-aux` 参数可以显示其他用户（包括系统）运行的进程，并显示详细信息（如CPU占用）。
    - 因为系统中进程数量太多不易查询，所以我们可以使用管道符实现进程的筛选。先打开Linux下的firefox浏览器，然后在终端中运行 `ps -aux | grep firefox` 可以显示所有进程名含firefox的进

程。

- 重定向符'>'和'>>'：它可以把前面一个命令的输出保存到文件内。如果文件不存在，则会创建文件。'>'表示覆盖文件，'>>'表示追加文件。
  - 举例：`ps -aux > ps.txt`可以把当前运行的所有进程信息输出到ps.txt。ps.txt的原有内容会被覆盖。
  - 举例：`ps -aux >> ps.txt`可以把当前运行的所有进程信息追加写到ps.txt。
- `cd`命令：我们知道，`cd`命令可以用来切换shell的当前目录。但是，请各位明白，不同于其他命令（比如ls，我们可以在/bin下面找到一个名为ls的可执行文件），`cd`命令其实是一个shell内置指令。这是因为，子进程无法修改父进程的参数。如果不使用内建命令而是fork出一个子进程并且在子进程中exec一个`cd`程序的话，因为执行结束了又回到了父shell环境，父shell的路径根本没有被改变，所以就会失败。所以同理，不仅是`cd`，改变当前shell的参数（如`source`命令、`exit`命令）基本都是由shell内建命令实现的。

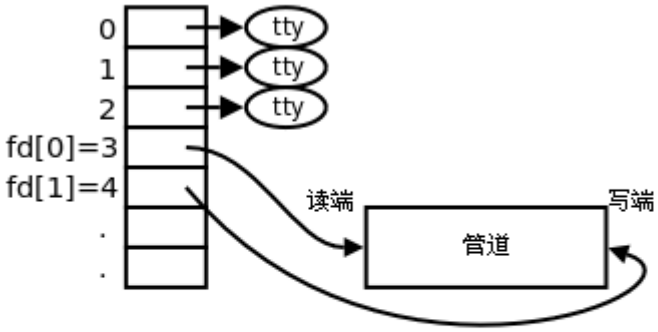
2.2 有关管道的背景知识

“一切皆文件”是Unix/Linux的基本哲学之一。普通文件、目录、I/O设备等在Unix/Linux都被当做文件来对待。虽然他们的类型不同，但是linux系统为它们提供了一套统一的操作接口，即文件的open/read/write/close等。当我们调用Linux的系统调用打开一个文件时，系统会返回一个文件描述符，每个文件描述符与一个打开的文件唯一对应。 之后我们可以通过文件描述符来对文件进行操作。管道也是一样，我们可以通过类似文件的read/write操作来对管道进行读写。为便于大家理解，本次实验使用匿名管道。匿名管道具有以下特点：

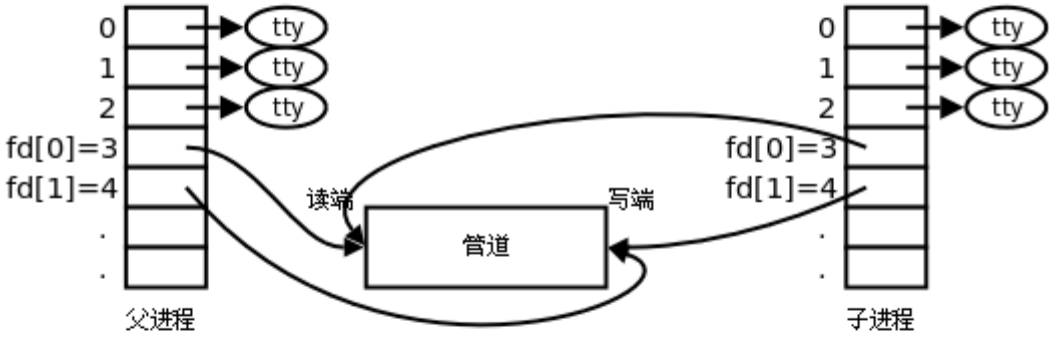
- 只能用于父子进程等有血缘关系的进程；
- 匿名管道是半双工的，在同一时刻，进程之间的通信是单向的。多个进程同时读/写会发生冲突，为了避免冲突，内核会给读写操作上锁。
- 写端不关闭，并且不写，读端读完，继续等待，此时阻塞。（就好比你的C程序在scanf，但你一直什么都不输入，程序会停住）
- 读端关闭，写端在写，那么写进程收到信号SIGPIPE，通常导致进程异常中止。

