辅助学习资料

参考书 1：《Unix 环境高级编程》W.Richard Stevens [美]

本讲课堂义作为 APUE 的引导。适合初学 Linux 的学员。

《TCP/IP 详解》(3 卷)，《UNIX 网络编程》（2 卷）

参考书 2：《Linux 系统编程》RobertLoVe [美]

参考书 3：《Linux/UNIX 系统编程手册》 Michael Kerrisk [德]

参考书 4：《Unix 内核源码剖析》 青柳隆宏[日]

业内知名：《Linux 内核源代码情景分析》、《Linux 内核设计与实现》、《深入理解 Linux 内核》

文件 IO

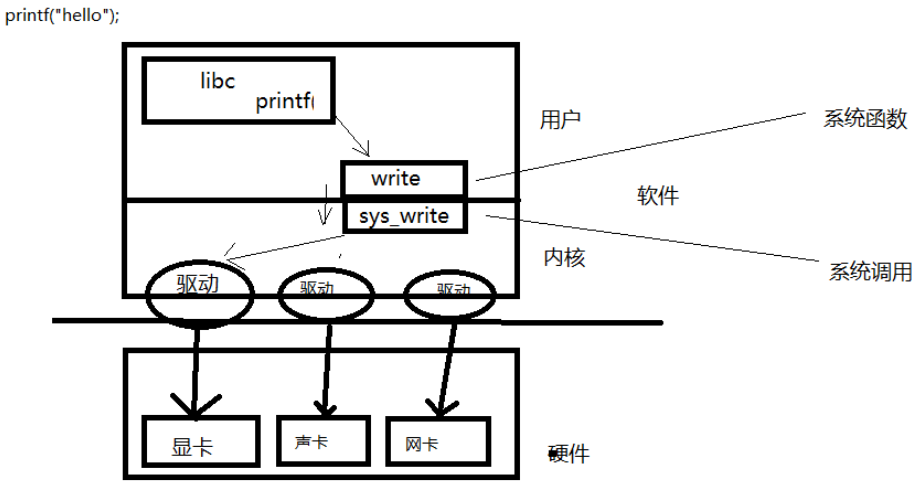
==========

系统调用

什么是系统调用：

由操作系统实现并提供给外部应用程序的编程接口。(Application Programming Interface，API)。是应用程序同系统之间数据交互的桥梁。

C 标准函数和系统函数调用关系。一个 helloworld 如何打印到屏幕。



C 标准库文件 IO 函数。

fopen、fclose、fseek、fgets、fputs、fread、fwrite.....
r 只读、 r+读写
w 只写并截断为 0、 w+读写并截断为 0
a 追加只写、 a+追加读写

open/close 函数

函数原型：

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
int close(int fd);
```

常用参数

O_RDONLY、O_WRONLY、O_RDWR
O_APPEND、O_CREAT、O_EXCL、 O_TRUNC、 O_NONBLOCK
创建文件时，指定文件访问权限。权限同时受 umask 影响。结论为：
文件权限 = mode & ~umask
使用头文件：<fcntl.h>

open 常见错误 ：

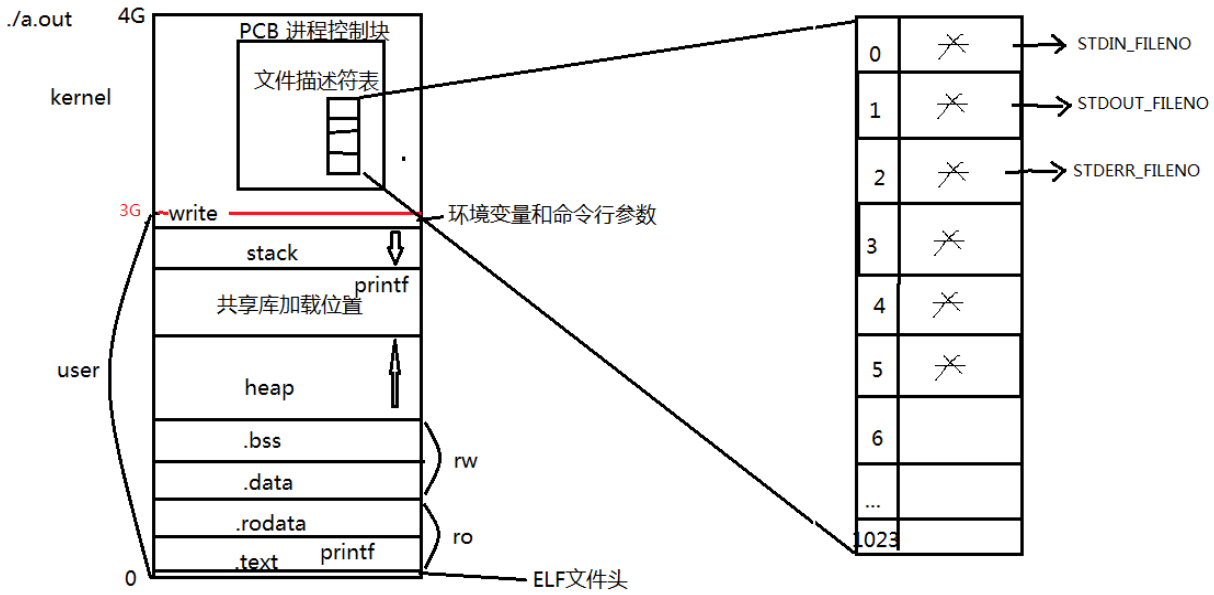
1. 打开文件不存在
2. 以写方式打开只读文件(打开文件没有对应权限)
3. 以只写方式打开目录

文件描述符：

PCB 进程控制块

可使用命令 locate sched.h 查看位置：
/usr/src/linux-headers-3.16.0-30/include/linux/sched.h

struct task_struct { 结构体



文件描述符表

结构体 PCB 的成员变量 file_struct *file 指向文件描述符表。

从应用程序使用角度，该指针可理解记忆成一个字符指针数组，下标 0/1/2/3/4...找到文件结构体。

本质是一个键值对 0、1、2...都分别对应具体地址。但键值对使用的特性是自动映射，我们只操作键不直接使用值。

新打开文件返回文件描述符表中未使用的最小文件描述符。

STDIN_FILENO	0
STDOUT_FILENO	1
STDERR_FILENO	2

最大打开文件数

一个进程默认打开文件的个数 1024。

命令查看 ulimit -a 查看 open files 对应值。默认为 1024

可以使用 ulimit -n 4096 修改

当然也可以通过修改系统配置文件永久修改该值，但是不建议这样操作。

cat /proc/sys/fs/file-max 可以查看该电脑最大可以打开的文件个数。受内存大小影响。

FILE 结构体

主要包含文件描述符、文件读写位置、IO 缓冲区三部分内容。

```
struct file {
    ...
    文件的偏移量;
    文件的访问权限;
    文件的打开标志;
    文件内核缓冲区的首地址;
    struct operations * f_op;
    ...
};
```

查看方法：

(1) /usr/src/linux-headers-3.16.0-30/include/linux/fs.h

(2) lxr：百度 lxr → lxr.oss.org.cn → 选择内核版本(如 3.10) → 点击 File Search 进行搜索

→ 关键字：“include/linux/fs.h” → Ctrl+F 查找 “struct file {”

→ 得到文件内核中结构体定义

→ “struct file_operations”文件内容操作函数指针

→ “struct inode_operations”文件属性操作函数指针

read/write 函数

ssize_t read(int fd, void *buf, size_t count);

ssize_t write(int fd, const void *buf, size_t count);

read 与 write 函数原型类似。使用时需注意：read/write 函数的第三个参数。

练习：编写程序实现简单的 cp 功能。

程序比较：如果一个只读一个字节实现文件拷贝，使用 read、write 效率高，还是使用对应的标库函数(fgetc、fputc)效率高呢？

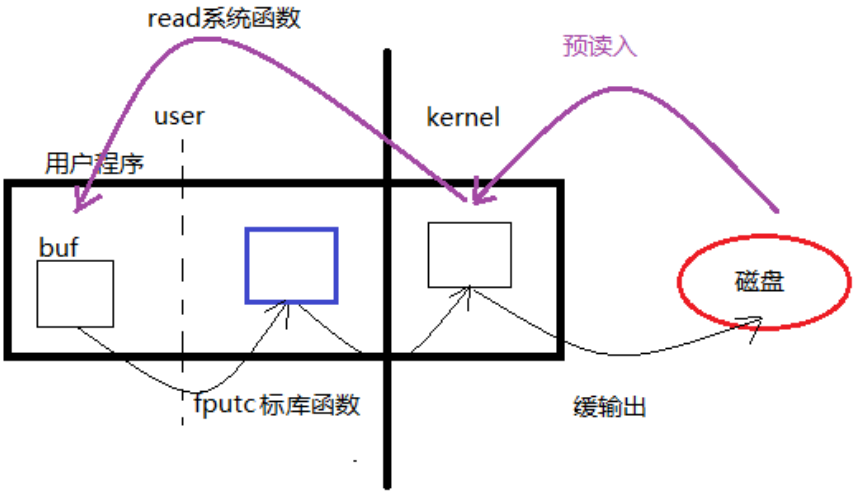
strace 命令

shell 中使用 strace 命令跟踪程序执行，查看调用的系统函数。

缓冲区

read、write 函数常常被称为 Unbuffered I/O。指的是无用户及缓冲区。但不保证不使用内核缓冲区。

预读入缓输出



错误处理函数：

错误号：errno

perror 函数：　　**void perror(const char *s);**
strerror 函数：　　**char *strerror(int errnum);**

查看错误号：
/usr/include/asm-generic/errno-base.h
/usr/include/asm-generic/errno.h

```
#define EPERM      1    /* Operation not permitted */
#define ENOENT     2    /* No such file or directory */
#define ESRCH      3    /* No such process */
#define EINTR      4    /* Interrupted system call */
#define EIO        5    /* I/O error */
#define ENXIO      6    /* No such device or address */
#define E2BIG      7    /* Argument list too long */
#define ENOEXEC    8    /* Exec format error */
#define EBADF      9    /* Bad file number */
#define ECHILD     10   /* No child processes */
#define EAGAIN     11   /* Try again */
#define ENOMEM     12   /* Out of memory */
#define EACCES     13   /* Permission denied */
```

