Linux 多线程总结

1 进程环境

C 程序总是从 main 函数开始执行。main 函数的原型是：

```
int main(int argc, char* argv[]);
```

当内核执行 C 程序时（使用一个 exec 函数），在调用 main 前先调用一个特殊的启动例程。启动例程从内核取得命令行参数和环境变量值，然后调用 main 函数。

1.1 进程终止

有 8 种方式能种子进程，其中 5 种是正常终止，3 种是异常终止。

正常：

- 从 main 返回；
- 调用 exit；
- 调用_exit 或_Exit；
- 最后一个线程从其启动例程返回；
- 从最后一个线程调用 pthread_exit。

异常：

- 调用 abort；
- 接到一个信号；
- 最后一个线程对取消请求做出响应。

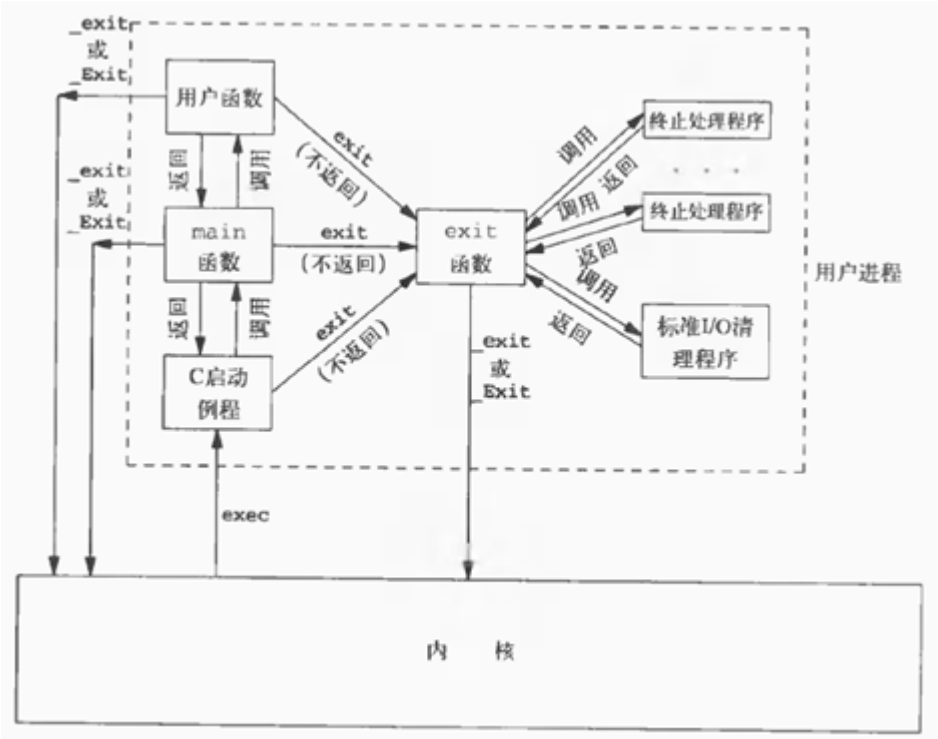
其中三个正常终止函数：_exit、_Exit 和 exit 之间是有区别的。_exit、_Exit 立即退出；而 exit 则先执行一些清理工作，然后再退出。

并且可以利用函数 atexit 来声明在 exit 退出前会被调用的函数(称为终止处理程序)，atexit 原型为：

```
int atexit( void (*func)(void) );
```

返回值：若成功返回 0，若出错返回非 0；

如下图所示的一个 C 程序是如何启动和终止的：



注意：内核使程序执行的**唯一**方法是调用一个 exec 函数。进程**自愿**终止的**唯一**方法是显示或隐式地（通过调用 exit）滴啊用_exit 或_Exit；进程也可**非自愿**地有 一个信号使其终止。

1.2 进程内存布局

每个进程所分配的内存由很多部分组成，通常称之为"段"。如下所示：

- **正文段：**包含了进程运行的程序**机器语言指令**，此段具有只读属性。
- **初始化数据段：**包含**显示**初始化的**全局变量和静态变量**。
- **为初始化数据段：**包含**未显示**初始化的**全局变量和静态变量**，系统为此段所有内存初始化为0。
- **栈：****自动变量**以及每次函数调用时所需保存的信息都存放在此段中，包括局部变量、实参、返回值、临时变量和环境信息。
- **堆：**通常在堆中进程**动态存储分配**。

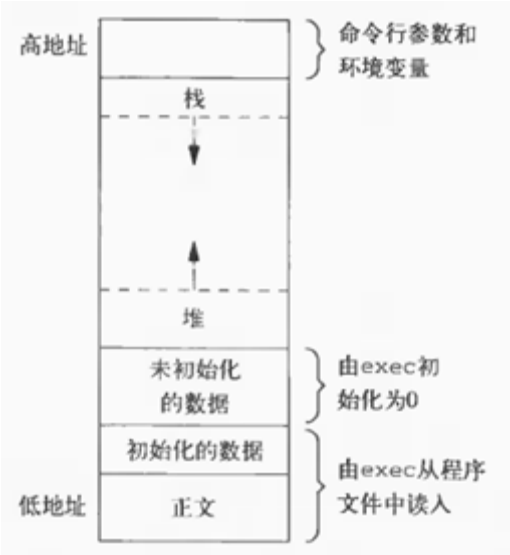


Figure 1 典型的存储空间安排

2 进程控制

每个进程都有进程 ID，其中 ID 为 0 的进程是调度进程（交换进程）；进程为 1 的是 init 进程，并且 init 进程决不会终止，

2.1 进程的创建

进程的创建可以通过函数 fork 和 vfork 进行。

重点注意：

不同的进程拥有不同的地址空间。如在不同的两个进程都拥有 100 的地址，那么这两个 100 存放的值是不一样的。

2.1.1 fork 函数

pid_t fork(void)