一般来说，匿名管道的使用方法是：

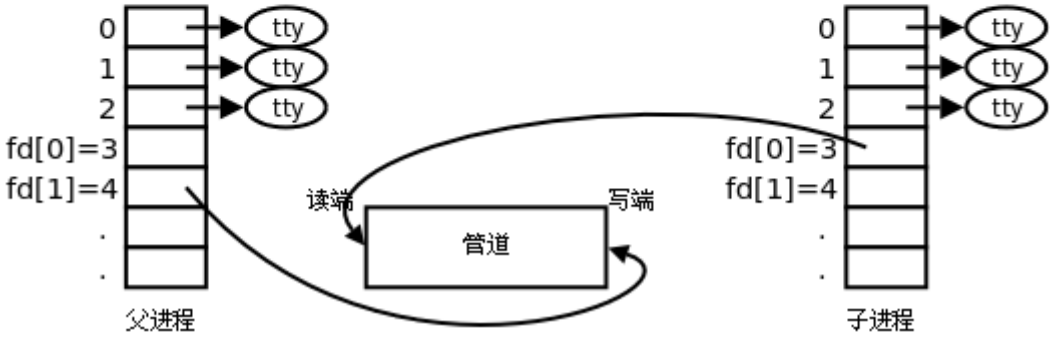
1. 父进程创建管道



2. 父进程fork出子进程



3. 父进程关闭fd[0]，子进程关闭fd[1]



- 首先，父进程调用pipe函数创建一个匿名管道。pipe的函数原型是 `int pipe(int pipefd[2])` 。我们传入一个长为2的一维数组 `pipefd` ，Linux会将 `pipefd[0]` 设为**读端**的文件描述符，并将 `pipefd[1]` 设为**写端**的文件描述符。（注：此时管道已被打开，相当于已调用了 `open` 函数打开文件）但是需要注意：此时管道的读端和写端接在了同一个进程上。如果你此时往 `pipefd[1]` 里写入数据，这些数据可以从 `pipefd[0]` 里读出来。不过这种原地tp没什么意义，我们接下来要把管道应用于进程通信。
- 然后，我们使用 `fork` 函数创建一个子进程。 `fork` 之后，数组 `pipefd` 也会被复制。此时，子进程也拥有了对管道的控制权。如果我们的目的是父进程向子进程发送数据，那么父进程就是写端，子进程就是读端。我们应该把父进程的读端关闭，把子进程的写端关闭，这样就可以使数据从父进程流向子进程。
- 因为匿名管道是单向的，所以如果想实现从子进程向父进程发送数据，就得另开一个管道。当然你也可以通过反复close/open管道来切换管道方向，但这样太麻烦了。
- 父子进程调用 `write` 函数和 `read` 函数写入、读出数据。
  - `write` 函数的原型是： `ssize_t write(int fd, const void * buf, size_t count);`
  - `read` 函数的原型是： `ssize_t read(int fd, void * buf, size_t count);`
  - 如果你要向管道里读写数据，那么这里的 `fd` 就是上面的 `pipefd[0]` 或 `pipefd[1]` 。
  - 这两个函数的使用方法在此不多赘述。如有疑问，可以百度。

> **但请注意！如果你的数组越了界，或在read/write的时候count的值比buf的大小更大，则会出现很多奇怪的错误（如段错误（Segmentation Fault）、其他数组的值被改变、输出其他数组的值或一段乱码（注意，烫烫烫是Visual C的特性，Linux下没有烫烫烫）等）。提问前请先排查是否出现了此类问题。**

