



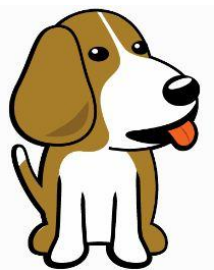
嵌入式系统开发上机手册

张笑、冯鸣夏、魏乐麒、李欢 编写

张亮 指导

西安电子科技大学嵌入式联合实验室

2014.12.20



Several experiments on BeagleBone Black

本文档包括以下内容：

- 实验准备工作
 - 安装 ARM 交叉编译工具链
 - 如何在 PC 端进入开发板目录
 - 在开发板上挂载 PC 目录
- IPC 实验
 - Unix Socket
 - FiFo Pipe
 - SystemV
 - Mmap
- 驱动实验
 - 字符驱动
 - Poll Device

每个实验具体包括：Note 和实验过程。Note 部分主要是讲解原理，实验过程主要是讲述如何在 PC 上或者板子上运行程序及其运行结果显示。

● 实验前准备工作

1. 安装 ARM 交叉编译工具链

交叉编译 (cross-compilation) 是指, 在某个主机平台上 (比如 PC 上) 用交叉编译器编译出可在其他平台上 (比如 ARM 上) 运行的代码的过程。此次实验中, 我们需要在 Ubuntu 下编译出可以在 ARM 开发板上运行的程序, 首先要在 Ubuntu 里安装交叉编译工具链。

- 进入到存放交叉编译器安装包目录下 (例如在 ~ 目录下)

```
$ cd ~
```

- 解压

```
$ tar jxvf arm-2010.09-50-arm-none-linux-gnueabi-i686-pc-linux-gnu.tar.bz2
```

- 解压成功后, 修改 PATH 环境变量:

```
$ vim /etc/profile
```

- 在该文件末尾加入交叉编译链所在的路径:

```
$ export PATH=~/.arm-2010.09/bin:$PATH
```

- 然后退出 vim。更新一下配置文件 /etc/profile, 将交叉编译器添加到环境变量中。

```
$ source /etc/profile
```

在终端输入 arm, 连续敲击 Tab 键两次, 可以看到交叉编译工具链。

```
root@lihuan-virtual-machine:/home/lihuan# source /etc/profile
root@lihuan-virtual-machine:/home/lihuan# arm
arm2hpd1      arm-none-linux-gnueabi-elfedit      arm-none-linux-gnueabi-gprof      arm-none-linux-gnueabi-size
arm-none-linux-gnueabi-addr2line    arm-none-linux-gnueabi-g++          arm-none-linux-gnueabi-ld          arm-none-linux-gnueabi-sprite
arm-none-linux-gnueabi-ar            arm-none-linux-gnueabi-gcc          arm-none-linux-gnueabi-nm          arm-none-linux-gnueabi-strings
arm-none-linux-gnueabi-as            arm-none-linux-gnueabi-gcc-4.5.1    arm-none-linux-gnueabi-objcopy     arm-none-linux-gnueabi-strip
arm-none-linux-gnueabi-c++          arm-none-linux-gnueabi-gcov          arm-none-linux-gnueabi-objdump     arm-none-linux-gnueabi-ranlib
arm-none-linux-gnueabi-c++filt       arm-none-linux-gnueabi-gdb          arm-none-linux-gnueabi-readelf
arm-none-linux-gnueabi-cpp           arm-none-linux-gnueabi-gdbtui
root@lihuan-virtual-machine:/home/lihuan# arm
```