返回值：子进程返回 0，父进程返回子进程的 ID；若出错返回-1；

fork 函数被调用一次，却返回两次，子进程返回 0，父进程返回子进程的 ID 值；并且子进程的内存是父进程的完全副本，包括局部、全局、静态的数据空间、堆和栈的副本。其中有些系统对 fork 进行了优化，即写时复制（Copy-On-Write）技术，在 fork 调用之后父子进程共享同一个区域，只有当父进程或子进程的任意一方试图修改这些区域，内核才会为修改区域的那块内存制作一个副本。

1) 缓冲问题

在 fork 函数调用之前，需要考虑缓冲中是否有数据存在。是为不带缓冲的系统调用，还是带缓冲的标准 I/O。

- 如果在调用 fork 函数之前，调用了不带缓冲的系统函数（如 write），则只输出一次之前的数据；
- 如果在调用 fork 之前，调用标准 I/O（带缓冲函数，如 printf），则可能输出会输出多次，因为父子进程都保存了缓冲中的数据，如果此时是与终端交互，则是行缓冲，而如果是磁盘交互，则是全缓冲。

2) 文件共享

在 fork 调用之后，子进程复制了父进程文件描述符，并且子进程和父进程共享该文件的偏移量。若父子进程之间没有任何形式的同步，那么它们的输出就会相互混合。在 fork 之后处理文件描述符有以下两种使用模式：

- 父进程等待子进程完成：父进程无需对其描述符做任何处理。当子进程终止后，它曾进行过读、写操作的任一共享描述符的偏移量已做了相应更新。
- 父进程和子进程各自执行不同的程序段：在 fork 之后，父进程和子进程各自关闭它们不需使用的文件描述符，这样就不会干扰对方使用的文件描述符。这种方法是网络服务器进程经常使用的

子进程还继承父进程的属性还有：

- 实际用户 ID、控制终端
- 设置用户 ID 标志和设置组 ID 标志
- 当前工作目录、环境、根目录、文件屏蔽字
- 对任一打开文件描述符的执行时关闭（close-on-exec）标志
- 连接的共享存储段、存储映像

3) fork 使用模式

fork 有两种用法，复制自己从而进行不同的操作，或是复制自己执行不同的程序。

- 父进程和子进程同时执行不同的代码段：如在网络服务进程中，父进程等待客户的服务请求，父进程调用 fork 使子进程处理此请求，父进程则继续等待下一个服务请求。
- 父进程和子进程要执行一个不同的程序：如在 shell 中，子进程从 fork 返回之后立即调用 exec。

2.1.2 vfork 函数

vfork 函数用于创建一个新进程，而该新进程的目的是 exec 一个新程序。所以 vfork 与 fork 一样都创建一个子进程，但是它并不将父进程的地址空间完全复制到子进程中，因为子进程会立即调用 exec（或 exit），于是也就不会引用该地址空间。vfork 保证子进程先运行，并在子进程调用 exec 或 exit 之前，它在父进程的空间中运行。

2.2 进程的终止

在 1.1 小节中，介绍了进程 5 种正常终止和 3 种异常终止的方式。其中不管进程如何终止，最后都会执行内核中的同一段代码，来为相应进程**关闭**所有**打开描述符**，及释放它所使用的存储器等。

子进程的通过如下方式来通知父进程自己是如何终止的：

- **对于 3 个终止函数（exit、_exit、_Exit），是将退出状态作为参数传送给函数。如 exit(0)。**
- **在异常终止情况下，内核（不是进程本身）产生一个指示器异常终止原因的终止状态 (termination status)。**

在任意一种情况下，该终止进程的父进程都能用 wait 或 waitpid 函数取得其终止状态。

注意：

- 一个已经终止、但其父进程尚未对其进行善后处理（获取终止子进程的有关信息，释放它仍占用的资源）的进程被称为**僵死进程**。
- 如果父进程在子进程之前终止，那么此已终止父进程的所有子进程，将会改变它们的父进程为 init 进程，称这些子进程被 **init 收养**。

2.3 监控子进程

当一个进程正常或异常终止时，内核就向其父进程发送 SIGCHLD 信号。因为子进程终止时一个异步事件(这可以在父进程运行的任何时候发生)，所以这种信号也是内核向父进程发的异步通知。其中父进程可以通过调用 wait 或 waitpid 来获得子进程的终止状态。

其中在调用上述两函数之后，可能会**发生如下的情况**：

- 如果其所有子进程都还在运行，则阻塞。
- 如果一个子进程已终止，正等待父进程获取其终止状态，则取得该子进程的终止状态立即返回。
- 如果它没有任何子进程，则立即出错返回。

```
pid_t wait(int *statloc);
pid_t waitpid(pid_t pid, int *statloc, int options);
```

两个函数返回值：若成功，返回进程 ID，statloc 返回终止状态；若出错，返回 0 或-1

这两个函数的区别是：

- 在一个子进程终止前，wait 使其调用者阻塞，而 waitpid 有一选项(ANOHANG)，可使调用者不阻塞。
- waitpid 并不等待在其调用之后第一个终止子进程，它由若干个选项，可以控制它所等待的进程。