```
#define EFAULT      14  /* Bad address */
#define ENOTBLK     15  /* Block device required */
#define EBUSY       16  /* Device or resource busy */
#define EEXIST      17  /* File exists */
#define EXDEV       18  /* Cross-device link */
#define ENODEV      19  /* No such device */
#define ENOTDIR     20  /* Not a directory */
#define EISDIR      21  /* Is a directory */
#define EINVAL      22  /* Invalid argument */
#define ENFILE      23  /* File table overflow */
#define EMFILE      24  /* Too many open files */
#define ENOTTY      25  /* Not a typewriter */
#define ETXTBSY     26  /* Text file busy */
#define EFBIG       27  /* File too large */
#define ENOSPC      28  /* No space left on device */
#define ESPIPE      29  /* Illegal seek */
#define EROFS       30  /* Read-only file system */
#define EMLINK      31  /* Too many links */
#define EPIPE       32  /* Broken pipe */
#define EDOM        33  /* Math argument out of domain of func */
#define ERANGE      34  /* Math result not representable */
```

阻塞、非阻塞

读常规文件是不会阻塞的，不管读多少字节，read 一定会在有限的时间内返回。从终端设备或网络读则不一定，如果从终端输入的数据没有换行符，调用 read 读终端设备就会阻塞，如果网络上没有接收到数据包，调用 read 从网络读就会阻塞，至于会阻塞多长时间也是不确定的，如果一直没有数据到达就一直阻塞在那里。同样，写常规文件是不会阻塞的，而向终端设备或网络写则不一定。

现在明确一下阻塞（Block）这个概念。当进程调用一个阻塞的系统函数时，该进程被置于睡眠（Sleep）状态，这时内核调度其它进程运行，直到该进程等待的事件发生了（比如网络上接收到数据包，或者调用 sleep 指定的睡眠时间到了）它才有可能继续运行。与睡眠状态相对的是运行（Running）状态，在 Linux 内核中，处于运行状态的进程分为两种情况：

正在被调度执行。CPU 处于该进程的上下文环境中，程序计数器（eip）里保存着该进程的指令地址，通用寄存器里保存着该进程运算过程的中间结果，正在执行该进程的指令，正在读写该进程的地址空间。

就绪状态。该进程不需要等待什么事件发生，随时都可以执行，但 CPU 暂时还在执行另一个进程，所以该进程在一个就绪队列中等待被内核调度。系统中可能同时有多个就绪的进程，那么该调度谁执行呢？内核的调度算法是基于优先级和时间片的，而且会根据每个进程的运行情况动态调整它的优先级和时间片，让每个进程都能比较公平地得到机会执行，同时要兼顾用户体验，不能让和用户交互的进程响应太慢。

阻塞读终端：　　【block_readtty.c】

非阻塞读终端　　【nonblock_readtty.c】

非阻塞读终端和等待超时　　【nonblock_timeout.c】

注意，阻塞与非阻塞是对于文件而言的。而不是 read、write 等的属性。read 终端，默认阻塞读。

总结 read 函数返回值：

1. 返回非零值：　实际 read 到的字节数
2. 返回-1：　1）：errno != EAGAIN (或!= EWOULDBLOCK)　read 出错
   2）：errno == EAGAIN (或== EWOULDBLOCK)　设置了非阻塞读，并且没有数据到达。
3. 返回 0：读到文件末尾

lseek 函数

文件偏移

Linux 中可使用系统函数 lseek 来修改文件偏移量(读写位置)

每个打开的文件都记录着当前读写位置，打开文件时读写位置是 0，表示文件开头，通常读写多少个字节就会将读写位置往后移多少个字节。但是有一个例外，如果以 O_APPEND 方式打开，每次写操作都会在文件末尾追加数据，然后将读写位置移到新的文件末尾。lseek 和标准 I/O 库的 fseek 函数类似，可以移动当前读写位置（或者叫偏移量）。

回忆 fseek 的作用及常用参数。 SEEK_SET、SEEK_CUR、SEEK_END

int fseek(FILE *stream, long offset, int whence); 成功返回 0；失败返回-1

特别的：超出文件末尾位置返回 0；往回超出文件头位置，返回-1

off_t lseek(int fd, off_t offset, int whence); 失败返回-1；**成功：返回的值是较文件起始位置向后的偏移量。**

特别的：lseek 允许超过文件结尾设置偏移量，文件会因此被拓展。

注意文件“读”和“写”使用同一偏移位置。　　【lseek.c】

lseek 常用应用：

1. 使用 lseek 拓展文件：write 操作才能实质性的拓展文件。单 lseek 是不能进行拓展的。
   一般：write(fd, "a", 1);

od -tcx filename　查看文件的 16 进制表示形式
od -tcd filename　查看文件的 10 进制表示形式

2. 通过 lseek 获取文件的大小：lseek(fd, 0, SEEK_END);　　【lseek_test.c】

【最后注意】：lseek 函数返回的偏移量总是相对于文件头而言。

fcntl 函数

改变一个【已经打开】的文件的 访问控制属性。
重点掌握两个参数的使用，F_GETFL 和 F_SETFL。
【fcntl.c】

ioctl 函数

对设备的 I/O 通道进行管理，控制设备特性。(主要应用于设备驱动程序中)。

通常用来获取文件的【物理特性】（该特性，不同文件类型所含有的值各不相同）
【ioctl.c】

传入传出参数

传入参数：

const 关键字修饰的 指针变量 在函数内部读操作。 char *strcpy(cnost char *src, char *dst);

传出参数：

1. 指针做为函数参数
2. 函数调用前，指针指向的空间可以无意义，调用后指针指向的空间有意义，且作为函数的返回值传出
3. 在函数内部写操作。

传入传出参数：

1. 调用前指向的空间有实际意义 2. 调用期间在函数内读、写(改变原值)操作 3.作为函数返回值传出。

扩展阅读：

关于虚拟 4G 内存的描述和解析：

一个进程用到的虚拟地址是由内存区域表来管理的，实际用不了 4G。而用到的内存区域，会通过页表映射到物理内存。

所以每个进程都可以使用同样的虚拟内存地址而不冲突，因为它们的物理地址实际上是不同的。内核用的是 3G 以上的 1G 虚拟内存地址，

其中 896M 是直接映射到物理地址的，128M 按需映射 896M 以上的所谓高位内存。各进程使用的是同一个内核。

首先要分清“可以寻址”和“实际使用”的区别。

其实我们讲的每个进程都有 4G 虚拟地址空间，讲的都是“可以寻址”4G，意思是虚拟地址的 0-3G 对于一个进程的用户态和内核态来说是可以访问的，而 3-4G 是只有进程的内核态可以访问的。并不是说这个进程会用满这些空间。

其次，所谓“独立拥有的虚拟地址”是指对于每一个进程，都可以访问自己的 0-4G 的虚拟地址。虚拟地址是“虚拟”的，需要转化为“真实”的物理地址。

好比你有你的地址簿，我有我的地址簿。你和我的地址簿都有 1、2、3、4 页，但是每页里面的实际内容是不一样的，我的地址簿第 1 页写着 3 你的地址簿第 1 页写着 4，对于你、我自己来说都是用第 1 页（虚拟），实际上用的分别是第 3、4 页（物理），不冲突。

内核用的 896M 虚拟地址是直接映射的，意思是只要把虚拟地址减去一个偏移量（3G）就等于物理地址。同样，这里指的还是寻址，实际使用前还是要分配内存。而且 896M 只是个最大值。如果物理内存小，内核能使用（分配）的可用内存也小。

文件系统

文件存储

首先了解如下文件存储相关概念：inode、 dentry、 数据存储、文件系统。

inode

其本质为结构体，存储文件的属性信息。如：权限、类型、大小、时间、用户、盘块位置……也叫作文件属性管理结构，大多数的 inode 都存储在磁盘上。

少量常用、近期使用的 inode 会被缓存到内存中。

dentry

目录项，其本质依然是结构体，重要成员变量有两个 {文件名，inode，...}，而文件内容(data)保存在磁盘盘块中。

文件系统

文件系统是，一组规则，规定对文件的存储及读取的一般方法。文件系统在磁盘格式化过程中指定。

常见的文件系统有：fat32 ntfs exfat ext2 、ext3 、ext4

文件操作

stat 函数

获取文件属性，(从 inode 结构体中获取)

int stat(const char *path, struct stat *buf); 成返回 0；失败返回-1 设置 errno 为恰当值。

参数 1：文件名

参数 2：inode 结构体指针 (传出参数)

文件属性将通过传出参数返回给调用者。

练习：使用 stat 函数查看文件属性

【stat.c】

lstat 函数

int lstat(const char *path, struct stat *buf); 成返回 0；失败返回-1 设置 errno 为恰当值。

练习：给定文件名，判断文件类型。
【get_file_type.c】

文件类型判断方法：st_mode 取高 4 位。 但应使用宏函数：

S_ISREG(m) is it a regular file?
S_ISDIR(m) directory?
S_ISCHR(m) character device?
S_ISBLK(m) block device?
S_ISFIFO(m) FIFO (named pipe)?
S_ISLNK(m) symbolic link? (Not in POSIX.1-1996.)
S_ISSOCK(m) socket? (Not in POSIX.1-1996.)

穿透符号链接：stat：会；lstat：不会

truncate 函数

截断文件长度成指定长度。常用来拓展文件大小，代替 lseek。

int truncate(const char *path, off_t length); 成功：0；失败：-1 设置 errno 为相应值
int ftruncate(int fd, off_t length);

link 函数

思考，为什么目录项要游离于 inode 之外，画蛇添足般的将文件名单独存储呢？？这样的存储方式有什么样的好处呢？

其目的是为了实现文件共享。Linux 允许多个目录项共享一个 inode，即共享盘块(data)。不同文件名，在人类眼中将它理解成两个文件，但是在内核眼里是同一个文件。

link 函数，可以为已经存在的文件创建目录项(硬链接)。

int link(const char *oldpath, const char *newpath); 成功：0；失败：-1 设置 errno 为相应值

注意：由于两个参数可以使用“相对/绝对路径+文件名”的方式来指定，所以易出错。

如：link("../abc/a.c", "../ioc/b.c")若 a.c，b.c 都对， 但 abc，ioc 目录不存在也会失败。

mv 命令既是修改了目录项，而并不修改文件本身。

unlink 函数

删除一个文件的目录项；

int unlink(const char *pathname); 成功：0；失败：-1 设置 errno 为相应值

练习：编程实现 mv 命令的改名操作
【imp_mv.c】

注意 Linux 下删除文件的机制：不断将 st_nlink -1，直至减到 0 为止。无目录项对应的文件，将会被操作系统择机释放。(具体时间由系统内部调度算法决定)

因此，我们删除文件，从某种意义上说，**只是让文件具备了被释放的条件。**

unlink 函数的特征：清除文件时，如果文件的硬链接数到 0 了，没有 dentry 对应，但该文件仍不会马上被释放。要等到所有打开该文件的进程关闭该文件，系统才会挑时间将该文件释放掉。 【unlink_exe.c】

隐式回收

当进程结束运行时，所有该进程打开的文件会被关闭，申请的内存空间会被释放。系统的这一特性称之为隐式回收系统资源。

目录操作

工作目录：“./”代表当前目录，指的是进程当前的工作目录，默认是进程所执行的程序所在的目录位置。

getcwd 函数

获取进程当前工作目录 (卷 3，标库函数)

char *getcwd(char *buf, size_t size); 成功：buf 中保存当前进程工作目录位置。失败返回 NULL。

chdir 函数

改变当前进程的工作目录

int chdir(const char *path); 成功：0；失败：-1 设置 errno 为相应值

练习：获取及修改当前进程的工作目录，并打印至屏幕。
【imp_cd.c】

文件、目录权限

注意：目录文件也是“文件”。其文件内容是该目录下所有子文件的目录项 dentry。 可以尝试用 vim 打开一个目录。

	r	w	x
文件	文件的内容可以被查看 cat、more、less…	内容可以被修改 vi、> …	可以运行产生一个进程 ./文件名
目录	目录可以被浏览 ls、tree…	创建、删除、修改文件 mv、touch、mkdir…	可以被打开、进入 cd

目录设置黏住位：若有 w 权限，创建不变，删除、修改只能由 root、目录所有者、文件所有者操作。

opendir 函数

根据传入的目录名打开一个目录 (库函数)　　DIR * 类似于 FILE *

DIR *opendir(const char *name);　成功返回指向该目录结构体指针，失败返回 NULL

参数支持相对路径、绝对路径两种方式：例如：打开当前目录：① getcwd() , opendir() ② opendir(".");

closedir 函数

关闭打开的目录

int closedir(DIR *dirp);　　成功：0；失败：-1 设置 errno 为相应值

readdir 函数

读取目录　　(库函数)

struct dirent *readdir(DIR *dirp);　成功返回目录项结构体指针；失败返回 NULL 设置 errno 为相应值

需注意返回值，读取数据结束时也返回 NULL 值，所以应借助 errno 进一步加以区分。

struct 结构体：