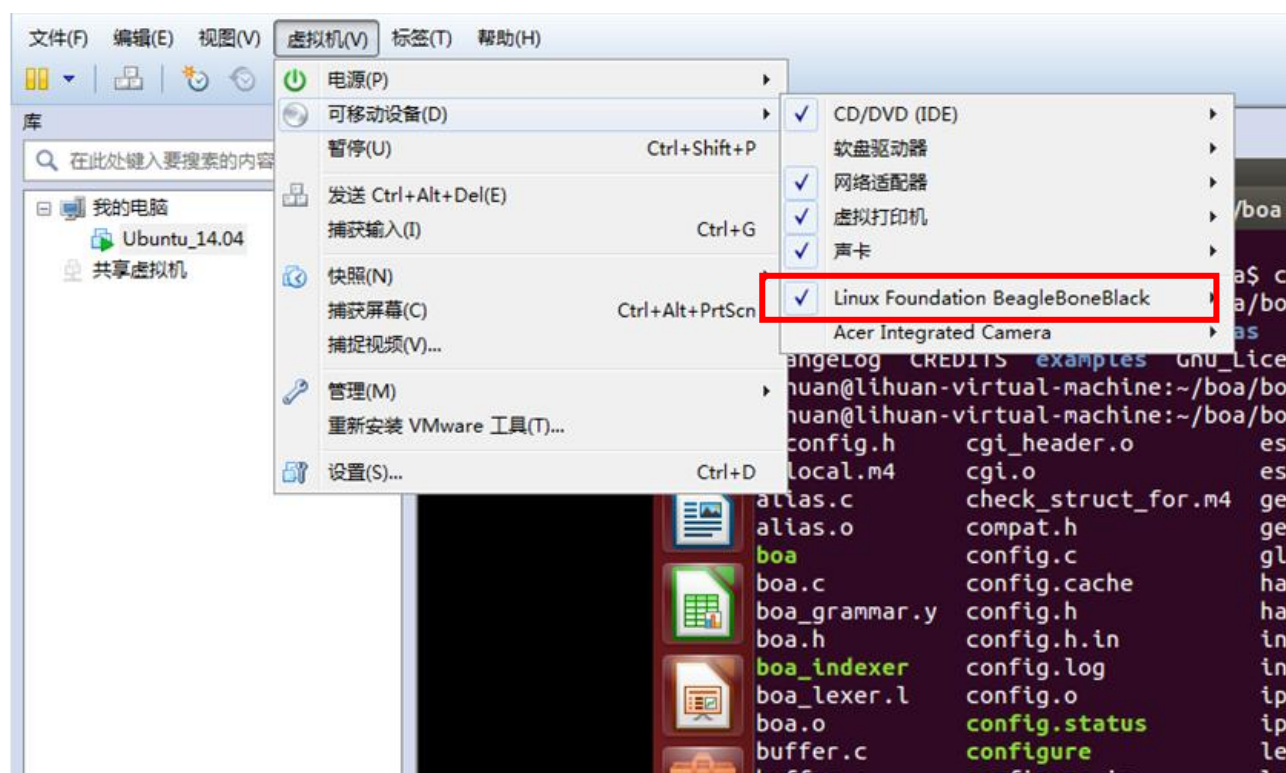
至此交叉编译器就可以使用了。



2. 如何在 PC 端进入开发板目录

这一步操作是为了我们下面, 可以方便地运行交叉编译好的代码。

开发板通过 USB 线连接到虚拟机, 如图示。



在 Ubuntu 终端里键入

```
# ssh 192.168.7.2
```

出现询问，输入 yes 继续

```
root@192.168.7.2's password:
```

键入密码，即可进入开发板目录查看

```
root@arm:~# ls
led.ko  usermode by_kernel.ko  usermode.ko
root@arm:~# exit
logout
Connection to 192.168.7.2 closed.
root@lihuan-virtual-machine:/home/lihuan# ls
```

输入 exit，即可返回虚拟机。

3. 在开发板上挂载 PC 目录

要进行代码调试运行，我们需要在开发板上挂载 PC 目录。

PC 的某个目录下放着交叉编译好的代码，可以直接在板子上运行。只要把该目录挂载在板子上的目录下，就可以直接进入板子目录运行。上面第 2 步操作可以让我们进入板子。要挂载首先需要配置 Linux 虚拟机的 NFS 服务，在 Linux 虚拟机下面执行以下命令：

- 1) 执行“`vim /etc/exports`”命令，编辑文件，添加最后面一行。如示：

```
# /etc/exports: the access control list for filesystems which may be exported
#               to NFS clients.  See exports(5).
#
# Example for NFSv2 and NFSv3:
# /srv/homes      hostname1(rw,sync,no_subtree_check) hostname2(ro,sync,no_subtree_check)
#
# Example for NFSv4:
# /srv/nfs4       gss/krb5i(rw,sync,fsid=0,crossmnt,no_subtree_check)
# /srv/nfs4/homes gss/krb5i(rw,sync,no_subtree_check)
#
#2014-12-19 add by lihuan

/home/lihuan/2012lab *(rw,sync,no_root_squash,no_subtree_check)
```

注：这个/home/lihuan/2012lab 就是 PC 上的目录，将会被挂载到板子上。

- 2) 注意上面的/home/lihuan/2012lab 和*之间都是 Tab 按键空格开，后面同样。
- 3) 保存上述配置退出
- 4) 在虚拟机的终端输入下面命令重启 NFS 服务

```
root@lihuan-virtual-machine:/home/lihuan# /etc/init.d/nfs-kernel-server restart
* Stopping NFS kernel daemon [ OK ]
* Unexporting directories for NFS kernel daemon... [ OK ]
* Exporting directories for NFS kernel daemon... [ OK ]
* Starting NFS kernel daemon [ OK ]
```

- 5) 待重启后，即可在板子的 SSH 终端上进行挂载了。具体方法为，在 PC 上进入板子目录，输入以下命令：

```
root@beaglebone:~# busybox mount -t nfs -o nolock 192.168.7.1:/home/lihuan/2012lab /mnt/
```

PC 目录

板子目录

即将/home/lihuan/2012lab 挂载到了板子的/mnt/目录。

- 6) 挂载好后，即可到板子的/mnt/目录下进行查看，如下所示：

```
root@beaglebone:~# cd /mnt/
root@beaglebone:/mnt# ls
FiFo_PiPe Mmap Poll Device SystemV Unix socket driver
```