由于 UNIX 信号一般是不排队的，所以当有多个子进程同时终止，那么 wait 和 waitpid 只能获得一个子进程的终止状态，为了防止余下的子进程得不到处理（成为僵死进程），那么只能使用 waitpid 监控了。

```
while( (pid=waitpid(-1,&stat, WNOHANG) )>0) //一直等待子进程终止，直到没有已终止子进程，则跳出循环
    printf("child %d teminated\n",pid);
```

上述代码**如果将 waitpid 改为 wait**，也可以获得所有子进程的终止状态，但父进程(当前进程)将进入**永远的阻塞状态**，因为如果没有子进程，那么 wait 将会阻塞，不能返回。

2.4 程序的执行

Linux 可以通过一系列 exec 函数（7 个）来执行另一个程序，被加载的新程序将替换某一进程的内存空间，旧进程的栈、数据以及堆段会被新程序的相应部件所替换，并且新程序会从 main()函数出开始执行。调用 exec 函数之后，进程的 ID 仍保持不变。

2.4.1 exec 函数

```
int execl(const char *pathname, const char *arg0,… );
int execv(const char *pathname, char*const argv[]);
int execle(const char *pathname, const char* arg0, …);
int execve(const char *pathname, char* const argv[], char *const envp[]);

int execlp(const char *filename, const char *arg0,…);
int execvp(const char *filename, char* const argv[]);

int fexecve(int fd, char *const argv[], char *const envp[]);
```

7 个函数返回值：若出错，返回-1，若成功，不返回

这些函数之间的第一个区别是：

前 4 个函数取路径名作为参数，后两个取文件名作为参数（**p**），最后一个取文件描述符作为参数（**f**）。

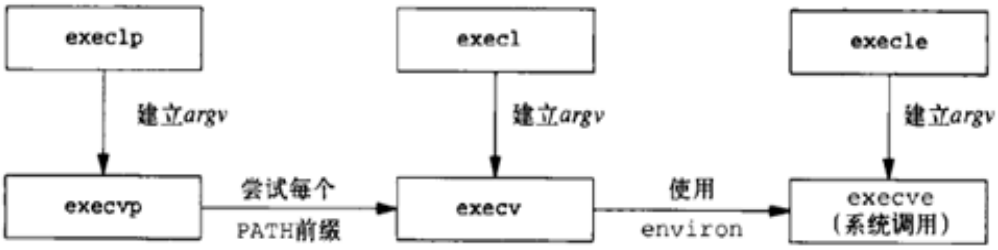
第二个区别是：

是参数表传递的不同，**l 表示列表 list**，**v 表示矢量 vector**。

第三个区别是：

是否传递**环境表 e**。

在 Linux 中只有 execve 是内核的系统调用，另外 6 个则只是库函数，6 个库函数都需调用该系统调用，如下图所示：



2.4.1 属性的继承

1) 基本属性

在执行 exec 之后，除了进程 ID 不变外，新程序还继承了如下属性：

- 进程 ID 和父进程 ID
- 实际用户 ID 和实际组 ID
- 附属组 ID、进程组 ID、会话 ID
- 控制终端
- 当前工作目录、跟目录
- 文件模式创建屏蔽字、进程信号屏蔽字
- 文件锁
- 未处理信号
- 资源限制

2) 文件描述符

其中打开文件的处理与每个描述符的**执行时关闭(close-on-exec)标志值**有关，即 FD_CLOEXEC，若对文件描述符设置了此标志，则在执行 exec 时关闭该描述符；否则该描述符仍打开。

3) 设置用户 ID 和设置组 ID

注意，在 exec 前后实际用户 ID 和实际组 ID 保持不变，但如果对 pathname 所指定的程序文件设置了 **set_user-ID**(set-group-ID)权限位，那么系统调用会在执行此文件时将进程的有效用户(组)ID 置为程序文件的属主(组)ID。

2.4.3 system 函数

程序可通过调用 system 函数来执行任意的 shell 命令。其函数原型如下：

```
#include <stdlib.h>
int system(const char* cmdstring);
```

函数 system()的实现中调用了 fork、exec 和 waitpid 等函数，其创建一个子进程来运行 shell，从而执行了命令 cmdstring。如 system("ls | less")

3 信号

信号是事件发生时对进程的一种通知方式。事件可以是硬件异常（如除以 0）、软件条件（如 alarm 定时器超时）、终端产生的信号或调用 kill 函数。

- 信号产生(generation)：是指事件发生的动作，信号产生的时间就是事件发生的时刻；
- 信号递送(delivery)：是指这样一个过程，即当信号产生时，内核会在进程表中以某种形式设置一个标志。
- 信号未决(pending)：是指在信号产生和递送之间的时间间隔内。

3.1 信号的传递

进程可以选用"阻塞信号递送"。如果为进程产生了一个阻塞的信号，而且对该信号的动作是系统默认动作或捕捉该信号，则该进程将此信号保持为未决状态，直到该进程对此信号解除了阻塞，或者将对此信号的动作更改为忽略。

内核将**信号传递**给进程有如下**几种可能**：

- 如果内核接下来要调度该进程运行，而等待信号会马上传递给进程。
- 如果进程正在运行，则会立即传递信号给进程。
- 如果所产生的信号是进程的阻塞信号之一，那么信号将保持等待状态，直至对该信号解除阻塞（从信号掩码移除）。