```
struct dirent {
    ino_t          d_ino;      inode 编号
    off_t          d_off;
    unsigned short d_reclen;   文件名有效长度
```

```
    unsigned char    d_type;        类型(vim 打开看到的类似@*/等)
    char             d_name[256];文件名
};
```

其成员变量重点记忆两个：d_ino、d_name。实际应用中只使用到 d_name。

练习 1：实现简单的 ls 功能。

【imp_ls.c】

练习 2：实现 ls 不打印隐藏文件。每 5 个文件换一个行显示。

【imp_ls2.c】

拓展 1：实现 ls -a -l 功能。

拓展 2：统计目录及其子目录中的普通文件的个数

递归遍历目录

查询指定目录，递归列出目录中文件，同时显示文件大小。

【ls_R.c】

重定向

dup 函数

功能:文件描述符拷贝。

使用现有的文件描述符，拷贝生成一个新的文件描述符，且函数调用前后这个两个文件描述符指向同一文件。

int dup(int oldfd); 成功：返回一个新文件描述符；失败：-1 设置 errno 为相应值

dup2 函数

功能:文件描述符拷贝。重定向文件描述符指向。

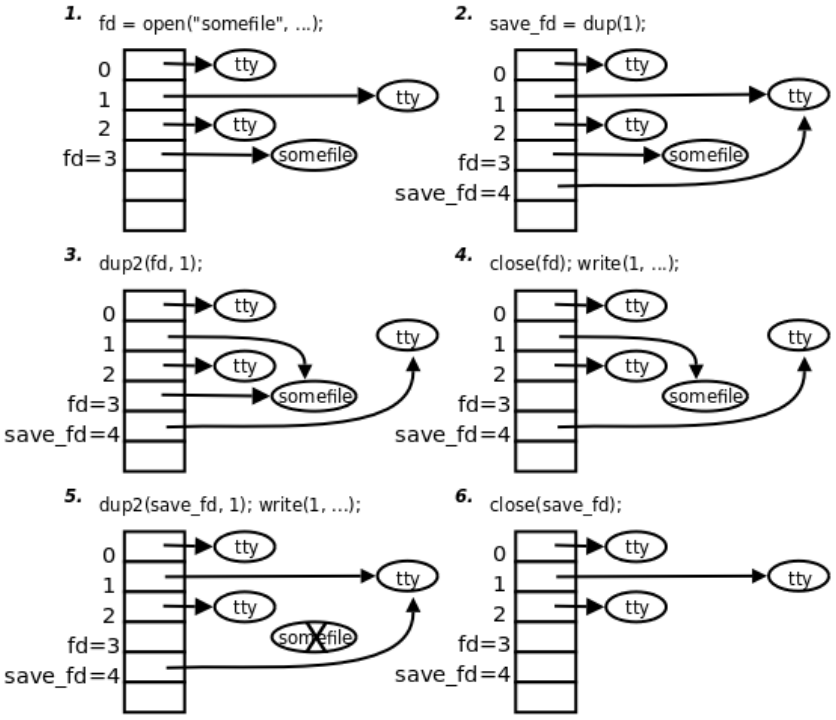
通过该函数可实现命令行“重定向”功能。使得原来指向某文件的文件描述符，指向其他指定文件。

int dup2(int oldfd, int newfd);

成功：返回一个新文件描述符；

如果 oldfd 有效，则返回的文件描述符与 oldfd 指向同一文件。

失败：如果 oldfd 无效，调用失败，关闭 newfd。返回-1，同时设置 errno 为相应值。



重定向示

记忆方法两种：

1. 文件描述符的本质角度理解记忆。
2. 从函数原型及使用角度，反向记忆。

练习：借助 dup 函数编写 mycat 程序，实现 cat file1 > file2 命令相似功能。

【mycat.c】

fcntl 函数

当 fcntl 的第二个参数为 F_DUPFD 时， 它的作用是根据一个已有的文件描述符，复制生成一个新的文件描述符。此时，fcntl 相当于 dup 和 dup2 函数。

参 3 指定为 0 时，因为 0 号文件描述符已经被占用。所以函数自动用一个最小可用文件描述符。

参 3 指定为 9 时，如果该文件描述符未被占用，则返回 9。否则，返回大于 9 的可用文件描述符。

【fcntl_dup.c】

进程相关概念

程序和进程

程序，是指编译好的二进制文件，在磁盘上，不占用系统资源(cpu、内存、打开的文件、设备、锁....)

进程，是一个抽象的概念，与操作系统原理联系紧密。进程是活跃的程序，占用系统资源。在内存中执行。(程序运行起来，产生一个进程)

程序 → 剧本(纸)　　进程 → 戏(舞台、演员、灯光、道具...)

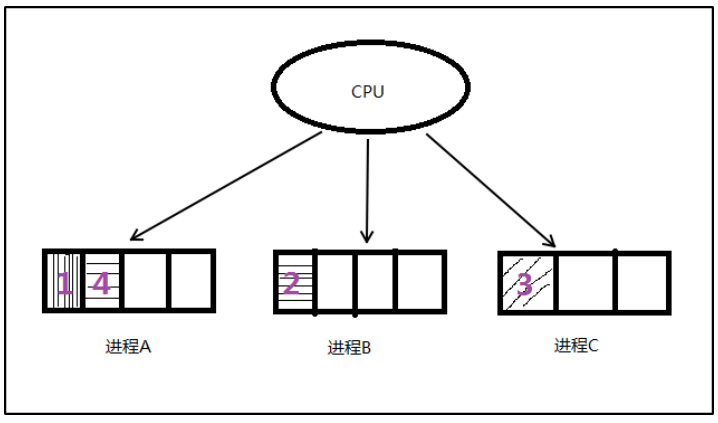
同一个剧本可以在多个舞台同时上演。同样，同一个程序也可以加载为不同的进程(彼此之间互不影响)

如：同时开两个终端。各自都有一个 bash 但彼此 ID 不同。

并发

并发，在操作系统中，一个时间段中有多个进程都处于已启动运行到运行完毕之间的状态。但，任一个时刻点上仍只有一个进程在运行。

例如，当下，我们使用计算机时可以边听音乐边聊天边上网。 若笼统的将他们均看做一个进程的话，为什么可以同时运行呢，因为并发。



分时复用 cpu

单道程序设计

所有进程一个一个排对执行。若 A 阻塞，B 只能等待，即使 CPU 处于空闲状态。而在人机交互时阻塞的出现时必然的。所有这种模型在系统资源利用上及其不合理，在计算机发展历史上存在不久，大部分便被淘汰了。•

多道程序设计

在计算机内存中同时存放几道相互独立的程序，它们在管理程序控制之下，相互穿插的运行。多道程序设计必须有硬件基础作为保证。

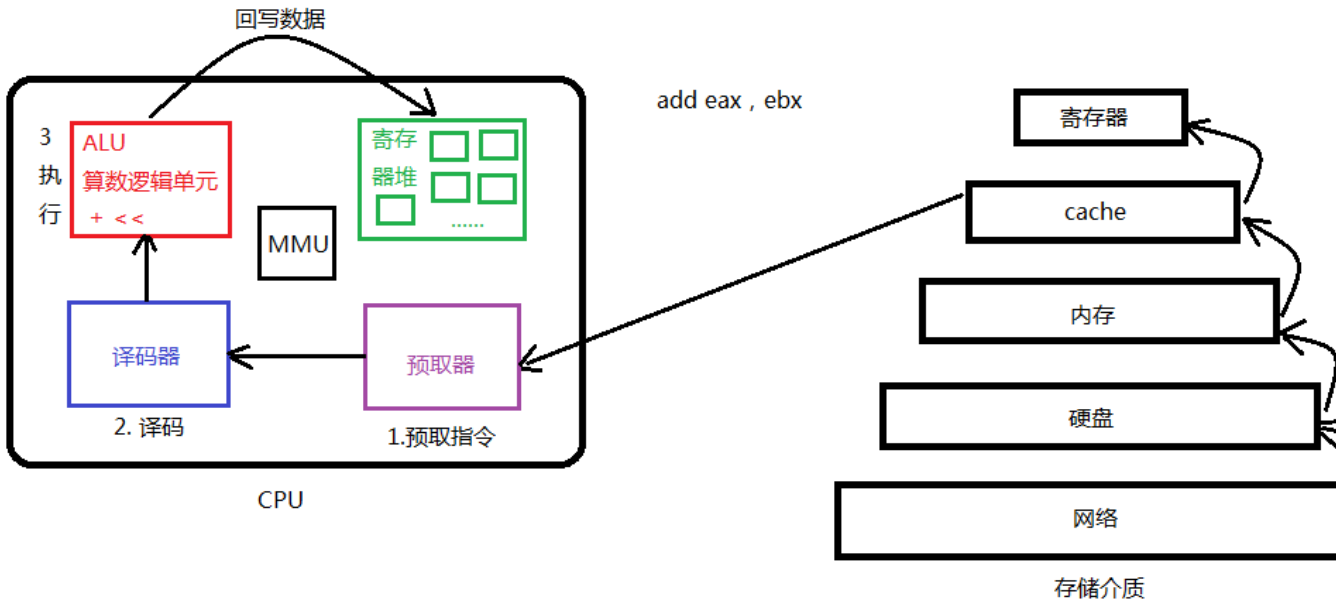
时钟中断即为多道程序设计模型的理论基础。 并发时，任意进程在执行期间都不希望放弃 cpu。因此系统需要一种强制让进程让出 cpu 资源的手段。时钟中断有硬件基础作为保障，对进程而言不可抗拒。 操作系统中的中断处理函数，来负责调度程序执行。

在多道程序设计模型中，多个进程轮流使用 CPU (分时复用 CPU 资源)。而当下常见 CPU 为纳秒级，1 秒可以执行大约 10 亿条指令。由于人眼的反应速度是毫秒级，所以看似同时在运行。

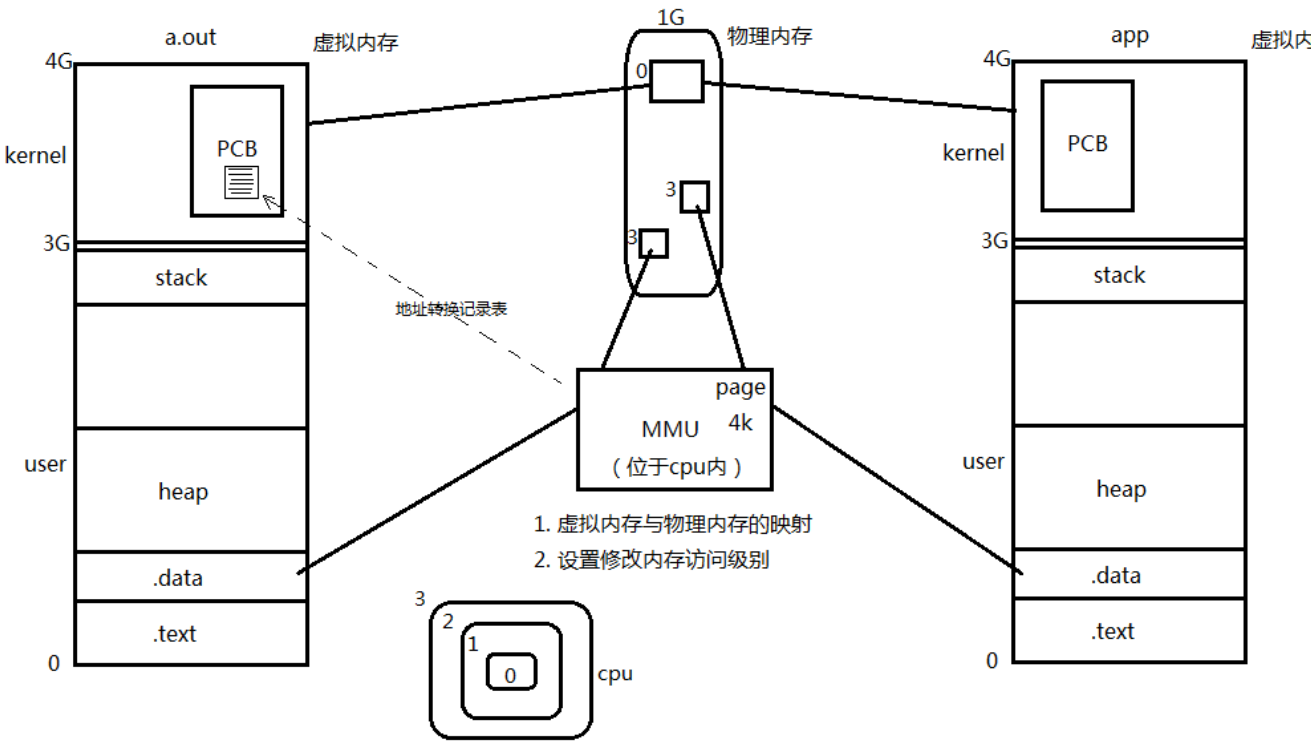
1s = 1000ms, 1ms = 1000us, 1us = 1000ns　　1000000000

实质上，并发是宏观并行，微观串行！　　-----推动了计算机蓬勃发展，将人类引入了多媒体时代。

CPU 和 MMU



中央处理器(CPU)



内存管理单元 MMU

进程控制块 PCB

我们知道，每个进程在内核中都有一个进程控制块（PCB）来维护进程相关的信息，Linux 内核的进程控制块是 task_struct 结构体。

/usr/src/linux-headers-3.16.0-30/include/linux/sched.h 文件中可以查看 struct task_struct 结构体定义。其内部成员有很多，我们重点掌握以下部分即可：

* 进程 id。系统中每个进程有唯一的 id，在 C 语言中用 pid_t 类型表示，其实就是一个非负整数。

* 进程的状态，有就绪、运行、挂起、停止等状态。

* 进程切换时需要保存和恢复的一些 CPU 寄存器。

* 描述虚拟地址空间的信息。

* 描述控制终端的信息。

* 当前工作目录（Current Working Directory）。

* umask 掩码。

* 文件描述符表，包含很多指向 file 结构体的指针。

* 和信号相关的信息。

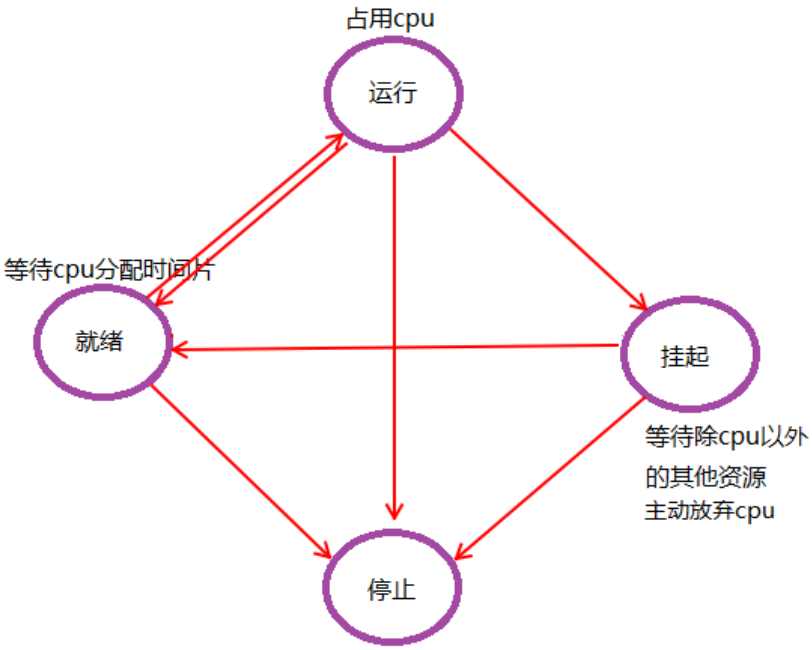
* 用户 id 和组 id。

* 会话（Session）和进程组。

* 进程可以使用的资源上限（Resource Limit）。

进程状态

进程基本的状态有 5 种。分别为初始态，就绪态，运行态，挂起态与终止态。其中初始态为进程准备阶段，常与就绪态结合来看。



环境变量：

环境变量，是指在操作系统中用来指定操作系统运行环境的一些参数。通常具备以下特征：

① 字符串(本质) ② 有统一的格式：名=值[:值] ③ 值用来描述进程环境信息。

存储形式：与命令行参数类似。char *[]数组，数组名 environ，内部存储字符串，NULL 作为哨兵结尾。

使用形式：与命令行参数类似。

加载位置：与命令行参数类似。位于用户区，高于 stack 的起始位置。

引入环境变量表：须声明环境变量。extern char ** environ;

练习：打印当前进程的所有环境变量。

【environ.c】

常见环境变量

按照惯例，环境变量字符串都是 name=value 这样的形式，大多数 name 由大写字母加下划线组成，一般把 name 的部分叫做环境变量，value 的部分则是环境变量的值。环境变量定义了进程的运行环境，一些比较重要的环境变量的含义如下：

PATH

可执行文件的搜索路径。ls 命令也是一个程序，执行它不需要提供完整的路径名/bin/ls，然而通常我们执行当前目录下的程序 a.out 却需要提供完整的路径名./a.out，这是因为 PATH 环境变量的值里面包含了 ls 命令所在的目录/bin，却不包含 a.out 所在的目录。PATH 环境变量的值可以包含多个目录，用:号隔开。在 Shell 中用 echo 命令可以查看这个环境变量的值：

