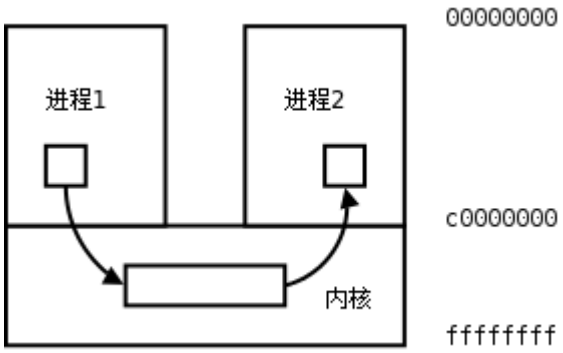
IPC 方法

Linux 环境下，进程地址空间相互独立，每个进程各自有不同的用户地址空间。任何一个进程的全局变量在另一个进程中都看不到，所以进程和进程之间不能相互访问，要交换数据必须通过内核，在内核中开辟一块缓冲区，进程 1 把数据从用户空间拷到内核缓冲区，进程 2 再从内核缓冲区把数据读走，内核提供的这种机制称为进程间通信（IPC，InterProcess Communication）。



在进程间完成数据传递需要借助操作系统提供特殊的方法，如：文件、管道、信号、共享内存、消息队列、套接字、命名管道等。随着计算机的蓬勃发展，一些方法由于自身设计缺陷被淘汰或者弃用。现今常用的进程间通信方式有：

① 管道 (使用最简单)

② 信号 (开销最小)

③ 共享映射区 (无血缘关系)

④ 本地套接字 (最稳定)

管道

管道的概念：

管道是一种最基本的 IPC 机制，作用于有血缘关系的进程之间，完成数据传递。调用 pipe 系统函数即可创建一个管道。有如下特质：

1. 其本质是一个伪文件(实为内核缓冲区)

2. 由两个文件描述符引用，一个表示读端，一个表示写端。

3. 规定数据从管道的写端流入管道，从读端流出。

管道的原理: 管道实为内核使用环形队列机制，借助内核缓冲区(4k)实现。

管道的局限性：

① 数据不能进程自己写，自己读。

② 管道中数据不可反复读取。一旦读走，管道中不再存在。

③ 采用半双工通信方式，数据只能在单方向上流动。

④ 只能在有公共祖先的进程间使用管道。

常见的通信方式有，单工通信、半双工通信、全双工通信。

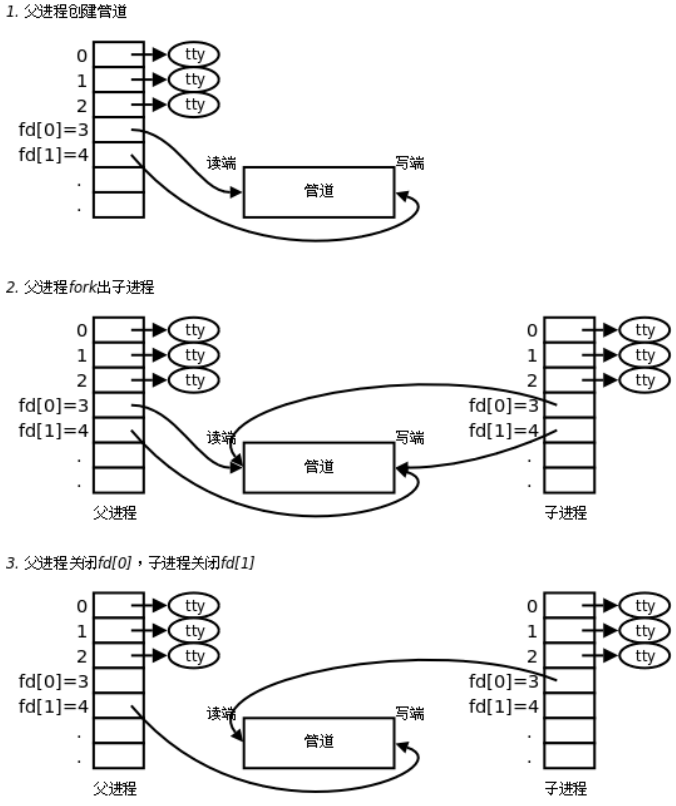
pipe 函数

创建管道

int pipe(int pipefd[2]);　　成功：0；失败：-1，设置 errno

函数调用成功返回 r/w 两个文件描述符。无需 open，但需手动 close。规定：fd[0] → r； fd[1] → w，就像 0 对应标准输入，1 对应标准输出一样。向管道文件读写数据其实是在读写内核缓冲区。