注意：

1. 由于信号是不进行排队处理的，所以如果在解除阻塞某种信号之前，该信号发生了多次，那么在解除之后该信号只会被递送给进程一次。
2. 由于不同信号的处理是无序的，所以在解除阻塞之前产生了多个信号，那么解除之后进程接收的顺序不一定与产生的顺序一致。

3.2 信号的处理方式

在某个信号出现时，可以告诉内核按下列 3 种方式之一进行处理：

1. 忽略此信号

   大多数可以这样处理，但有两种信号不能被忽略，是 SIGKILL 和 SIGSTOP。

2. 捕捉信号

   用户可以通知内核在某种信号发生时，调用一个用户函数。但 SIGKILL 和 SIGSTOP 不能被捕捉。

3. 执行系统默认动作

   每一种信号都有默认动作，基本都是终止该进程。

3.3 接收信号：signal 和 sigaction

signal 和 sigaction 函数都是设定接收信号的处理方式，若未对信号重新设置，则进程按默认方式处理。

3.3.1 signal

UNIX 系统信号机制最简单的接口是 signal 函数：

```
void (*signal(int signo, void (*func)(int) ) )(int);
```

返回值：若成功，返回以前的信号处理函数；若出错，返回 SIG_ERR

signo 参数是要处理的信号名，func 的值可以是常量 SIG_IGN，常量 SIG_DEL 或当接到信号后要调用的函数地址。即对应信号的三种处理方式。

- SIG_IGN：忽略此信号
- SIG_DEL：按系统默认动作
- 函数地址：在信号发生时，调用该函数（信号处理程序）；

在 exec 或 fork 后的信号处理方式;

1) 调用 exec

一个进程原先要捕捉的信号，当调用 exec 执行一个新程序后，这些被捕获信号的处理方式都被更改为**按系统默认方式处理**，因为信号捕捉函数的地址很可能在所执行的新程序文件中已无意义。

2) fork 进程

当一个进程调用 fork 时，其子进程**继承父进程的信号处理方式**。因为子进程在开始时复制了父进程内存映象，所以信号捕捉函数的地址在子进程中是有意义的。

3.3.2 sigaction

sigaction 函数的功能是检测或修改与制定信号相关联的处理动作。

```
int sigaction(int signo, const struct sigaction* restrict act, struct sigaction* restrict oact);
返回值：若成功，返回 0；若出错，返回-1

struct sigaction{
    void (*sa_hander)(int);          //处理函数，或 SIG_IGN, SIG_DEL
    sigset_t sa_mask;                //屏蔽字
    int sa_flags;                    //选项
 void (*sa_sigaction)(int , siginfo_t*, void*);  //选择项
}
```

- signo 是要检测或修改其具体动作的信号编号。
- 若 act 指针非空，则要修改其动作。
- 如果 oact 指针非空，则系统经由 oact 指针返回该信号的上一个动作。

使用方法：

当更改信号动作时，如果 sa_handler 字段包含一个信号捕捉函数的地址（不是常量 SIG_IGN 或 SIG_DFL），则 sa_mask 字段说明了一个信号集，在调用该信号捕捉函数之前，这一信号集要加到进程的信号屏蔽字中。仅当信号捕捉函数返回时再将进程的信号屏蔽字恢复为原先值。

3.4 发送信号：kill、raise 和 alarm

kill 函数将信号**发送**给指定进程或进程组；raise 函数是**立即**向进程自身发送信号；alarm 函数是设置一个定时器，当**超时后**向自身发送一个信号。

3.4.1 kill 函数

int kill(pid_t pid, int signo);

两个函数返回值：若成功，返回 0；若出错，返回-1；

kill 的 pid 参数有以下 4 种不同的情况：

- pid>0：将信号发送给进程 ID 为 pid 的进程。
- pid==0：将信号发送给与发送进程属于同一进程组的所有进程。
- pid<0：将信号发送给进程组 ID 等于 pid 绝对值。
- pid==-1：将信号发送给所有进程（有权限的进程）。

注意：将信号发生给其它进程需要权限

- 超级用户：可将信号发送给任一进程；

- 非超级用户：信号发生进程的实际用户 ID 或有效用户 ID 必须等于接收进程的实际用户 ID 或有效用户 ID。

3.4.2 raise 函数

```
int raise(int signo);
```

两个函数返回值：若成功，返回 0；若出错，返回-1；

当进程调用 raise 函数，信号立即被传递(即，在 raise 函数返回调用者之前)。并且 raise 出错的唯一原因是 signo 无效。

3.4.3 alarm 函数

```
#include <unistd.h>
unsigned int alam(unsigned int seconds)
```

返回值：0 或以前设置的闹钟时间的余留秒数

使用 alarm 函数可以设置一个定时器（闹钟时间），在将来的某个时刻该定时器会超时。当定时器超时时，会产生 SIGALRM 信号。如果忽略或不捕捉此信号，则其默认动作是终止调用该 alarm 函数的进程。其中进程调用 alam()函数后是不挂起的，仍然继续执行。

3.5 挂起进程： sleep 和 pause

sleep 和 pause 函数都能使调用进程挂起，但 sleep 在指定的时间超时或接收到信号时就能唤醒；而 pause 只能接收到信号才得以唤醒，否则将一直处于挂起状态。