```
$ echo $PATH
```

SHELL

当前 Shell，它的值通常是/bin/bash。

TERM

当前终端类型，在图形界面终端下它的值通常是 xterm，终端类型决定了一些程序的输出显示方式，比如图形界面终端可以显示汉字，而字符终端一般不行。

LANG

语言和 locale，决定了字符编码以及时间、货币等信息的显示格式。

HOME

当前用户主目录的路径，很多程序需要在主目录下保存配置文件，使得每个用户在运行该程序时都有自己的一套配置。

getenv 函数

获取环境变量值

char *getenv(const char *name); 成功：返回环境变量的值；失败：NULL (name 不存在)

练习：编程实现 getenv 函数。

【getenv.c】

setenv 函数

设置环境变量的值

int setenv(const char *name, const char *value, int overwrite); 成功：0；失败：-1

参数 overwrite 取值： 1：覆盖原环境变量

0：不覆盖。(该参数常用于设置新环境变量，如：ABC = haha-day-night)

unsetenv 函数

删除环境变量 name 的定义

int unsetenv(const char *name); 成功：0；失败：-1

注意事项：name 不存在仍返回 0(成功)，当 name 命名为"ABC="时则会出错。

进程控制

fork 函数

创建一个子进程。

pid_t fork(void); 失败返回-1；成功返回：① 父进程返回子进程的 ID(非负) ② 子进程返回 0

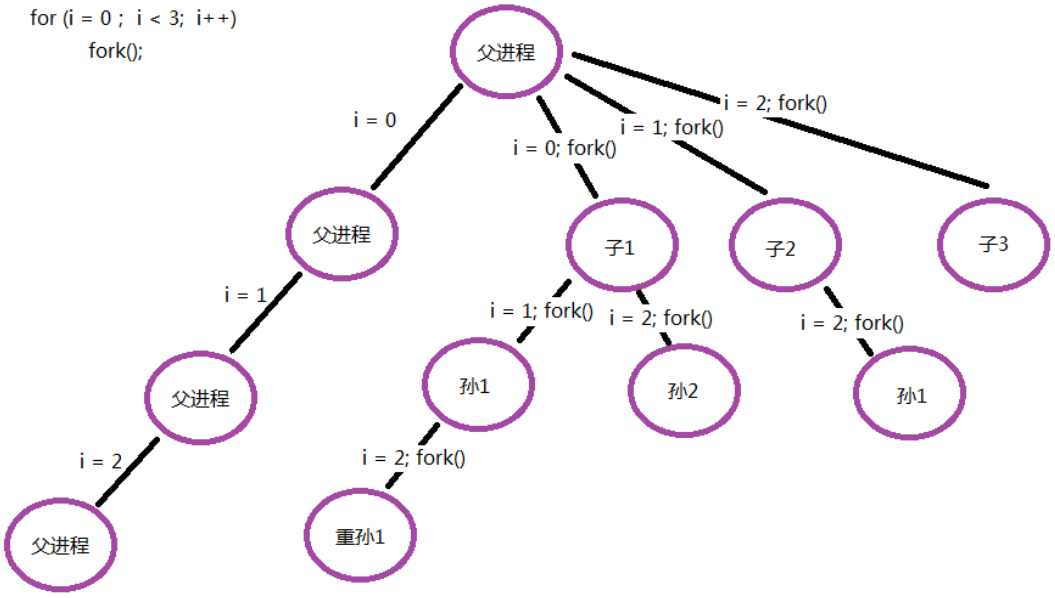
pid_t 类型表示进程 ID，但为了表示-1，它是有符号整型。(0 不是有效进程 ID，init 最小，为 1)

注意返回值，不是 fork 函数能返回两个值，而是 fork 后，fork 函数变为两个，父子需【各自】返回一个。

循环创建 n 个子进程

一次 fork 函数调用可以创建一个子进程。那么创建 N 个子进程应该怎样实现呢？

简单想，for(i = 0; i < n; i++) { fork() } 即可。但这样创建的是 N 个子进程吗？



循环创建 N 个子进程

从上图我们可以很清晰的看到，当 n 为 3 时候，循环创建了(2^n)-1 个子进程，而不是 N 的子进程。需要在循环的过程，保证子进程不再执行 fork ，因此当(fork() == 0)时，子进程应该立即 break;才正确。

练习：通过命令行参数指定创建进程的个数，每个进程休眠 1S 打印自己是第几个被创建的进程。如：第 1 个子进程休眠 0 秒打印：“我是第 1 个子进程”；第 2 个进程休眠 1 秒打印：“我是第 2 个子进程”；第 3 个进程休眠 2 秒打印：“我是第 3 个子进程”。

【fork1.c】

通过该练习**掌握框架**：循环创建 n 个子进程，使用循环因子 i 对创建的子进程加以区分。

getpid 函数

获取当前进程 ID

```
pid_t getpid(void);
```

getppid 函数

获取当前进程的父进程 ID

```
pid_t getppid(void);
```

区分一个函数是“系统函数”还是“库函数”依据：

② 是否访问内核数据结构

② 是否访问外部硬件资源　　二者有任一 → 系统函数；二者均无 → 库函数

getuid 函数

获取当前进程实际用户 ID

uid_t getuid(void);

获取当前进程有效用户 ID

uid_t geteuid(void);

getgid 函数

获取当前进程使用用户组 ID

gid_t getgid(void);

获取当前进程有效用户组 ID

gid_t getegid(void);

进程共享

父子进程之间在 fork 后。有哪些相同，那些相异之处呢？

刚 fork 之后：

父子相同处: 全局变量、.data、.text、栈、堆、环境变量、用户 ID、宿主目录、进程工作目录、信号处理方式...

父子不同处: 1.进程 ID　2.fork 返回值　3.父进程 ID　4.进程运行时间　5.闹钟(定时器)　6.未决信号集

似乎，子进程复制了父进程 0-3G 用户空间内容，以及父进程的 PCB，但 pid 不同。真的每 fork 一个子进程都要将父进程的 0-3G 地址空间完全拷贝一份，然后在映射至物理内存吗？

当然不是！父子进程间遵循**读时共享写时复制**的原则。这样设计，无论子进程执行父进程的逻辑还是执行自己的逻辑都能节省内存开销。

练习：编写程序测试，父子进程是否共享全局变。

【fork_shared.c】

重点注意！躲避父子进程共享全局变量的知识误区！

【重点】：父子进程共享：1. 文件描述符(打开文件的结构体)　2. mmap 建立的映射区(进程间通信详解)

特别的，fork 之后父进程先执行还是子进程先执行不确定。取决于内核所使用的调度算

法。

gdb 调试

使用 gdb 调试的时候，gdb 只能跟踪一个进程。可以在 fork 函数调用之前，通过指令设置 gdb 调试工具跟踪父进程或者是跟踪子进程。默认跟踪父进程。

set follow-fork-mode child 命令设置 gdb 在 fork 之后跟踪子进程。

set follow-fork-mode parent 设置跟踪父进程。

注意，一定要在 fork 函数调用之前设置才有效。

【follow_fork.c】

exec 函数族

fork 创建子进程后执行的是和父进程相同的程序（但有可能执行不同的代码分支），子进程往往要调用一种 exec 函数以执行另一个程序。当进程调用一种 exec 函数时，该进程的用户空间代码和数据完全被新程序替换，从新程序的启动例程开始执行。调用 exec 并不创建新进程，所以调用 exec 前后该进程的 id 并未改变。

将当前进程的.text、.data 替换为所要加载的程序的.text、.data，然后让进程从新的.text 第一条指令开始执行，但进程 ID 不变，换核不换壳。

其实有六种以 exec 开头的函数，统称 exec 函数：

```
int execl(const char *path, const char *arg, ...);

int execlp(const char *file, const char *arg, ...);

int execle(const char *path, const char *arg, ..., char *const envp[]);

int execv(const char *path, char *const argv[]);

int execvp(const char *file, char *const argv[]);