管道创建成功以后，创建该管道的进程（父进程）同时掌握着管道的读端和写端。如何实现父子进程间通信呢？通常可以采用如下步骤：



1. 父进程调用 pipe 函数创建管道，得到两个文件描述符 fd[0]、fd[1]指向管道的读端和写端。

2. 父进程调用 fork 创建子进程，那么子进程也有两个文件描述符指向同一管道。

3. 父进程关闭管道读端，子进程关闭管道写端。父进程可以向管道中写入数据，子进程将管道中的数据读出。由于管道是利用环形队列实现的，数据从写端流入管道，从读端流出，这样就实现了进程间通信。

练习：父子进程使用管道通信，父写入字符串，子进程读出并，打印到屏幕。

【pipe.c】

思考：为甚么，程序中没有使用 sleep 函数，但依然能保证子进程运行时一定会读到数据呢？

管道的读写行为

使用管道需要注意以下 4 种特殊情况（假设都是阻塞 I/O 操作，没有设置 O_NONBLOCK 标志）：

1. 如果所有指向管道写端的文件描述符都关闭了（管道写端引用计数为 0），而仍然有进程从管道的读端读数据，那么管道中剩余的数据都被读取后，再次 read 会返回 0，就像读到文件末尾一样。

2. 如果有指向管道写端的文件描述符没关闭（管道写端引用计数大于 0），而持有管道写端的进程也没有向管道中写数据，这时有进程从管道读端读数据，那么管道中剩余的数据都被读取后，再次 read 会阻塞，直到管道中有数据可读了才读取数据并返回。

3. 如果所有指向管道读端的文件描述符都关闭了（管道读端引用计数为 0），这时有进程向管道的写端 write，那么该进程会收到信号 SIGPIPE，通常会导致进程异常终止。当然也可以对 SIGPIPE 信号实施捕捉，不终止进程。具体方法信号章节详细介绍。

4. 如果有指向管道读端的文件描述符没关闭（管道读端引用计数大于 0），而持有管道读端的进程也没有从管道中读数据，这时有进程向管道写端写数据，那么在管道被写满时再次 write 会阻塞，直到管道中有空位置了才写入数据并返回。

总结：

① 读管道： 1. 管道中有数据，read 返回实际读到的字节数。

2. 管道中无数据：

(1) 管道写端被全部关闭，read 返回 0 (好像读到文件结尾)

(2) 写端没有全部被关闭，read 阻塞等待(不久的将来可能有数据递达，此时会让出 cpu)

② 写管道： 1. 管道读端全部被关闭， 进程异常终止(也可使用捕捉 SIGPIPE 信号，使进程不终止)

2. 管道读端没有全部关闭：

(1) 管道已满，write 阻塞。

(2) 管道未满，write 将数据写入，并返回实际写入的字节数。

练习：使用管道实现父子进程间通信，完成：ls | wc –l。假定父进程实现 ls，子进程实现 wc。

ls 命令正常会将结果集写出到 stdout，但现在会写入管道的写端；wc –l 正常应该从 stdin 读取数据，但此时会从管道的读端读。

【pipe1.c】

程序执行，发现程序执行结束，shell 还在阻塞等待用户输入。这是因为，shell → fork → ./pipe1， 程序 pipe1 的子进程将 stdin 重定向给管道，父进程执行的 ls 会将结果集通过管道写给子进程。若父进程在子进程打印 wc 的结果到屏幕之前被 shell 调用 wait 回收，shell 就会先输出$提示符。