3.5.1 sleep 函数

```
#include <unistd.h>
int sleep(int seconds)
```

返回值：0 或未休眠完成的秒数

此函数使调用进程被挂起知道满足以下两个条件之一：

1. 已经过了 seconds 锁指定的墙上时钟时间；
2. 调用进程捕捉到一个信号并从信号处理程序返回。

3.5.2 pause 函数

```
#include <unistd.h>
int pause(void)
```

返回值：-1，errno 设置为 EINTR

pause 函数使得调用进程自己挂起，直至当前进程接收到某种信号，才得以唤醒；否则一直处于阻塞状态。

注意：

只有执行了一个信号处理程序并从其返回时，pause 才返回。

3.6 信号屏蔽功能：sigprocmask、sigpending 和 sigsuspend

内核会为每个进程维护一个信号掩码（即一个信号集），从而进程将屏蔽信号掩码中的信号集。若将被屏蔽的信号发送给进程，那么该信号的传递将**被延后（只是被延后，不是被删除）**，直至该进程解除该信号，从而该进程仍能够接收到先前被屏蔽的信号。

关于进程的信号屏蔽，UNIX 还提供了如下功能：

- **sigprocmask**

  调用函数 sigprocmask 可以检测或更改，或同时进行检测和更改进程的**信号屏蔽字**。

- **sigpending**

  该函数返回正处于等待状态的信号集，即由于某个信号是屏蔽字段的类型之一，并且发送给了调用进程，那么将从 set 返回。

- **sigsuspend(sigset_t *sigmask)**

  该函数的功能是将屏蔽字**更改**为 sigmask，并**挂起**调用进程；直至**接收到** sigmask 之外的信号，才**唤醒**该进程，并将屏蔽字**恢复**为调用 sigsuspend 函数之前的**屏蔽字**。该过程可分为 3 个步骤完成：首先设置指定屏障字；然后使进程挂起(类似 pause 功能)；最后接收到除 sigmask 信号，并恢复之前的屏蔽字。

注意：

屏蔽信号，并不是删除信号，当解除屏蔽信号时，被阻塞的信号仍能被进程接收。

4 进程通信

4.1 管道和 FIFO

4.1.1 管道

管道有两种局限性：

- 历史上，它们是半双工的，虽然某些系统提供全双工管道。
- 管道只能在具有公共祖先的两个进程之间使用。

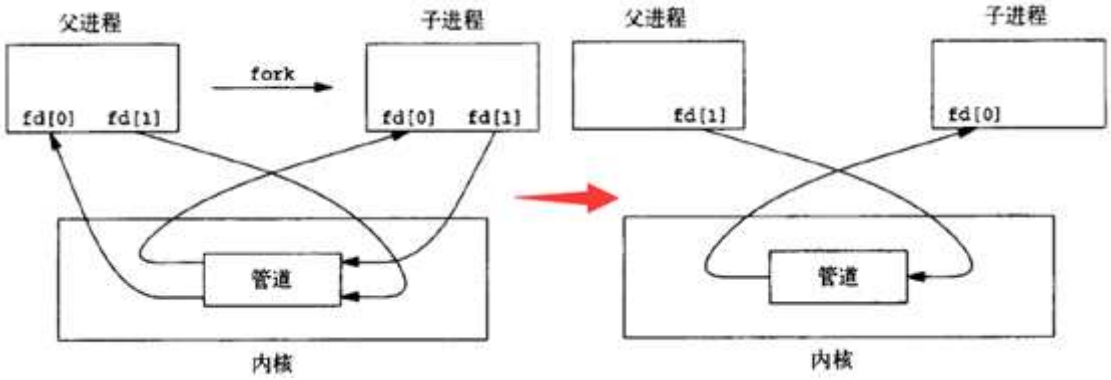
创建管道函数原型是：

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

返回值：若成功，返回 0,，若出错，返回-1；

经由参数 fd 返回两个文件描述符，fd[0]为读而打开，fd[1]为写而打开，然后就能像使用普通文件描述符一样，进行读写了。

单个进程中的观点几乎没有任何用处。通常，进程会先调用 pipe，接着调用 fork，从而创建从父进程到子进程的 IPC 通道。对于从父进程到子进程的管道，如父进程关闭管道的读端(fd[0])，子进程关闭写端(fd[1])。如下图所示：



当管道的一端被关闭后，下列两条规则起作用：

1. 当读(read)一个写端已被关闭的管道时，在所有数据都被读取后，read 返回 0，表示文件结束。
2. 如果写(write)一个读端已被关闭的管道，则产生信号 SIGPIPE。如果忽略该信号或者捕获该信号并从其处理程序返回，则 write 返回-1，errno 设置为 EPIPE。

如下程序：

```
1 main(){
2    int n,fd[2];    pid_t pid;    char line[100];
3    pipe(fd);
4    if((pid=fork()<0)
5        return;
6    else if(pid>0){
7        close(fd[0]);
8        write(fd[1],"hello world\n",12); //向管道中写入数据
9    }
10    else{
11        close(fd[1]);
```

```
12        n = read(fd[0],line,12);    //从管道中读取数据
13        write(STDOUT_FILENO, lien, n);//输出终端
14    }
15 }
```

4.1.2 FIFO

FIFO 也称为命名管道。与管道不同，通过 FIFO **不相关的进程也能交换数据**。FIFO 的使用方式是：先创建(mkfifo)，然后使用 open 打开，接着就能使用 I/O 系统调用(如 read()、write()和 close())操作打开的文件描述符了。

```
int mkfifo(const char* path, mode_t mode);
                                返回值：若成功，返回 0；若出错，返回-1；
```