int execve(const char *path, char *const argv[], char *const envp[]);
```

execlp 函数

加载一个进程，借助 PATH 环境变量

int execlp(const char *file, const char *arg, ...); 成功：无返回；失败：-1

参数 1：要加载的程序的名字。该函数需要配合 PATH 环境变量来使用，当 PATH 中所有目录搜索后没有参数 1 则出错返回。

该函数通常用来调用系统程序。如：ls、date、cp、cat 等命令。

execl 函数

加载一个进程， 通过 路径+程序名 来加载。

int execl(const char *path, const char *arg, ...); 成功：无返回；失败：-1

对比 execlp，如加载"ls"命令带有-l，-F 参数

execlp("ls", "ls", "-l", "-F", NULL); 使用程序名在 PATH 中搜索。

execl("/bin/ls", "ls", "-l", "-F", NULL); 使用参数 1 给出的绝对路径搜索。

execvp 函数

加载一个进程，使用自定义环境变量 env

int execvp(const char *file, const char *argv[]);

变参形式： ①... ② argv[] (main 函数也是变参函数，形式上等同于 int main(int argc, char *argv0, ...))

变参终止条件：① NULL 结尾 ② 固参指定

execvp 与 execlp 参数形式不同，原理一致。

练习：将当前系统中的进程信息，打印到文件中。

【exec_ps.c】

exec 函数族一般规律

exec 函数一旦调用成功即执行新的程序，不返回。**只有失败才返回，错误值-1**。所以通常我们直接在 exec 函数调用后直接调用 perror()和 exit()，无需 if 判断。

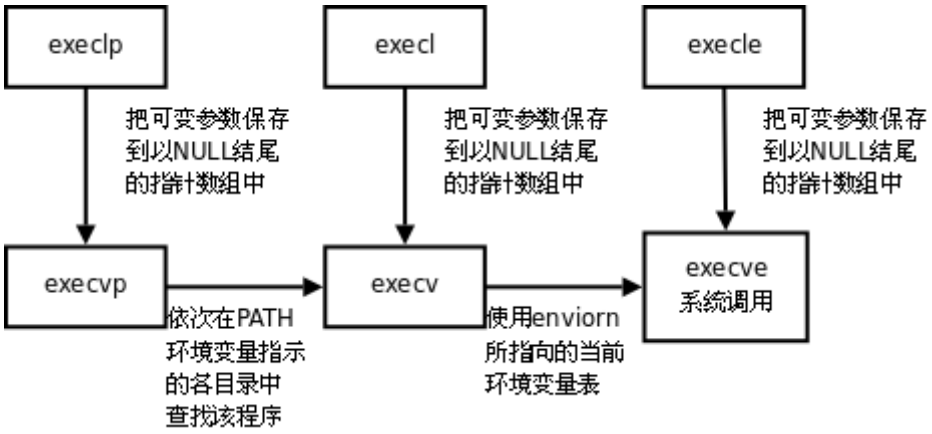
l (list) 命令行参数列表

p (path) 搜索 file 时使用 path 变量

v (vector) 使用命令行参数数组

e (environment) 使用环境变量数组,不使用进程原有的环境变量，设置新加载程序运行的环境变量

事实上，只有 execve 是真正的系统调用，其它五个函数最终都调用 execve，所以 execve 在 man 手册第 2 节，其它函数在 man 手册第 3 节。这些函数之间的关系如下图所示。



exec 函数族

回收子进程

孤儿进程

孤儿进程: 父进程先于子进程结束，则子进程成为孤儿进程，子进程的父进程成为 init 进程，称为 init 进程领养孤儿进程。

【orphan.c】

僵尸进程

僵尸进程: 进程终止，父进程尚未回收，子进程残留资源（PCB）存放于内核中，变成僵尸（Zombie）进程。

【zoom.c】

特别注意，僵尸进程是不能使用 kill 命令清除掉的。因为 kill 命令只是用来终止进程的，而僵尸进程已经终止。思考！用什么办法可清除掉僵尸进程呢？

wait 函数

一个进程在终止时会关闭所有文件描述符，释放在用户空间分配的内存，但它的 PCB 还保留着，内核在其中保存了一些信息：如果是正常终止则保存着退出状态，如果是异常终止则保存着导致该进程终止的信号是哪个。这个进程的父进程可以调用 wait 或 waitpid 获取这些信息，然后彻底清除掉这个进程。我们知道一个进程的退出状态可以在 Shell 中用特殊变量$?查看，因为 Shell 是它的父进程，当它终止时 Shell 调用 wait 或 waitpid 得到它的退出状态同时彻底清除掉这个进程。

父进程调用 wait 函数可以回收子进程终止信息。该函数有三个功能：

① 阻塞等待子进程退出

② 回收子进程残留资源

③ 获取子进程结束状态(退出原因)。

pid_t wait(int *status); 成功：清理掉的子进程 ID；失败：-1 (没有子进程)

当进程终止时，操作系统的隐式回收机制会：1.关闭所有文件描述符 2. 释放用户空间分配的内存。内核的 PCB 仍存在。其中保存该进程的退出状态。(正常终止→退出值；异常终止→终止信号)

可使用 wait 函数传出参数 status 来保存进程的退出状态。借助宏函数来进一步判断进程终止的具体原因。宏函数可分为如下三组：

1. WIFEXITED(status) 为非 0 → 进程正常结束

   WEXITSTATUS(status) 如上宏为真，使用此宏 → 获取进程退出状态 (exit 的参数)

2. WIFSIGNALED(status) 为非 0 → 进程异常终止

   WTERMSIG(status) 如上宏为真，使用此宏 → 取得使进程终止的那个信号的编号。

*3. WIFSTOPPED(status) 为非 0 → 进程处于暂停状态

   WSTOPSIG(status) 如上宏为真，使用此宏 → 取得使进程暂停的那个信号的编号。

   WIFCONTINUED(status) 为真 → 进程暂停后已经继续运行

【

w
a
i
t
1
.
c
、
w
a

i
t
2
.
c
】

waitpid 函数

作用同 wait，但可指定 pid 进程清理，可以不阻塞。

pid_t waitpid(pid_t pid, int *status, in options); 成功：返回清理掉的子进程 ID；失败：-1(无子进程)

特殊参数和返回情况：

参数 pid：

> 0 回收指定 ID 的子进程

-1 回收任意子进程（相当于 wait）

0 回收和当前调用 waitpid 一个组的所有子进程

< -1 回收指定进程组内的任意子进程

返回 0：参 3 为 WNOHANG，且子进程正在运行。

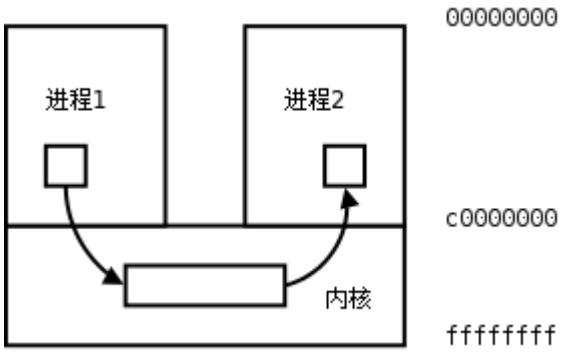
注意：**一次 wait 或 waitpid 调用只能清理一个子进程**，清理多个子进程应使用循环。

【waitpid.c】

作业：父进程 fork 3 个子进程，三个子进程一个调用 ps 命令， 一个调用自定义程序 1(正常)，一个调用自定义程序 2(会出段错误)。父进程使用 waitpid 对其子进程进行回收。

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。

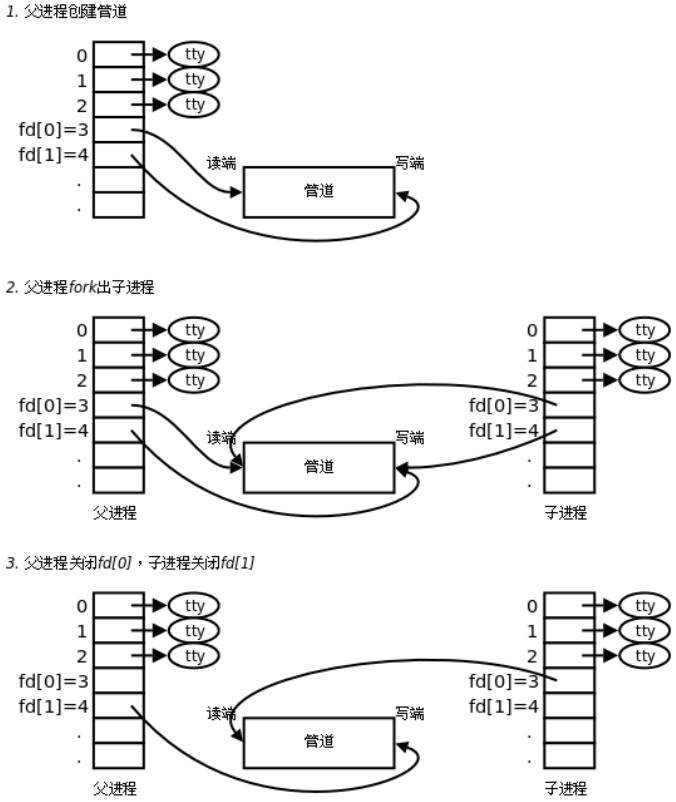
pipe 函数

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】

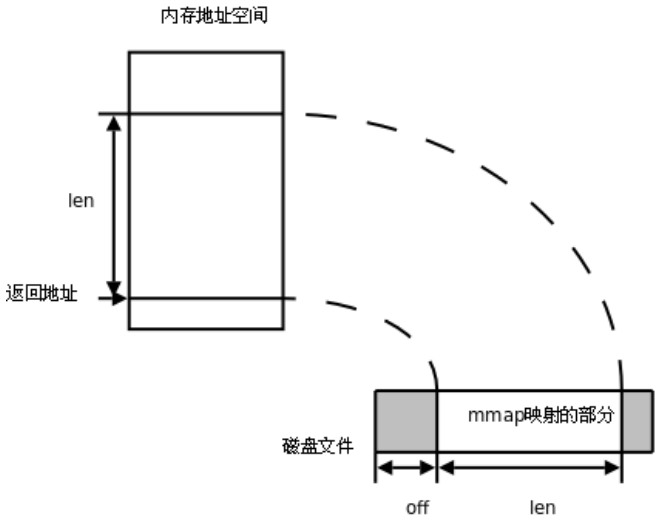
思考，无血缘关系的进程可以打开同一个文件进行通信吗？为什么？

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

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)

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.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】

信号的概念

信号在我们的生活中随处可见， 如：古代战争中摔杯为号；现代战争中的信号弹；体育比赛中使用的信号枪.....他们都有共性：1. 简单 2. 不能携带大量信息 3. 满足某个特设条件才发送。

信号是信息的载体，Linux/UNIX 环境下，古老、经典的通信方式， 现下依然是主要的通信手段。

Unix 早期版本就提供了信号机制，但不可靠，信号可能丢失。Berkeley 和 AT&T 都对信号模型做了更改，增加了可靠信号机制。但彼此不兼容。POSIX.1 对可靠信号例程进行了标准化。

信号的机制

A 给 B 发送信号，B 收到信号之前执行自己的代码，收到信号后，不管执行到程序的什么位置，都要暂停运行，去处理信号，处理完毕再继续执行。与硬件中断类似——异步模式。但信号是软件层面上实现的中断，早期常被称为“软中断”。

信号的特质：由于信号是通过软件方法实现，其实现手段导致信号有很强的延时性。但对于用户来说，这个延迟时间非常短，不易察觉。

每个进程收到的所有信号，都是由内核负责发送的，内核处理。

与信号相关的事件和状态

产生信号:

1. 按键产生，如：Ctrl+c、Ctrl+z、Ctrl+\
2. 系统调用产生，如：kill、raise、abort
3. 软件条件产生，如：定时器 alarm
4. 硬件异常产生，如：非法访问内存(段错误)、除 0(浮点数例外)、内存对齐出错(总线错误)
5. 命令产生，如：kill 命令

递达：递送并且到达进程。

未决：产生和递达之间的状态。主要由于阻塞(屏蔽)导致该状态。

信号的处理方式:

1. 执行默认动作
2. 忽略(丢弃)