这样，当程序编译好后，就进入该目录，直接运行，很方便。

- 7) 在 Linux 虚拟机下面进行程序编辑和编译，编辑方法用 vim 工具，编译时候用 arm-none-linux-gnueabi-gcc，即我们安装好的交叉编译链，使用方法和 gcc 相同。

例如我们将 test_xd.c 编译生成 test_xd 可执行文件。

```
bash-3.00# arm-none-linux-gnueabi-gcc test_xd.c -o test_xd
bash-3.00#
```

8) 然后在板子的终端上即可看到 test_xd 文件，直接执行即可，如下所示：

```
root@beaglebone:/mnt# ls
arm-2009q1 arm_2009q1.tar.gz hello_xd setpwd.sh test_xd test_xd.c test_xd_pc
root@beaglebone:/mnt# ./test_xd
hello,xidian university
hello,xidian university
hello,xidian university
hello,xidian university
hello,xidian university
hello,xidian university
hello,xidian university
hello,xidian university
hello,xidian university
root@beaglebone:/mnt#
```

● IPC 实验

1. Unix Socket

Note

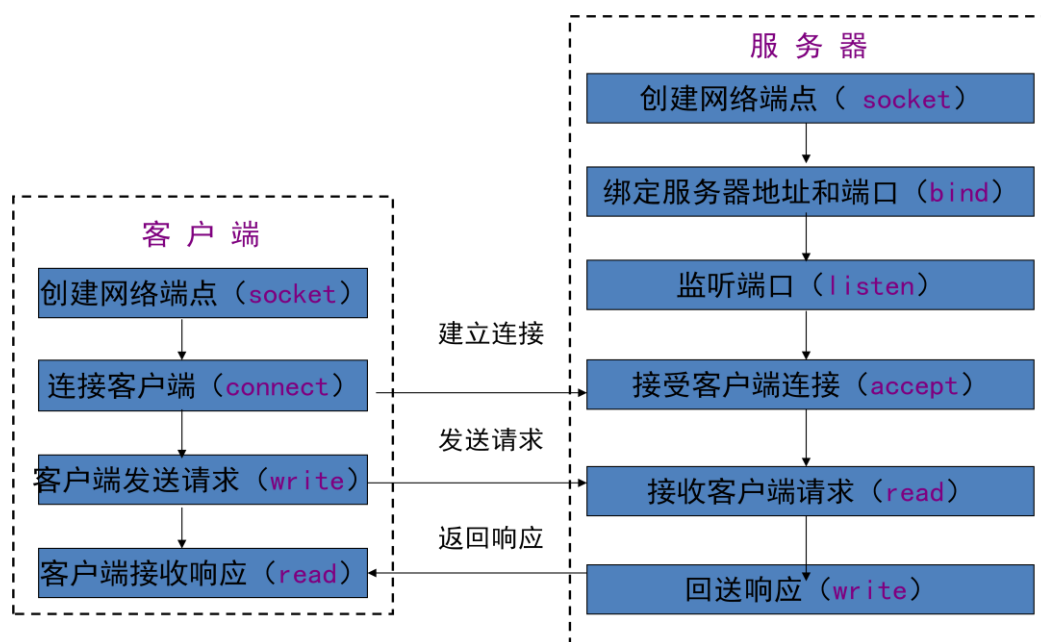


socket API 原本是为网络通讯设计的，但后来在 socket 的框架上发展出一种 IPC 机制，就是 UNIX Domain Socket。虽然网络 socket 也可用于同一台主机的进程间通讯（通过 loopback 地址 127.0.0.1），但是 UNIX Domain Socket 用于 IPC 更有效率：不需要经过网络协议栈，不需要打包拆包、计算校验和、维护序号和应答等，只是将应用层数据从一个进程拷贝到另一个进程。

使用 UNIX Domain Socket 的过程和网络 socket 十分相似。要先调用 `socket()` 创建一个 socket 文件描述符，address family 指定为 `AF_UNIX`，type 可以选择 `SOCK_DGRAM` 或 `SOCK_STREAM`，protocol 参数仍然指定为 0 即可。

UNIX Domain Socket 与网络 socket 编程最明显的不同在于**地址格式不同**，用结构体 `sockaddr_un` 表示，网络编程的 socket 地址是 IP 地址加端口号，而 **UNIX Domain Socket 的地址是一个 socket 类型的文件在文件系统中的路径**，这个 socket 文件由 `bind()` 调用创建，如果调用 `bind()` 时该文件已存在，则 `bind()` 错误返回。

下图是网络 socket 编程中一个典型的 C/S 架构图：



为了大家可以充分理解，我们对其流程详细说明：



（1）我们调用 `socket` 函数创建一个 `socket`：

```
int socket(int domain, int type, int protocol)
```