练习：使用管道实现兄弟进程间通信。 兄：ls 弟： wc -l 父：等待回收子进程。

要求，使用“循环创建 N 个子进程”模型创建兄弟进程，使用循环因子 i 标示。注意管道读写行为。

【pipe2.c】

测试：是否允许，一个 pipe 有一个写端，多个读端呢？

是否允许有一个读端多个写端呢？ 【pipe3.c】

课后作业: 统计当前系统中进程 ID 大于 10000 的进程个数。

管道缓冲区大小

可以使用 ulimit –a 命令来查看当前系统中创建管道文件所对应的内核缓冲区大小。通常为：

pipe size (512 bytes, -p) 8

也可以使用 fpathconf 函数，借助参数 选项来查看。使用该宏应引入头文件<unistd.h>

long fpathconf(int fd, int name); 成功：返回管道的大小 失败：-1，设置 errno

管道的优劣

优点：简单，相比信号，套接字实现进程间通信，简单很多。

缺点：1. 只能单向通信，双向通信需建立两个管道。

2. 只能用于父子、兄弟进程(有共同祖先)间通信。该问题后来使用 fifo 有名管道解决。

FIFO

FIFO 常被称为命名管道，以区分管道(pipe)。管道(pipe)只能用于“有血缘关系”的进程间。但通过 FIFO，不相关的进程也能交换数据。

FIFO 是 Linux 基础文件类型中的一种。但，FIFO 文件在磁盘上没有数据块，仅仅用来标识内核中一条通道。各进程可以打开这个文件进行 read/write，实际上是在读写内核通道，这样就实现了进程间通信。

创建方式：

1. 命令：mkfifo 管道名

2. 库函数：int mkfifo(const char *pathname, mode_t mode); 成功：0； 失败：-1

一旦使用 mkfifo 创建了一个 FIFO，就可以使用 open 打开它，常见的文件 I/O 函数都可用于 fifo。如：close、read、write、unlink 等。

【fifo_w.c/fifo_r.c】

共享存储映射

文件进程间通信

使用文件也可以完成 IPC，理论依据是，fork 后，父子进程共享文件描述符。也就共享打开的文件。

练习：编程测试，父子进程共享打开的文件。借助文件进行进程间通信。 【fork_shared_fd.c】

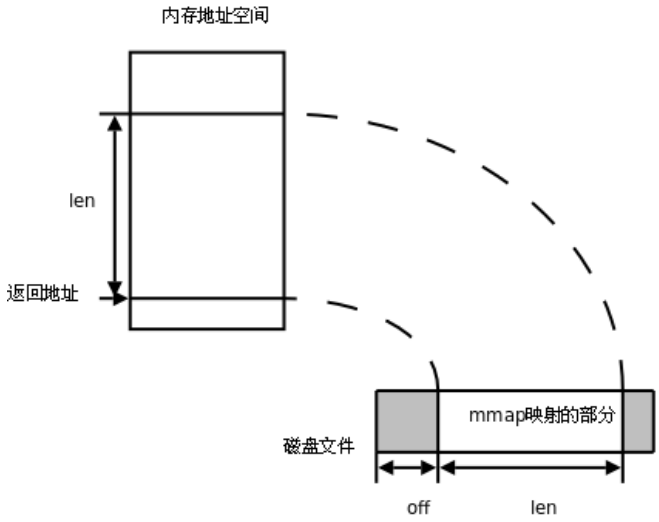
思考，无血缘关系的进程可以打开同一个文件进行通信吗？为什么？

存储映射 I/O

存储映射 I/O (Memory-mapped I/O) 使一个磁盘文件与存储空间中的一个缓冲区相映射。于是当从缓冲区中取数据，就相当于读文件中的相应字节。于此类似，将数据存入缓冲区，则相应的字节就自动写入文件。这样，就可

在不适用 read 和 write 函数的情况下，使用地址（指针）完成 I/O 操作。

使用这种方法，首先应通知内核，将一个指定文件映射到存储区域中。这个映射工作可以通过 mmap 函数来实现。



mmap 函数

void *mmap(void *adrr, size_t length, int prot, int flags, int fd, off_t offset);

返回：成功：返回创建的映射区首地址；**失败：MAP_FAILED 宏**

参数：

addr: 建立映射区的首地址，由 Linux 内核指定。使用时，直接传递 NULL

length： 欲创建映射区的大小（<=文件的实际大小）

prot： 映射区权限 PROT_READ、PROT_WRITE、PROT_READ|PROT_WRITE

flags： 标志位参数(常用于设定更新物理区域、设置共享、创建匿名映射区)

MAP_SHARED: 会将映射区所做的操作反映到物理设备（磁盘）上。

MAP_PRIVATE: 映射区所做的修改不会反映到物理设备。

fd： 用来建立映射区的文件描述符

offset： 映射文件的偏移(4k 的整数倍)，默认0，表示映射文件全部

munmap 函数

同 malloc 函数申请内存空间类似的，mmap 建立的映射区在使用结束后也应调用类似 free 的函数来释放。

int munmap(void *addr, size_t length); 成功：0； 失败：-1

借鉴 malloc 和 free 函数原型，尝试装自定义函数 smalloc，sfree 来完成映射区的建立和释放。思考函数接口该如何设计？ 【smalloc.c】

mmap 注意事项

mmap函数的保险调用方式：
1、fd = open("文件名", O_RDWR);
2、mmap(NULL,有效文件大小,PROT_READ|PROT_WRITE,MAP_SHARED,fd,0)