3. 捕捉(调用户处理函数)

Linux 内核的进程控制块 PCB 是一个结构体，task_struct, 除了包含进程 id，状态，工作目录，用户 id，组 id，文件描述符表，还包含了信号相关的信息，主要指阻塞信号集和未决信号集。

阻塞信号集(信号屏蔽字)： 将某些信号加入集合，对他们设置屏蔽，当屏蔽 x 信号后，再收到该信号，该信号的处理将推后(解除屏蔽后)

未决信号集:

1. 信号产生，未决信号集中描述该信号的位立刻翻转为 1，表信号处于未决状态。当信号被处理对应位翻转回为 0。这一时刻往往非常短暂。

2. 信号产生后由于某些原因(主要是阻塞)不能抵达。这类信号的集合称之为未决信号集。在屏蔽解除前，信号一直处于未决状态。

信号的编号

可以使用 kill –l 命令查看当前系统可使用的信号有哪些。

1) SIGHUP 2) SIGINT 3) SIGQUIT 4) SIGILL 5) SIGTRAP
6) SIGABRT 7) SIGBUS 8) SIGFPE 9) SIGKILL 10) SIGUSR1
11) SIGSEGV 12) SIGUSR2 13) SIGPIPE 14) SIGALRM 15) SIGTERM
16) SIGSTKFLT 17) SIGCHLD 18) SIGCONT 19) SIGSTOP 20) SIGTSTP
21) SIGTTIN 22) SIGTTOU 23) SIGURG 24) SIGXCPU 25) SIGXFSZ
26) SIGVTALRM 27) SIGPROF 28) SIGWINCH 29) SIGIO 30) SIGPWR
31) SIGSYS 34) SIGRTMIN 35) SIGRTMIN+1 36) SIGRTMIN+2 37) SIGRTMIN+3
38) SIGRTMIN+4 39) SIGRTMIN+5 40) SIGRTMIN+6 41) SIGRTMIN+7 42) SIGRTMIN+8
43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9 56) SIGRTMAX-8 57) SIGRTMAX-7
58) SIGRTMAX-6 59) SIGRTMAX-5 60) SIGRTMAX-4 61) SIGRTMAX-3 62) SIGRTMAX-2
63) SIGRTMAX-1 64) SIGRTMAX

不存在编号为 0 的信号。其中 1-31 号信号称之为常规信号（也叫普通信号或标准信号），34-64 称之为实时信号，驱动编程与硬件相关。名字上区别不大。而前 32 个名字各不相同。

信号 4 要素

与变量三要素类似的，每个信号也有其必备 4 要素，分别是：

1. 编号 2. 名称 3. 事件 4. 默认处理动作

可通过 man 7 signal 查看帮助文档获取。也可查看/usr/src/linux-headers-3.16.0-30/arch/s390/include/uapi/asm/signal.h

Signal	Value	Action	Comment
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers
SIGALRM	14	Term	Timer signal from alarm(2)
SIGTERM	15	Term	Termination signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2

SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25	Cont	Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,24	Stop	Stop typed at terminal
SIGTTIN	21,21,26	Stop	Terminal input for background process
SIGTTOU	22,22,27	Stop	Terminal output for background process

The signals SIGKILL and SIGSTOP cannot be caught, blocked, or ignored.

在标准信号中，有一些信号是有三个“Value”，第一个值通常对 alpha 和 sparc 架构有效，中间值针对 x86、arm 和其他架构，最后一个应用于 mips 架构。一个‘-’表示在对应架构上尚未定义该信号。

不同的操作系统定义了不同的系统信号。因此有些信号出现在 Unix 系统内，也出现在 Linux 中，而有的信号出现在 FreeBSD 或 Mac OS 中却没有出现在 Linux 下。这里我们只研究 Linux 系统中的信号。

默认动作：

Term：终止进程

Ign： 忽略信号 (默认即时对该种信号忽略操作)

Core：终止进程，生成 Core 文件。(查验进程死亡原因， 用于 gdb 调试)

Stop：停止（暂停）进程

Cont：继续运行进程

注意从 man 7 signal 帮助文档中可看到 : The signals SIGKILL and SIGSTOP cannot be caught, blocked, or ignored.

这里特别强调了 **9) SIGKILL 和 19) SIGSTOP 信号，不允许忽略和捕捉，只能执行默认动作。甚至不能将其设置为阻塞。**

另外需清楚，只有每个信号所对应的事件发生了，该信号才会被递送(但不一定递达)，不应乱发信号!!

Linux 常规信号一览表

1) SIGHUP: 当用户退出 shell 时，由该 shell 启动的所有进程将收到这个信号，默认动作为终止进程

2) SIGINT：当用户按下了<Ctrl+C>组合键时，用户终端向正在运行中的由该终端启动的程序发出此信号。默认动作为终止进程。

3) SIGQUIT：当用户按下<ctrl+\>组合键时产生该信号，用户终端向正在运行中的由该终端启动的程序发出些信号。默认动作为终止进程。

4) SIGILL：CPU 检测到某进程执行了非法指令。默认动作为终止进程并产生 core 文件

5) SIGTRAP：该信号由断点指令或其他 trap 指令产生。默认动作为终止里程 并产生 core 文件。

6) SIGABRT: 调用 abort 函数时产生该信号。默认动作为终止进程并产生 core 文件。

7) SIGBUS：非法访问内存地址，包括内存对齐出错，默认动作为终止进程并产生 core 文件。

8) SIGFPE：在发生致命的运算错误时发出。不仅包括浮点运算错误，还包括溢出及除数为 0 等所有的算法错误。默认动作为终止进程并产生 core 文件。

9) SIGKILL：无条件终止进程。本信号不能被忽略，处理和阻塞。默认动作为终止进程。它向系统管理员提供了可以杀死任何进程的方法。

10) SIGUSE1：用户定义 的信号。即程序员可以在程序中定义并使用该信号。默认动作为终止进程。

11) SIGSEGV：指示进程进行了无效内存访问。默认动作为终止进程并产生 core 文件。

12) SIGUSR2：另外一个用户自定义信号，程序员可以在程序中定义并使用该信号。默认动作为终止进程。

13) SIGPIPE：Broken pipe 向一个没有读端的管道写数据。默认动作为终止进程。

14) SIGALRM: 定时器超时，超时的时间 由系统调用 alarm 设置。默认动作为终止进程。

15) SIGTERM：程序结束信号，与 SIGKILL 不同的是，该信号可以被阻塞和终止。通常用来要示程序正常退出。执行 shell 命令 Kill 时，缺省产生这个信号。默认动作为终止进程。

16) SIGSTKFLT：Linux 早期版本出现的信号，现仍保留向后兼容。默认动作为终止进程。

17) SIGCHLD：子进程状态发生变化时，父进程会收到这个信号。默认动作为**忽略**这个信号。

18) SIGCONT：如果进程已停止，则使其继续运行。默认动作为继续/忽略。

19) SIGSTOP：停止进程的执行。信号不能被忽略，处理和阻塞。默认动作为暂停进程。

20) SIGTSTP：停止终端交互进程的运行。按下<ctrl+z>组合键时发出这个信号。默认动作为暂停进程。

21) SIGTTIN：后台进程读终端控制台。默认动作为暂停进程。

22) SIGTTOU: 该信号类似于 SIGTTIN，在后台进程要向终端输出数据时发生。默认动作为暂停进程。

23) SIGURG：套接字上有紧急数据时，向当前正在运行的进程发出些信号，报告有紧急数据到达。如网络带外数据到达，默认动作为忽略该信号。

24) SIGXCPU：进程执行时间超过了分配给该进程的 CPU 时间 ，系统产生该信号并发送给该进程。默认动作为终止进程。

25) SIGXFSZ：超过文件的最大长度设置。默认动作为终止进程。

26) SIGVTALRM：虚拟时钟超时时产生该信号。类似于 SIGALRM，但是该信号只计算该进程占用 CPU 的使用时间。默认动作为终止进程。