- **path 参数：**表示 FIFO 文件的保存路径，即 mkfifo 创建一个名为 path 的 FIFO。
- **mode 参数：**指定了新 FIFO 的权限。

与管道一样，FIFO 也有一个写入端和读取端，并且从管道中读取数据的顺序与写入的顺序是一样的。

其中需注意：

- 打开一个 FIFO 以便**读取数据**(open() O_RDONLY 标志)将会阻塞，直到另一个进程开大 FIFO 以**写入数据**(open() O_WRONLY 标志)**为止**。
- 相应地，打开一个 FIFO 会**同步读取进程和写入进程**。如果一个 FIFO 的另一端已经打开(可能是因为一对进程已经打开了 FIFO 的两端)，那么 open()**调用会立即成功**。

4.2 消息队列

消息队列与 FIFO 类似，都是需要先创建一个标识，然后通过这个标识进行消息的发送和接收。

1) 创建或打开一个消息队列

```
int msgget(ket_t key, int msgflag);
                        返回值：若成功，返回消息队列 ID；若出错，返回-1；
```

- key 参数：是消息队列的一个标识，将这个标识作为消息队列的外部名；
- msgflag 参数：是消息队列的权限。

2) 发送消息

msgsnd 功能是将新消息添加到队列尾端。

int msgsnd(int msqid const void* ptr, size_t nbytes, int flag);
返回值：若陈工，返回 0，若出错，返回-1

3) 接收消息

msgrcv 用于从队列中取消息，可以按先进先出次序取消息，也可以按消息的类型字段取消息。

int msgrcv(int msqid, void* ptr, size_t nbytes, long type, int flag);
返回值：若成功，返回消息数据部分的长度；若出错，返回-1

4.3 信号量

信号量不是用来在进程间传输数据的，相反，它是用来同步进程的动作。信号量的一个常见用途是同步对一块共享内存的访问以防止出现一个进程在访问共享内存的同时另一个进程更新这块内存的情况。

为了获得共享资源，进程需要执行下列操作：

1. 测试控制该资源的信号量。
2. 若此信号量的值为正，则进程可以使用该资源。在这种情况下，进程会将信号量值减 1，表示它使用了一个资源单位。
3. 否则，若此信号量的值为 0，则进程进入休眠状态，直至信号量值大于 0.进程被唤醒后，它返回至步骤 1)。

使用信号量的常规步骤如下：

1. 使用 **semget()** 创建或打开一个信号量集。
2. 使用 **semctl()** SETVAL 或 SETALL 操作初始化集合中的信号量。(只有一个进程需要完成这个任务)
3. 使用 **semop()** 操作信号量值。使用信号量的进程通常会使用这些操作来表示一种共享资源的获取和释放。
4. 当所有进程都不在需要使用信号量集之后使用 **semctl()** IPC_RMID 操作删除这个集合。(只有一个进程需要完成这个任务)

4.4 共享内存

共享存储允许多个进程共享一个给定的存储区。因为数据不需要在客户进程和服务器进程之间复制，所以这是最快的一种 IPC。为了防止多个进程同时访问共享存储区，通常使用信号量同步共享存储区的访问（也可使用记录锁或互斥量）。

共享存储与内存映射不同之处在于，前者没有相关的文件，共享存储段是内存的匿名段。

为使用一个共享内存段通常需要执行下面的步骤：

1. **调用 shmget()创建一个新共享内存段或取得一个既有共享内存段的标识符(**有其它进程创建的共享内存段)。这个调用将返回后续调用中需要用到的共享内存标识符。
2. **使用 shmat()来附上共享内存段，**即使该段成为调用进程的虚拟内存的一部分。
3. **此刻在程序中可以像对待其它可用内存那样对待这个共享内存段**。为引用这块共享内存，程序需要**使用由** shmat()调用返回的 **addr 值**，它是一个执行进程的虚拟地址空间中该共享内存段的起点的指针。
4. **调用 shmdt()来分离共享内存段**。这个调用之后，进程就无法在引用这块共享内存了。这一步是可选的，并且在进程终止时会自动完成这一步。
5. **调用 shmctl()来删除共享内存段**。是由当当前所有附加内存段的进程都与之分离之后内存段才会被销毁。只有一个进程需要执行这一步。

5 线程

5.1 基本概念

1) 什么是线程

线程是一个进程内部的一个控制序列。所有的进程都至少有一个执行线程。

一个进程的所有信息对该进程的所有线程都是共享的，包括可执行程序的代码，程序的全局内存和堆内存、栈以及文件描述符，即同一个进程的线程同处于一个地址空间，彼此能够互相访问栈地址（只要可见）。但不同的进程拥有完成不相干的地址空间。

如：

```
1 #include <stdio.h>
2 #include <pthread.h>
3 #include <unistd.h>
4 int main()
5 {
6    int num=100, a;
7    int *p = &num;
8    pid_t pid;
9
10    printf("before fork num:%d\n",num);
11    if((pid=fork())==0)
12    {
13        *p = 200; //修改了子进程的 p 地址所指的值
14        printf("%d@%d:%d\n",getppid(),getpid(),num);
```

```
15    }else if(pid>0)
16    {
17        wait(&a);
18        printf("%d@%d:%d\n",getppid(),getpid(),num);
19    }
20 }
21
22 输出：即父子进程地址 P 的值相同，但彼此不相关
23     before fork num:100
24    30754@30755:200
25    18163@30754:100
```