2.3 输入/输出的重定向

> 注：本节描述的是如何将一个程序产生的标准输出转移到其他非标准输出的地方，不特指'>'和'>>'符号。

在使用'|', '>', '>>'时，我们需要将程序输出的内容重定向为文件输出或其他程序的输入。在本次实验中，为方便起见，我们将shell作为重定向的中转站。

- 当出现三种符号时，我们需要把前一指令的标准输出重定向为管道，让父进程（即shell）截获它的标准输出，然后由shell决定将前一指令的输出转发到下一进程、文件或标准输出（即屏幕）。
- 当出现'|'时，我们需要把后一指令的标准输入重定向为管道，让父进程（即shell）把前一进程的被截获的标准输出通过管道发给后一进程。

重定向使用`dup2`系统调用完成。其原型为：`int dup2(int oldfd, int newfd);`。该函数相当于将`newfd`标识符变成`oldfd`的一个拷贝，产生的输入/输出"转发"给`newfd`，变成`newfd`的输入/输出。如果`newfd`之前已被打开，则先将其关闭。举例：下述程序会在屏幕上输出"hello!"，并在文件输出"goodbye!"。屏幕上不会出现"goodbye"。

```
int main(void)
{
    int fd;

    fd = open("./test.txt", O_RDWR | O_CREAT | O_TRUNC, 0666);
    printf("hello!", fd);

    dup2(fd, 1);
    printf("goodbye!\n");
    return 0;
}
```

> 易混淆的地方：向文件/管道内写入数据实际是程序的一个输出过程。

2.4 一些shell命令的例子和结果分析 *

> 本部分目的在于帮助想要进一步理解真实shell行为的同学，希望我们的例子和结果分析可以为同学们带来启发。

这一节中，我们给出一些shell命令的例子，并分析命令的输出结果，让大家能够更直观的理解shell的运行。**我们并不要求实验中实现的shell行为和真实的shell完全一致**，以及本部分中测试的功能、子命令和重定向也不是实验内容，但在实现shell时，可以参考真实shell的这些行为。这些实验都是在bash下测试的，大家也可以自己在Ubuntu等的终端中重复这些实验。在这一节的代码中，每行开头为`$`的，是输入的shell命令，剩下的行是shell的输出结果。如：

```
$ cd /bin
$ pwd
/bin
```

表示在shell中了先运行了 `cd /bin` ，然后运行了 `pwd` ，第一条命令没有任何输出，第二条命令输出为 `/bin` 。 `pwd` 命令会显示shell的当前目录，我们先用 `cd` 命令进入了 `/bin` 目录，所以 `pwd` 输出 `/bin` 。

2.4.1 多个子命令 *

由 `;` 分隔的多个子命令，和在多行中依次运行这些子命令效果相同。

```
$ cd /bin ; pwd
/bin
$ echo hello ; echo my ; echo shell
hello
my
shell
```

`echo` 会将它的参数打印到标准输出，如 `echo hello world` 会输出 `hello world` 。这个实验用 `;` 分隔了三个shell命令，结果和在三行中分别运行这些命令一致。

2.4.2 管道符

这个实验展示了shell处理管道时的行为，相同的命令在管道中行为可能和单独运行时不同。

```
$ echo hello | echo my | echo shell
shell
```

如上，由于管道中的上一条命令的输出被重定向至下一条命令的输入，所以前两个 `echo` 的结果不会显示。

```
$ cd /bin ; pwd
/bin
$ cd /etc | pwd
/bin
```

如上， `cd /etc` 并没有改变shell的当前目录，这是因为**管道中的内置命令也是在新的子进程中运行的**，所以不会改变当前进程（shell）的状态
而**不包含管道的命令中，内置命令在shell父进程中运行，外部命令在子进程中运行**，所以，不包含管道的内置命令能够改变当前进程（shell）的状态。可以用 `type` 命令检查命令是否是内部命令。

```
$ type cd
cd is a shell builtin
$ type cat
cat is hashed (/bin/cat)
```

说明 `cd` 是内置命令，而 `cat` 则是调用 `/bin/cat` 的外置命令。
接下来我们测试管道中命令的运行顺序。

```
$ sleep 0.02 | sleep 0.02 | sleep 0.02 | ps | grep sleep
15840 tty1     00:00:00 sleep
$ sleep 0.02 | sleep 0.02 | sleep 0.02 | ps | grep sleep
15845 tty1     00:00:00 sleep
15846 tty1     00:00:00 sleep
15847 tty1     00:00:00 sleep
```