`domain`：指定 `socket` 所属的域，常用的是 `AF_UNIX` 或 `AF_INET`

`AF_UNIX` 表示以文件方式创建 `socket`，`AF_INET` 表示以端口方式创建 `socket`（我们会在后面详细讲解 `AF_INET`）

`type`：指定 `socket` 的类型，可以是 `SOCK_STREAM` 或 `SOCK_DGRAM`

`SOCK_STREAM` 表示创建一个有序的，可靠的，面向连接的 `socket`，因此如果我们要使用 TCP，就应该指定为 `SOCK_STREAM`

`SOCK_DGRAM` 表示创建一个不可靠的，无连接的 `socket`，因此如果我们要使用 UDP，就应该指定为 `SOCK_DGRAM`

`protocol`：指定 `socket` 的协议类型，我们一般指定为 0 表示由第一第二两个参数自动选择。

`socket()` 函数返回新创建的 `socket`，出错则返回 -1

（2）地址格式

常用的有两种 `socket` 域：`AF_UNIX` 或 `AF_INET`，因此就有两种地址格式：`sockaddr_un` 和 `sockaddr_in`，分别定义如下：

```
struct sockaddr_un
{
    sa_family_t sun_family; /* AF_UNIX */
    char sun_path[];        /* pathname */
}

struct sockaddr_in
{
    short int sin_family;    /* AF_INET */
    unsigned short int sin_port; /* port number */
    struct in_addr sin_addr; /* internet address */
}
```

其中 `in_addr` 正是用来描述一个 ip 地址的：

```
struct in_addr
{
    unsigned long int s_addr;
}
```


从上面的定义我们可以看出，`sun_path` 存放 `socket` 的本地文件名，`sin_addr` 存放 `socket` 的 ip 地址，`sin_port` 存放 `socket` 的端口号。

(3) 创建完一个 `socket` 后，我们需要使用 `bind` 将其绑定：

```
int bind(int socket, const struct sockaddr * address, size_t address_len)
```

如果我们使用 `AF_UNIX` 来创建 `socket`，相应的地址格式是 `sockaddr_un`，而如果我们使用 `AF_INET` 来创建 `socket`，相应的地址格式是 `sockaddr_in`，因此我们需要将其强制转换为 `sockaddr` 这一通用的地址格式类型，而 `sockaddr_un` 中的 `sun_family` 和 `sockaddr_in` 中的 `sin_family` 分别说明了它的地址格式类型，因此 `bind()` 函数就知道它的真实的地址格式。第三个参数 `address_len` 则指明了真实的地址格式的长度。

`bind()` 函数正确返回 0，出错返回 -1

(4) 接下来我们需要开始监听：

```
int listen(int socket, int backlog)
```

`backlog`：等待连接的最大个数，如果超过了这个数值，则后续的请求连接将被拒绝

`listen()` 函数正确返回 0，出错返回 -1

(5) 接受连接：

```
int accept(int socket, struct sockaddr * address, size_t * address_len)
```

同样，第二个参数也是一个通用地址格式类型，这意味着我们需要进行强制类型转化

这里需要注意的是，`address` 是一个传出参数，它保存着接受连接的客户端的地址，如果我们不需要，将 `address` 置为 `NULL` 即可。

`address_len`：我们期望的地址结构的长度，注意，这是一个传入和传出参数，传入时指定我们期望的地址结构的长度，如果多于这个值，则会被截断，而当 `accept()` 函数返回时，`address_len` 会被设置为客户端连接的地址结构的实际长度。

另外如果没有客户端连接时，`accept()` 函数会阻塞

`accept()` 函数成功时返回新创建的 `socket` 描述符，出错时返回 -1

(6) 客户端通过 `connect()` 函数与服务器连接：

```
int connect(int socket, const struct sockaddr * address, size_t address_len)
```

对于第二个参数，我们同样需要强制类型转换

`address_len` 指明了地址结构的长度

`connect()` 函数成功时返回 0，出错时返回 -1

(7) 双方都建立连接后，就可以使用常规的 `read/write` 函数来传递数据了

(8) 通信完成后，我们需要关闭 `socket`:

```
int close(int fd)
```

`close` 是一个通用函数（和 `read`, `write` 一样），不仅可以关闭文件描述符，还可以关闭 `socket` 描述符。

OK! 有了这些讲解，大家再看示例程序逻辑就很容易懂了。



“喂，我想我关心的是圣诞礼物”

实验过程

```
root@lihuan-virtual-machine:/home/lihuan/2012lab/Unix socket# arm-none-linux-gnueabi-gcc server.c -o server
root@lihuan-virtual-machine:/home/lihuan/2012lab/Unix socket# ls
client.c  server  server.c
root@lihuan-virtual-machine:/home/lihuan/2012lab/Unix socket# arm-none-linux-gnueabi-gcc client.c -o client
root@lihuan-virtual-machine:/home/lihuan/2012lab/Unix socket# ls
client  client.c  server  server.c
root@lihuan-virtual-machine:/home/lihuan/2012lab/Unix socket# ./server
bash: ./server: cannot execute binary file: 可执行文件格式错误
```

如上图示，我们使用交叉编译链编译得到两个可执行文件。试着运行，会发现提示格式错误，而它们可以在板子上运行。

先运行 `./server`,

```
root@beaglebone:/mnt/Unix socket# ls
client  client.c  server  server.c
root@beaglebone:/mnt/Unix socket# ./server
```