27) SGIPROF：类似于 SIGVTALRM，它不公包括该进程占用 CPU 时间还包括执行系统调用时间。默认动作为终止进程。

28) SIGWINCH：窗口变化大小时发出。默认动作为忽略该信号。

29) SIGIO：此信号向进程指示发出了一个异步 IO 事件。默认动作为忽略。

30) SIGPWR：关机。默认动作为终止进程。

31) SIGSYS：无效的系统调用。默认动作为终止进程并产生 core 文件。

34) SIGRTMIN ～ (64) SIGRTMAX：LINUX 的实时信号，它们没有固定的含义（可以由用户自定义）。所有的实时信号的默认动作都为终止进程。

信号的产生

终端按键产生信号

Ctrl + c → 2) SIGINT（终止/中断） "INT" ----Interrupt

Ctrl + z → 20) SIGTSTP（暂停/停止） "T" ----Terminal 终端。

Ctrl + \ → 3) SIGQUIT（退出）

硬件异常产生信号

除 0 操作 → 8) SIGFPE (浮点数例外) "F" -----float 浮点数。

非法访问内存 → 11) SIGSEGV (段错误)

总线错误 → 7) SIGBUS

kill 函数/命令产生信号

kill 命令产生信号：kill -SIGKILL pid

kill 函数：给指定进程发送指定信号(不一定杀死)

int kill(pid_t pid, int sig); 成功：0；失败：-1 (ID 非法，信号非法，普通用户杀 init 进程等权级问题)，设置 errno

sig：不推荐直接使用数字，应使用宏名，因为不同操作系统信号编号可能不同，但名称一致。

pid > 0: 发送信号给指定的进程。

pid = 0: 发送信号给 与调用 kill 函数进程属于同一进程组的所有进程。

pid < 0: 取|pid|发给对应进程组。

pid = -1：发送给进程有权限发送的系统中所有进程。

进程组：每个进程都属于一个进程组，进程组是一个或多个进程集合，他们相互关联，共同完成一个实体任务，每个进程组都有一个进程组长，默认进程组 ID 与进程组长 ID 相同。

权限保护：super 用户(root)可以发送信号给任意用户，普通用户是不能向系统用户发送信号的。 kill -9 (root 用户的 pid) 是不可以的。同样，普通用户也不能向其他普通用户发送信号，终止其进程。 只能向自己创建的进程发送信号。普通用户基本规则是：发送者实际或有效用户 ID == 接收者实际或有效用户 ID

练习：循环创建 5 个子进程，父进程用 kill 函数终止任一子进程。 【kill.c】

软件条件产生信号

alarm 函数

设置定时器(闹钟)。在指定 seconds 后，内核会给当前进程发送 14）SIGALRM 信号。进程收到该信号，默认动作终止。

每个进程都有且只有唯一个定时器。

unsigned int alarm(unsigned int seconds); 返回 0 或剩余的秒数，无失败。

常用：取消定时器 alarm(0)，返回旧闹钟余下秒数。

例：alarm(5) → 3sec → alarm(4) → 5sec → alarm(5) → alarm(0)

定时，与进程状态无关(自然定时法)！就绪、运行、挂起(阻塞、暂停)、终止、僵尸...无论进程处于何种状态，alarm 都计时。

练习：编写程序，测试你使用的计算机 1 秒钟能数多少个数。 【alarm .c】

使用 time 命令查看程序执行的时间。 程序运行的瓶颈在于 IO，优化程序，首选优化 IO。

实际执行时间 = 系统时间 + 用户时间 + 等待时间

setitimer 函数

设置定时器(闹钟)。 可代替 alarm 函数。精度微秒 us，可以实现周期定时。

int setitimer(int which, const struct itimerval *new_value, struct itimerval *old_value); 成功：0；失败：-1，设置 errno

参数：which：指定定时方式

① 自然定时：ITIMER_REAL → 14）SIGLARM　　计算自然时间

② 虚拟空间计时(用户空间)：ITIMER_VIRTUAL → 26）SIGVTALRM　　只计算进程占用 cpu 的时间

③ 运行时计时(用户+内核)：ITIMER_PROF → 27）SIGPROF　　计算占用 cpu 及执行系统调用的时间

练习：使用 setitimer 函数实现 alarm 函数，重复计算机 1 秒数数程序。　　【setitimer_alarm.c】

拓展练习，结合 man page 编写程序，测试 it_interval、it_value 这两个参数的作用。　　【setitimer_cycle.c】

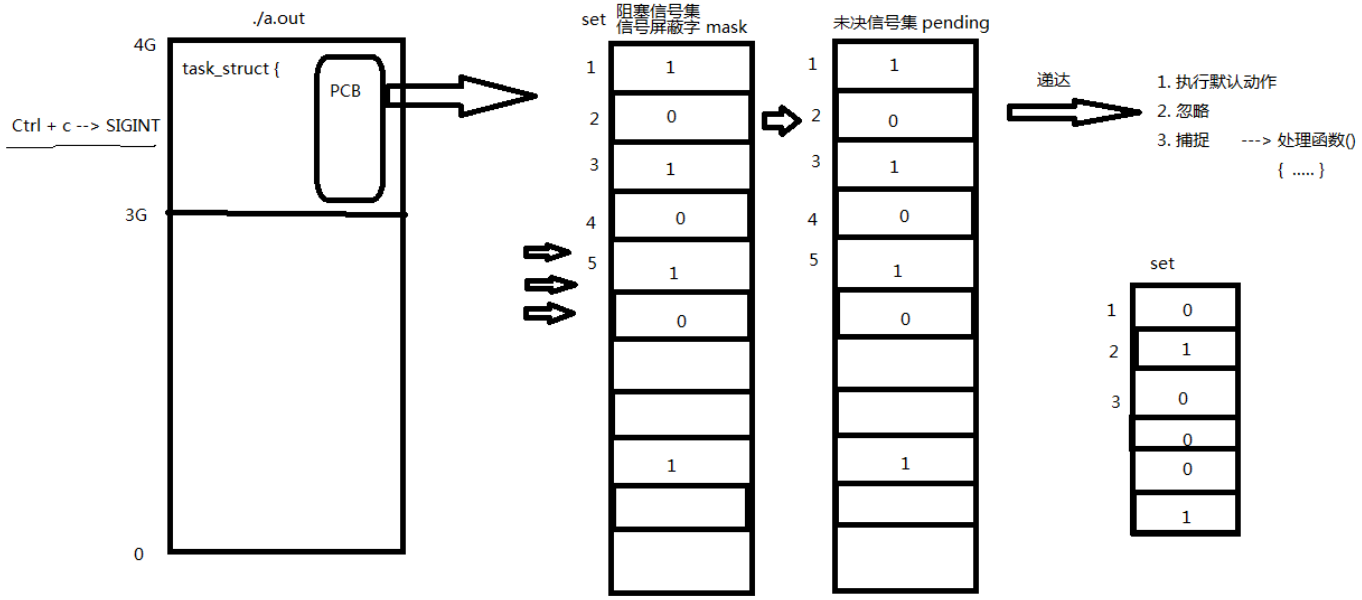
提示：　it_interval：用来设定两次定时任务之间间隔的时间。

it_value：定时的时长

两个参数都设置为 0，即清 0 操作。

信号集操作函数

内核通过读取未决信号集来判断信号是否应被处理。信号屏蔽字 mask 可以影响未决信号集。而我们可以在应用程序中自定义 set 来改变 mask。已达到屏蔽指定信号的目的。



信号集设定

sigset_t　set;　　// typedef unsigned long sigset_t;

int sigemptyset(sigset_t *set);　　将某个信号集清 0　　成功：0；失败：-1

int sigfillset(sigset_t *set);　　将某个信号集置 1　　成功：0；失败：-1

int sigaddset(sigset_t *set, int signum);　　将某个信号加入信号集　　成功：0；失败：-1

int sigdelset(sigset_t *set, int signum);　　将某个信号清出信号集　　成功：0；失败：-1

int sigismember(const sigset_t *set, int signum);判断某个信号是否在信号集中　返回值：在集合：1；不在：0；出错：-1

sigset_t 类型的本质是位图。但不应该直接使用位操作，而应该使用上述函数，保证跨系统操作有效。

对比认知 select 函数。

sigprocmask 函数

用来屏蔽信号、解除屏蔽也使用该函数。其本质，读取或修改进程的信号屏蔽字(PCB 中)

严格注意，屏蔽信号：只是将信号处理延后执行(延至解除屏蔽)；而忽略表示将信号丢处理。

int sigprocmask(int how, const sigset_t *set, sigset_t *oldset); 成功：0；失败：-1，设置 errno

参数：

set：传入参数，是一个位图，set 中哪位置 1，就表示当前进程屏蔽哪个信号。

oldset：传出参数，保存旧的信号屏蔽集。

how 参数取值： 假设当前的信号屏蔽字为 mask

1. SIG_BLOCK: 当 how 设置为此值，set 表示需要屏蔽的信号。相当于 mask = mask|set
2. SIG_UNBLOCK: 当 how 设置为此，set 表示需要解除屏蔽的信号。相当于 mask = mask & ~set
3. SIG_SETMASK: 当 how 设置为此，set 表示用于替代原始屏蔽及的新屏蔽集。相当于 mask = set 若，调用 sigprocmask 解除了对当前若干个信号的阻塞，则在 sigprocmask 返回前，至少将其中一个信号递达。

sigpending 函数

读取当前进程的**未决**信号集

int sigpending(sigset_t *set); set 传出参数。 返回值：成功：0；失败：-1，设置 errno

练习：编写程序。把所有常规信号的未决状态打印至屏幕。 【sigpending.c】

信号捕捉

signal 函数

注册一个信号捕捉函数：

typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);

该函数由 ANSI 定义，由于历史原因在不同版本的 Unix 和不同版本的 Linux 中可能有不同的行为。因此应该尽量避免使用它，取而代之使用 sigaction 函数。

void (*signal(int signum, void (*sighandler_t)(int))) (int);

能看出这个函数代表什么意思吗？ 注意多在复杂结构中使用 typedef。

sigaction 函数

修改信号处理动作（通常在 Linux 用其来注册一个信号的捕捉函数）

int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);　成功：0；失败：-1，设置 errno

参数：

act：传入参数，新的处理方式。

oldact：传出参数，旧的处理方式。　【signal.c】

struct sigaction 结构体