我们运行了两次相同的命令，却得到了不同的结果，多次运行该命令，每次输出的sleep行数不同，从0行到3行都有可能（根据电脑速度和核数，可能需要将sleep后面的数字增大或缩小来重复该实验）。这说明**管道中各个命令是并行执行的**，`ps` 命令运行时，前面的 `sleep` 命令可能执行结束，也有可能仍在执行。

2.4.3 重定向 *

```
$ cd /tmp ; mkdir test ; cd test
$ echo hello > a >> b > c
$ ls
a  b  c
$ cat a
$ cat b
$ cat c
hello
```

当一个命令中出现多个输出重定向时，虽然所有文件都会被建立，但是只有最后一个文件会真正被写入命令的输出。

```
$ cd /tmp ; mkdir test ; cd test
$ echo hello > out | grep hello
$ cat out
hello
```

当输出重定向和管道符同时使用时，命令会将结果输出到文件中，而管道中的下一个命令将接收不到任何字符。

```
$ cd /tmp ; mkdir test ; cd test
$ > out2 echo hello ; cat out2
hello
```

重定向符可以写在命令前。

2.5 总结：本次实验中shell执行命令的流程

- 第一步：打印命令提示符（类似shell ->）把管道符'|'连接的各部分分割开；
- 第二步：如果为单条命令没有管道，先检查是否是shell内置指令。是则处理内置指令后进入下一个循环；如果不是，则fork出一个子进程，然后在fork出的子进程中exec运行命令。（如果不fork直接exec，会怎么样？）
- 第三步：如果只有一个管道，创建一个管道，并将子进程1的**标准输出重定向到管道写端**，然后fork出一个子进程，在子进程中先检查是否为内置指令，是则处理内置指令，否则exec运行命令；子进程2的**标准输入重定向到管道读端**，同子进程1的运行思路。（注：2.4的样例分析中我们得出，管道的多个命令之间，虽然某个命令可能会因为等待前一个进程的输出而阻塞，但整体是没有顺序执行的，即并发执行。所以我们为了让多个内置指令可以并发，需要在fork出子进程后才执行内置指令）
- 第四步：如果有多个管道，参考第三步，n个进程创建n-1个管道，每次将子进程的**标准输出重定向到管道写端**，父进程保存**对应管道的读端**（上一个子进程向管道写入的内容），并使得下一个进程的**标准输入重定向到保存的读端**，直到最后一个进程使用标准输出将结果打印到终端。
- 第四步：打印新的命令提示符，进入下一轮循环。

2.6 任务目标

代码填空实现一个shell。这个shell不要求在qemu下运行。功能包括：

- 不要求实现子命令符';'和重定向符'>', '>>'，做了也不加分。
- 管道符'|'（支持一条命令中有单个管道符和多个管道符）。
- 实现exit, cd两个shell内置指令。
- 在shell上能显示当前所在的目录。如：{ustc}shell: [/home/ustc/exp2/] > （后面是用户的输入）
- 我们向各位提供本次实验使用的系统调用API的范围，请同学们自行查询它们的使用方法。你在实验中可能会用到它们中的一部分。
    - read/write
    - open/close
    - pipe
    - dup/dup2
    - getpid/getcwd
    - fork/vfork/clone/wait
    - mkdir/rmdir/chdir
    - exit/kill
    - shutdown/reboot
    - chmod/chown

如果你想问如何编译shell的代码，请参考1.4.2。不过我们不需要在qemu下运行自己写的shell，所以不用静态编译。如果你想问如何运行你编译出来的可执行文件，请上网搜索。

2.7 任务评分规则

满分5分，包括：

- 支持基本的单条命令运行，且命令行上能显示当前目录，可以得3分。
- 支持两条命令间的管道和内建命令（只要求cd/exit），得1分。
- 支持多条命令间的管道操作，得1分。

> 注：实验检查时会要求先确认运行结果（请**自行准备**可以验证相应功能支持的测试样例），再简单解释一下如何实现。