当运行 `srv` 程序后，该程序将处于监听状态。这时，可以通过 `netstat` 命令查看 `LISTENING`。

```
netstat -an | grep /tmp/UNIX2.domain
```

```
root@beaglebone:/mnt/Unix socket# netstat -an | grep /tmp/UNIX2.domain
unix 2      [ ACC ]     STREAM  LISTENING   6179 /tmp/UNIX2.domain
```

再运行 `./client`

```
root@beaglebone:/mnt/Unix socket# ./client
IPC通信线程
3
receive from server over
0 1 2 3 4 5 6 7 0 0 0 0 0 0 0 0 0 0 0 0
```

`server` 端收到数据并发给 `client`

```

root@beaglebone:/mnt/Unix socket# ./server

=====recv=====
0 1 2 3 4 5 6 7 0 0 0 0 0 0 0 0 0 0 0 0

=====send=====
0 1 2 3 4 5 6 7 0 0 0 0 0 0 0 0 0 0 0 0

```

2.FiFo Pipe

Note



匿名管道 (unnamed pipe): 只能用于同一个祖先的进程组通信

有名管道 (named fifo): 不相关的进程也可以使用来通信

匿名管道函数

```
int pipe(int filedes[2]);
```

filedes[0] 用于从管道中读取数据

filedes[1] 用于将数据写入管道

不需要 open, 直接 read/write 等系统调用, 系统自动删除, 进程不需要考虑。

有名管道函数

```
int mkfifo(const char *pathname, mode_t mode);
```

FIFO 必须显式删除 (调用 unlink), 但 fifo 中保存的数据是进程相关的。

我们详细来看其原理:

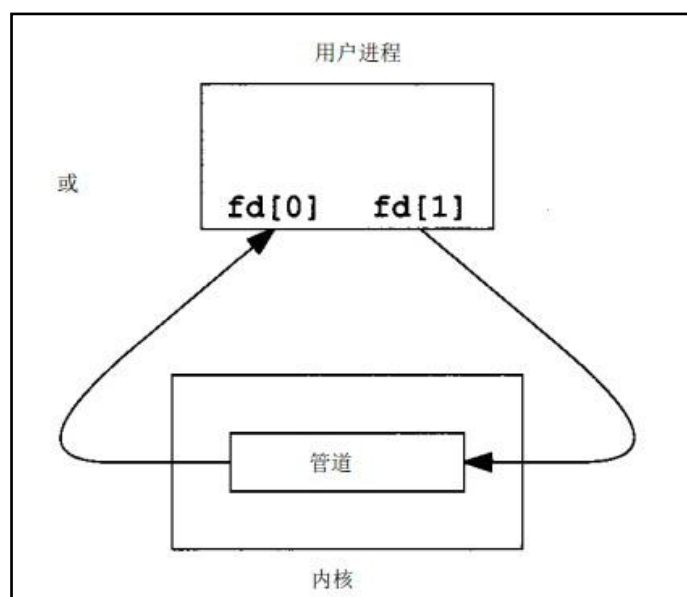
➤ **pipe** 用于具有相同祖先的进程 (最简单的情况: 父子进程) 之间的通信。

在进程中调用 pipe 函数创建管道。这个管道是由内核创建并管理的。

```
#include <unistd.h>
```

```
int pipe(int filedes[2]);
```

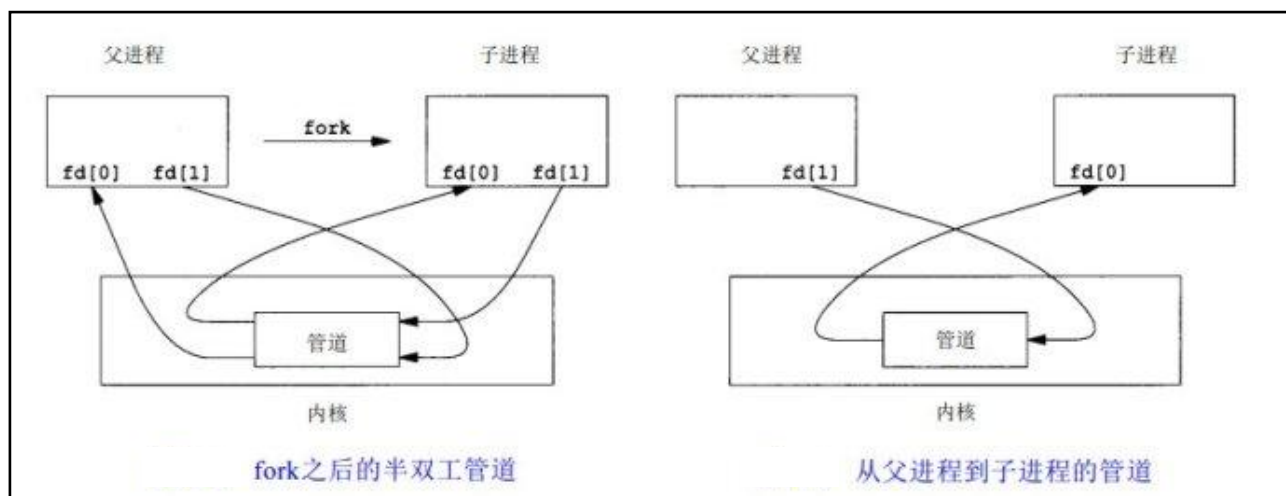
该函数返回一个文件描述符数组 fd[], fd[0] 用于读管道, fd[1] 用于写管道。如下图:



进程调用 `fork` 之后，子进程也会得到这两个文件描述符，且操作的是同一个管道。

这样就在父子进程之间建立了通信连接。

若父进程关闭写端 `f[1]`，子进程关闭读端 `f[0]`，那么就建立了一条子进程到父进程的单向管道。类似地，还可以建立一条父进程到子进程的单向管道。



如果父进程在 `fork` 之前调用了 `pipe` 两次，创建了两条管道，那么可以在之后利用上面的方法建立两条半双工管道，一条从子进程读数据，一条向子进程写数据。这样就相当于建立了父子进程之间的全双工管道。协同进程（`coprocess`）就是基于这个方法实现的。

➤ 再来看 FIFO：

FIFO 也叫作命名管道，因此两者本质上还是很相像的。

FIFO 不同于 `pipe` 的地方：

1) FIFO 可以看作高级的管道。它突破了 `pipe` 的限制（只能用于同源进程之间的通信），可以给任意进程之间建立通信连接；

2) **FIFO 是一个实际存在于磁盘中的文件**；而 pipe 是由进程创建的，依赖于进程的存活期。

可以调用 mkfifo 函数来创建一个 FIFO 文件。

```
#include <sys/stat.h>
```

```
int mkfifo(const char* pathname, mode_t mode);
```

因为它本质上是一个文件，所以**进程用 open 函数来打开一个 FIFO**，并在打开时指定文件操作模式（只读，只写还是读写）。之后**用 read (write) 函数来读（写）FIFO**。

当一个进程以只写方式打开 FIFO 文件，另一个进程以只读方式打开同一个 FIFO 文件，这样就建立了两个进程之间的通信管道。

实际工作时，pipe 和 FIFO 基本相同。下面是两者相同的性质：

- 1) 当读一个写端已经被关闭的 pipe（或者是读一个没有为写打开的进程的 FIFO）时，在所有数据都被读取后，read 返回 0，以指示到达了文件结束处
- 2) 当写一个读端已经被关闭的 pipe（或者是写一个没有为读打开的进程的 FIFO）时，write 返回-1，并将 errno 设置为 EPIPE
- 3) 在写 pipe 或者 FIFO 时，**常量 PIPE_BUF**（一般为 4096）**规定了内核中管道缓冲区的大小**。如果多个进程同时写一个管道（或者 FIFO），要**保证写的字节数不大于 PIPE_BUF**，这样多个写进程的数据不会相互穿插而造成混乱。

读写规则：

O_NONBLOCK 设置：

只读 open

- FIFO 已经被只写打开：成功返回
- FIFO 没有被只写打开：成功返回

只写 open

- FIFO 已经被只读打开：成功返回
- FIFO 没有被只读打开：返回 ENXIO 错误

从空 PIPE 或空 FIFO 中 read

- FIFO 或 PIPE 已经被只写打开：返回 EAGAIN 错误
- FIFO 或 PIPE 没有被只写打开：返回 0 (文件结束符)

write

- FIFO 或 PIPE 已经被只读打开：

写入数据量不大于 PIPE_BUF (保证原子性)：有足够空间存放则一次性全部写入，没有则返回


EAGAIN 错误 (不会部分写入)。

写入数据量大于 PIPE_BUF (不保证原子性): 有足够空间存放则全部写入, 没有则部分写入, 函数立即返回。

- FIFO 或 PIPE 没有被只读打开: 给线程产生 SIGPIPE (默认终止进程)。

PIPE 或 FIFO 若干额外的规则:

- 如果请求读取的数据量多余当前可用的数据量, 那么返回这些可用的数据。
- 如果请求写入的数据字节数小于或等于 PIPE_BUF, 那么 write 操作保证是原子的 (O_NONBLOCK 标志的设置对原子性没有影响)。
- 当对 PIPE 或 FIFO 最后一个关闭时, 仍在该 PIPE 或 FIFO 上的数据将被丢弃。

◆ 掌握了这些, 我们的示例程序就很容易读懂了。  “是真的吗”

实验过程

对于 Fifo, 一开始运行 ./fifo_read_new,

```
root@lihuan-virtual-machine:/home/lihuan/2012lab/FiFo_PiPe# ./fifo_read_new
```

```
No opened by write_only
No opened by write_only
No opened by write_only
No opened by write_only
No opened by write_only
```

运行 ./fifo_write_new,

```
root@lihuan-virtual-machine:/home/lihuan/2012lab/FiFo_PiPe# ./fifo_write_new
```

可看到 read 终端显示,

```
No data yet
No data yet
No data yet
No data yet
No data yet
No data yet
No data yet
No data yet
No data yet
```