2) 线程标识

每个线程也有一个线程 ID。进程 ID 在整个系统中是唯一的，但线程 ID 不同，线程 ID 只有在它所属的进程上下文中才有意义。

```
#include<pthread.h>
int pthread_equal(pthread_t tid1, pthread_t tid2)
                                    返回值：若相等，返回非 0 数值：否则，返回 0
pthread_t pthread_self(void)
                                    返回值：调用线程的线程 ID
```

5.2 线程创建

在 POSIX 线程(pthread)情况下，程序开始运行时，它也是以**单**进程中的**单**个控制线程**（主线程）**启动的。

```
int pthread_create(pthread_t *tidp, pthread_atti_t *attr, void *(*start_rtn)(void*), void* arg)
返回值：若成功，返回 0；否则，返回出错编号
```

当创建成功返回时，新创建线程的线程 ID 会被设置成 **tidp** 指向的内存单元；新创建的线程从 **start_rtn（称为启动例程）**函数的地址开始运行，该函数只有一个不类型指针参数 art。

5.3 线程终止

如果进程中的任意线程调用了 exit、_Exit 或者_exit，那么整个进程就会终止，其中主线程退出也会使整个进程终止。与此类似，如果信号的默认动作是终止进程，那么，发送到线程的信号就会终止整个进程。

单个线程可以通过 3 种方式退出，因此可以在不终止整个进程的情况下，停止它的控制流：

1. 线程可以简单地从启动例程中返回，返回值是线程的退出码；
2. 线程可以被同一进程中的其他线程取消；
3. 线程调用 pthread_exit。

5.3.1 线程退出

```
#include <pthread.h>
void pthread_exit(void* rval_ptr);
```

pthread_exit 是线程退出函数，rval_ptr 是推出的状态码；

5.3.2 线程等待

```
#include <pthread.h>
int pthread_join(pthread_t thread, void** rval_ptr)
```

调用 pthead_join 线程将获得指定线程的退出状态；当线程调用了 pthread_join 函数则将一直**阻塞**，直到指定的线程调用 **pthread_exit**、从**启动例程**中返回或者**被取消**。

5.3.3 线程取消

```
#include <pthread.h>
void pthread_cancel(pthread_t tid);
```

线程可以通过调用 pthrad_cancel 函数来请求取消同一个进程中其他线程。在默认情况下，pthread_cancel 函数会使得 tid 标识的线程的行为表现为如同调用了参数为 PTHREAD_CANCEL 的 pthread_exit 函数，但是，线程可以忽略取消或者控制如何被取消。

5.3.4 线程清理处理程序

线程内存空间中存在着一个特殊栈，此栈用来记录清理函数的地址。该栈由如下两个函数来完成入栈和出栈操作。它们的执行顺序与它们的注册时相反。

```
#include <pthread.h>
void pthread_cleanup_push(void (*rtn)(void*), void * arg);
void pthread_cleanup_pop(int execute);
```

当线程执行**以下动作时**，清理函数 rtn 将被调用执行，其中 arg 是传递的函数。

- 调用 pthread_exit 时
- 响应取消请求时
- 用非零 execute 参数调用 pthread_cleanup_pop 时。

5.4 线程同步

术语临界区（critical section）是指访问某一共享资源的代码片段，并且这段代码的执行应为原子操作，亦即，同时访问同一共享资源的其他线程不应中断该片段的执行。

线程有如下的 5 种基本的同步机制。

5.4.1 互斥量(pthread_mutex_t)

互斥量(mutex)是线程同步的一把锁，该锁有两个状态：**已锁定**和**未锁定**。互斥量保证了同一个时间只有一个线程（获得锁的进程）访问临界区，并且该锁只能由获得锁的线程主动释放。

1) 初始化与销毁

互斥量的数据类型是 pthread_mutex_t。在使用互斥量之前，首先必须对它进行初始化，有如下两种情况：

- **静态分配**的互斥量：将互斥量赋值为常量 PTHREAD_MUTEX_INITIALIZER；
- **动态分配**的互斥量：通过调用 pthread_mutex_init 函数进行初始化，

如果是动态分配的互斥量 (例如，通过调用 malloc 函数或全局变量)，在释放内存前需要调用 pthread_mutex_destroy。

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

要默认的属性初始化互斥量，只需把 attr 设为 NULL。

2) 加锁与解锁

对互斥量进行加锁，需要调用 pthread_mutex_lock，如果互斥量已经上锁了，调用线程将阻塞知道互斥量被解锁；对互斥量解锁，需要调用 pthread_mutex_unlock。

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex)
int pthread_mutex_trylock(pthread_mutex_t *mutex)
int pthread_mutex_unlock(pthread_mutex_t *mutex)
```

所有函数的返回值：若成功，返回 0；否则，返回错误编号

如果线程不希望被阻塞，可以使用 pthread_mutex_trylock 尝试对互斥量进行加锁。调用 pthread_mutex_trylock 有两种情况：

- 互斥量处于**未锁住**状态：则直接返回 0，那么成功锁住互斥量，不会出现阻塞；
- 互斥量处于**已锁住**状态：则返回 EBUSY，不能锁住互斥量。

5.4.2 条件变量(pthread_cond_t)

互斥量为防止多个线程同时访问同一个共享量，而条件变量允许一个线程就某个共享变量（或其他共享资源）的状态变化通知其他线程，并让其他线程等待（阻塞于）这一通知。条件变量总是结合互斥量一起使用，条件变量就共享变量的状态改变发出通知，而互斥量则提供对该共享变量访问的互斥。

1) 初始化与销毁

与互斥量类似，对条件变量的初始化分为静态类型和动态类型，动态类型的初始化和销毁函数为：