【mmap.c】

思考：

1. 可以 open 的时候 O_CREAT 一个新文件来创建映射区吗?
2. 如果 open 时 O_RDONLY，mmap 时 PROT 参数指定 PROT_READ|PROT_WRITE 会怎样？
3. 文件描述符先关闭，对 mmap 映射有没有影响？
4. 如果文件偏移量为 1000 会怎样？
5. 对 mem 越界操作会怎样？
6. 如果 mem++，munmap 可否成功？
7. mmap 什么情况下会调用失败？
8. 如果不检测 mmap 的返回值，会怎样？

总结：使用 mmap 时务必注意以下事项：

1. 创建映射区的过程中，隐含着一次对映射文件的读操作。
2. 当 MAP_SHARED 时，要求：映射区的权限应 <= 文件打开的权限(出于对映射区的保护)。而 MAP_PRIVATE 则无所谓，因为 mmap 中的权限是对内存的限制。
3. 映射区的释放与文件关闭无关。只要映射建立成功，文件可以立即关闭。
4. 特别注意，当映射文件大小为 0 时，不能创建映射区。所以：用于映射的文件必须要有实际大小!! mmap 使用时常常会出现总线错误，通常是由于共享文件存储空间大小引起的。如，400 字节大小的文件，在建立映射区时 offset 4096 字节，则会报出总线错。
5. munmap 传入的地址一定是 mmap 的返回地址。坚决杜绝指针++操作。
6. 如果文件偏移量必须为 4K 的整数倍
7. mmap 创建映射区出错概率非常高，一定要检查返回值，确保映射区建立成功再进行后续操作。

mmap 父子进程通信

父子等有血缘关系的进程之间也可以通过 mmap 建立的映射区来完成数据通信。但相应的要在创建映射区的时候指定对应的标志位参数 flags：

MAP_PRIVATE: (私有映射) 父子进程各自独占映射区；

MAP_SHARED: (共享映射) 父子进程共享映射区；

练习：父进程创建映射区，然后 fork 子进程，子进程修改映射区内容，而后，父进程读取映射区内容，查验是否共享。

【fork_mmap.c】

结论：**父子进程共享：1. 打开的文件 2. mmap 建立的映射区(但必须要使用 MAP_SHARED)**

mmap 无血缘关系进程间通信

实质上 mmap 是内核借助文件帮我们创建了一个映射区，多个进程之间利用该映射区完成数据传递。由于内核空间多进程共享，因此无血缘关系的进程间也可以使用 mmap 来完成通信。只要设置相应的标志位参数 flags 即可。

若想实现共享，当然应该使用 MAP_SHARED 了。

值得注意的是：MAP_ANON 和 /dev/zero 都不能应用于非血缘关系进程间通信。只能用于亲子进程间。

【mmp_w.c/mmp_r.c】

匿名映射

通过使用我们发现，使用映射区来完成文件读写操作十分方便，父子进程间通信也较容易。但缺陷是，每次创建映射区一定要依赖一个文件才能实现。通常为了建立映射区要 open 一个 temp 文件，创建好了再 unlink、close 掉，比较麻烦。 可以直接使用匿名映射来代替。其实 Linux 系统给我们提供了创建匿名映射区的方法，无需依赖一个文件即可创建映射区。同样需要借助标志位参数 flags 来指定。

使用 MAP_ANONYMOUS (或 MAP_ANON)， 如:

int *p = mmap(NULL, 4, PROT_READ|PROT_WRITE, MAP_SHARED|MAP_ANONYMOUS, -1, 0);

"4"随意举例，该位置表大小，可依实际需要填写。

【fork_map_anon_linux.c】

需注意的是，MAP_ANONYMOUS 和 MAP_ANON 这两个宏是 Linux 操作系统特有的宏。在类 Unix 系统中如无该宏定义，可使用如下两步来完成匿名映射区的建立。

① fd = open("/dev/zero", O_RDWR);

② p = mmap(NULL, size, PROT_READ|PROT_WRITE, MMAP_SHARED, fd, 0);

【fork_map_anon.c】