在 write 终端输入 hello,

```
root@lihuan-virtual-machine:/home/lihuan/2012lab/FiFo_PiPe# ./fifo_write_new
hello
```


在 read 端显示,

```
No data yet
hello
No data yet
```

对于 Pipe,

```
root@lihuan-virtual-machine:/home/lihuan/2012lab/FiFo_PiPe# ./pipetest_simple
parent write over
parent close fd[1]-write over
#####
child read num is 4 , the data read from the pipe is 111
```

pipetest_second 与 pipetest_simple 同理, 自己参看。

扩展问题: 

Q:

PIPE 普遍用于 SHELL 中。根据上面的讲解, 思考 `ls | more` 背后发生了什么? 结合管道工作流程描述这个过程。

A:


step1: shell 创建一个 pipe (假设其 file descriptor 是 3 和 4, 这样方便讨论)。

step2: shell fork 出两个子进程。此时, 这两个子进程同继承了父进程的文件描述符, 也就是说, 他们同样有 3 和 4。

step3: 第一个子进程调用 `dup2(4, 1)`。这就将 1 (标准输出) 的 file object 关闭了, 并且, 1 这个文件描述符, 此时指向了 4 文件描述符指向的 file object, 即 pipe 的写端。子进程关闭 3 和 4。子进程 `execve()`, 执行 `ls` 命令。由于 `execve` 也是用的同一个文件描述符表, 所以此时 `ls` 的输出实际上是 pipe 的写端。

step4: 第二个子进程调用 `dup2(3, 0)`。如上, 0 (标准输入) 的 file object 关闭, 并且, 0 这个文件描述符, 指向了 3 这个文件描述符指向的 file object, 即 pipe 的读端。子进程关闭 3 和 4。子进程 `execve()`, 执行 `more` 命令。此时, `more` 的输入实际上是 pipe 的读端。

于是, `ls` 的输出就顺利的重定向到 `more` 的输入了。

Are you understand? 

3. SystemV

Note 

Linux 下的进程间通信方法是从 Unix 平台继承而来的。Linux 遵循 POSIX 标准 (计算机环境的可移植性操作系统界面)。进程间通信有 system V IPC 标准、POSIX IPC 标准及基于套接口 (socket) 的进程间通信机制。前两者通信进程局限在单个计算机内, 后者则在单个计算机及不同计算机上都可以通信。

System V IPC 包括 System V 消息队列、System V 信号灯和 System V 共享内存;

Posix IPC 包括 Posix 消息队列、Posix 信号灯和 Posix 共享内存区。Linux 支持所有 System V IPC 和 Posix IPC, 并分别为它们提供了系统调用。其中, System V IPC 在内核以统一的数据结构方式实现, Posix IPC 一般以文件系统的机制实现。

System V IPC functions

mechanism	function	meaning
message queues	msgget	create or access
	msgctl	control
	msgsnd	send message
	msgrcv	recieve message
semaphores	semget	create or access
	semctl	control
	semop	execute operation (wait or signal)
shared memory	shmget	create or access
	shmctl	control
	shmat	attach memory to process
	shmdt	detach memory to process

key_t 键和 ftok 函数

三种类型的 IPC 使用 key_t 值作为他们的名字，头文件<sys/types.h>把 key_t 定义为一个整数，通常是一个至少 32 位的整数，由 ftok 函数赋予的。函数 ftok 把一个已存的路径和一个整数标识符转换成一个 key_t 值，称为 IPC 键。函数原型如下：

```
#include <sys/types.h>
```

```
#include <sys/ipc.h>
```

```
key_t ftok(const char *pathname, int proj_id); //成功返回 IPC 键，出错返回-1
```

对于共享内存来说，主要有以下几个 API：shmget()、shmat()、shmdt() 及 shmctl()。

```
#include <sys/ipc.h>
```


```
#include <sys/shm.h>
```

shmget() 用来获得共享内存区域的 ID，如果不存在指定的共享区域就创建相应的区域。

shmat() 把共享内存区域映射到调用进程的地址空间中去，这样，进程就可以方便地对共享区域进行访问操作。

shmdt() 调用用来解除进程对共享内存区域的映射。

shmctl 实现对共享内存区域的控制操作。

◆ 有了这些基础，理解我们的示例程序就很简单了。 “我想也是”

PROT_WRITE //页可以被写入

PROT_NONE //页不可访问

flags: 指定映射对象的类型, 映射选项和映射页是否可以共享。它的值可以是一个或者多个以下位的组合体。实验只需使用 MAP_SHARED。

MAP_SHARED //与其它所有映射这个对象的进程共享映射空间。对共享区的写入, 相当于输出到文件。直到 msync() 或者 munmap() 被调用, 文件实际上不会被更新。

fd: 有效的文件描述词。如果 MAP_ANONYMOUS 被设定, 为了兼容问题, 其值应为-1。

offset: 被映射对象内容的起点 (一般为 0)。

实验过程

需要在板子和 Ubuntu 下测试, 根据需要修改 Makefile 文件中的编译器。在 Mmap 目录下 make, 两个二进制文件分别执行 mmap_write、mmap_read。