```
#include <pthread.h>
int pthread_cond_init(pthread_cond_t *cond, pthread_condattr *attr)
int pthread_cond_destroy(pthread_cond_t *cond)
```

2) 等待通知

等待其他线程的发送的条件变量，可以使用如下两个函数，第二个只是增加了等待时间，其余功能都一样。

```
#include <pthread.h>
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)
int pthread_cond_timewait(pthread_cond_t *cond, pthread_mutex_t *mutex, struct timespec *tsptr)
;
```

调用线程将锁住的 mutex 信号量传递给 pthread_cond_wait 函数，然后该线程自动进入等待状态（阻塞），并对此互斥量解锁。从 pthread_cond_wait 唤醒必须**同时**具备如下条件：

- 其它线程对 cond 条件变量发送了唤醒操作：pthread_cond_signal 或 pthread_cond_broadcast。
- mutex 没有被锁住

3) 发送通知

```
#include <pthread.h>
int pthread_cond_signal(pthread_cond_t *cond)
int pthread_cond_broadcast(pthread_cond_t *cond)
```

这两个函数都可以用于唤醒阻塞于 cond 条件变量的线程，pthread_cond_signal 函数至少能唤醒一个等待该条件变量的线程，而 pthread_cond_broadcast 函数则能唤醒等待该条件变量的所有线程。

注意：

1. pthread_cond_signal 和 pthread_cond_broadcast 只是通知的 cond 条件变量的线程，而未对 mutex 信号量进行修改，需在调用上述两个唤醒函数之前手动进行解锁(pthread_mutex_unlock)。
2. 若调用了 pthread_cond_broadcast 唤醒所有等待线程，则只有一个线程能从阻塞状态唤醒，因为只有一个线程能获得 mutex 互斥量，其它线程只能等待获得 mutex 锁。
3. 应该把调用 pthread_cond_wait 函数放在 while 之中，判断条件是"共享资源不可用"，即当判断了共享资源不可用，应继续调用 pthread_cond_wait 等待。

5.4.3 读写锁(pthread_rwlock_t)

与只有 2 种状态的互斥量不同，读写锁有 3 种状态：**读**模式下**加锁**状态、**写**模式下**加锁**状态、**不加锁**状态。其中一次只有一个线程可以占有写模式的读写锁，但是多个线程可以同时占有读模式的读写锁。

在读写锁的 3 种状态下，进行读或写申请给出的响应各不相同：

- **写加锁状态：**所有试图对这个锁进行加锁的线程都会被阻塞。
- **读加锁状态：**所有试图以**读模式**对这个锁进行加锁都可以访问，但是任何以**写模式**对此锁进行加锁的线程都会阻塞。

1) 初始化与销毁

与互斥量相比，读写锁在使用之前必须初始化在释放内存之间必须销毁。

```
#include <pthread.h>
int pthread_rwlock_init(pthread_rwlock_t *rwlock , pthread_rwlockattr_t *attr);
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
返回值：若成功，返回 0；否则，返回错误编号
```

2) 加锁与解锁

对读写锁进行读加锁、写加锁，以及解锁的 3 个函数如下：

```
#include <pthread.h>
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock)
int pthread_rwlock_wdlock(pthread_rwlock_t *rwlock)
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock)
返回值：若成功，返回 0；否则，返回错误编号
```

Single UNIX Specification 还定义了读写锁原语的条件版本

```
#include <pthread.h>
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock)
int pthread_rwlock_trywdlock(pthread_rwlock_t *rwlock)
```

返回值：若成功，返回 0；否则，返回错误编号

5.4.4 自旋锁(pthread_spinlock_t)

自旋锁与互斥量类似，但它不是通过休眠使进程阻塞，而是在获取锁之前一直处于忙等（自旋）阻塞状态。自旋锁可用于以下情况：锁被持有的时间短，而且线程并不希望在重新调度上花费太多的成本。

自选锁通常作为底层原语用于实现其他类型的锁。但是，在用户层，自旋锁并不是非常有用，除非运行在不允许抢占的实时调度类中。所以这里不过多叙述.

5.4.5 屏障(pthread_barrier_t)

屏障是用户协调多个线程并行工作的同步机制。屏障允许每个线程等待，直到所有合作线程都到达某一点，然后从该点继续执行。

1) 初始化与销毁

与读写锁类似，屏障不分静态和动态变量，所有屏障类型变量都需通过 pthread_barrier_init 进行初始化。

```
#include <pthread.h>
int pthread_barrier_init(pthread_barrier_t *barrier, pthread_barrierattr_t *attr, unsigned int count)
int pthread_barrier_destroy(pthread_barrier_t *barrier);
```

初始化屏障时，可以使用 count 参数指定，在允许所有线程继续运行之前，必须到达屏障的线程数目。

2)等待

可以使用 pthread_barrier_wait 函数来表明，线程已完成工作，准备等所有其他线程赶上来。

```
#include <pthread.h>
int pthread_barrier_wait(pthread_barrier_t *barrier)
```

返回值：若成功，返回 0 或者 PTHREAD_BARRTER_SERIAL_THREAD；否则，返回错误编号

调用 pthread_barrier_wait 的线程在屏障计数(调用 pthread_barrier_init 时设定)未满足条件时，会进入休眠状态。如果该线程是最后一个调用 pthread_barrier_wait 的线程，就满足了屏障计数，所有线程都被唤醒。