● 驱动实验

1. 字符驱动测试

● module_param() 测试

在用户态下编程可以通过 main() 来传递命令行参数, 而编写一个内核模块则可通过 module_param() 来传递命令行参数。

进入 driver/目录下

```
#make
```

生成 globalvar.ko 文件

```
root@ubuntu:/home/zhangxiao/embeddedSystem/example/driver# ls
globalvar_array.c globalvar.c globalvar.ko globalvar.mod.c globalvar.mod.o globalvar.o Makefile modules.order Module.symvers test test.c
```

命令行输入

```
#insmod globalvar.ko test_var=0x123
```

```
root@ubuntu:/home/zhangxiao/embeddedSystem/example/driver# insmod globalvar.ko test_var=0x123
```

命令行输入

```
#dmesg | tail -5
```

查看打印信息

```
root@ubuntu:/home/zhangxiao/embeddedSystem/example/driver# dmesg | tail -5
[21477.511235] globalvar_exit called.
[21536.452664] globalvar init succeed
[21536.452671] globalvar kcalloc succeed.
[21536.452677] globalvar cdev_add succeed.
[21536.452680] test_var = 0x123.
```

● 字符驱动测试

主要测试对字符驱动文件的 open、release、read、write、ioctl 操作。

加载字符驱动后使用

```
#cat /proc/devices
```

查看字符设备的主设备号。

```
root@ubuntu:/home/zhangxiao/embeddedSystem/example/driver# cat /proc/devices
Character devices:
 1 mem
 4 /dev/vc/0
 4 tty
 4 ttyS
 5 /dev/tty
 5 /dev/console
 5 /dev/ptmx
 6 lp
 7 vcs
10 misc
13 input
14 sound
21 sg
29 fb
99 ppdev
108 ppp
116 alsa
128 ptm
136 pts
180 usb
189 usb_device
216 rfcomm
250 globalvar
251 hidraw
252 usbmon
```

由上一步得出的打印信息得出字符设备的主设备号为 250，下面开始创建设备节点：

```
#mknod /dev/globalvar c 250 0
```

```
root@ubuntu:/home/zhangxiao/embeddedSystem/example/driver# mknod /dev/globalvar c 250 0
```

编译并运行测试文件

```
#gcc -o test test.c
```

```
#./test
```

```
root@ubuntu:/home/zhangxiao/embeddedSystem/example/driver# gcc -o test test.c
root@ubuntu:/home/zhangxiao/embeddedSystem/example/driver# ls
globalvar_array.c globalvar.c globalvar.ko globalvar.mod.c globalvar.mod.o globalvar.o Makefile modules.order Module.symvers test test.c
root@ubuntu:/home/zhangxiao/embeddedSystem/example/driver# ./test
readnum var is 0x253
```

打开 test.c 源文件，反注释掉 ioctl 的部分，保存退出重复上述步骤后，在终端输入

```
#dmesg | tail -5
```

查看打印信息。

```

root@ubuntu:/home/zhangxiao/embeddedSystem/example/driver# dmesg | tail -5
[23362.937777] globalvar open called
[23362.937788] globalvar write called
[23362.937794] globalvar read called, global_var = 0x253.
[23362.937871] cmd = 0x11
[23362.937876] globalvar release called

```

2. POLL Device

过程:

测试 char_dev.c 和 hello_param.c 的编译, 应用程序调用。

测试 char_dev.c 和 char_dev_new.c 两个一起进行 poll 调用, 按照 test.c 文件执行。

创建完成后, 输入

```
#make
```

会在 devices 目录下生成 .ko 文件 (模块文件)

加载 char_dev 模块键入

```
#insmod char_dev.ko
```

加载完成后, 输入

```
#dmesg | tail -12
```

 (输出最后 12 行) 查看加载信息。

```

[ 600.827130] My device register success !, major = 250 name = char_dev
root@ubuntu:/opt/test/devices#

```

加载带参数 hello_param 模块

键入

```
insmod hello_param.ko int_var=5 str_var="hello param!"
```

加载 hello_param 模块

加载完成后, 输入 dmesg | tail -12 (输出最后 12 行) 查看加载信息

```

[ 686.895293] Hello, param module.
[ 686.895319] int_var 5.
[ 686.895337] str_var hello_param.
root@ubuntu:/opt/test/devices#

```

为模块创建对应的虚拟设备

```
mknod /dev/char_dev c 250 0
```

```
mknod /dev/char_dev_new c 249 0
```

```

249 char_dev_new
250 char_dev

```

在虚拟机下用 gcc 编译 test.c 文件, 运行, 可以看到 poll 调用两个设备的信息

```
root@ubuntu:/opt/test/devices# gcc test.c -o test
root@ubuntu:/opt/test/devices# ./test
fd = 3.
fd_new = 4.
write success.
I am xidian
read success.
I am xidian
timeout
The End
root@ubuntu:/opt/test/devices#
```