```
struct sigaction {
    void     (*sa_handler)(int);
    void     (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t   sa_mask;
    int        sa_flags;
    void     (*sa_restorer)(void);
};
```

sa_restorer：该元素是过时的，不应该使用，POSIX.1 标准将不指定该元素。(弃用)

sa_sigaction：当 sa_flags 被指定为 SA_SIGINFO 标志时，使用该信号处理程序。(很少使用)

重点掌握：

① sa_handler：指定信号捕捉后的处理函数名(即注册函数)。也可赋值为 SIG_IGN 表忽略 或 SIG_DFL 表执行默认动作

② sa_mask: 调用信号处理函数时，所要屏蔽的信号集合(信号屏蔽字)。注意：仅在处理函数被调用期间屏蔽生效，是临时性设置。

③ sa_flags：通常设置为 0，表使用默认属性。

信号捕捉特性

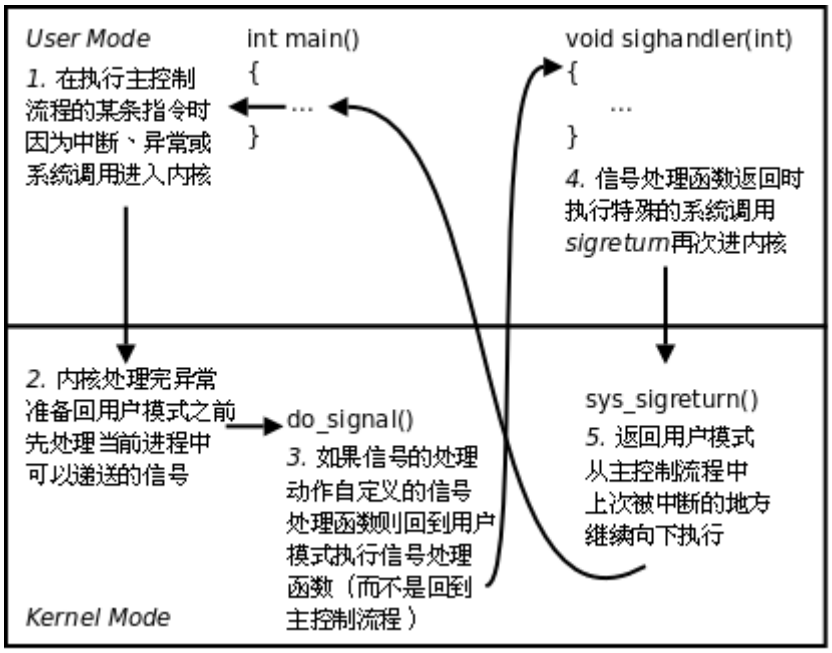
1. 进程正常运行时，默认 PCB 中有一个信号屏蔽字，假定为☆，它决定了进程自动屏蔽哪些信号。当注册了某个信号捕捉函数，捕捉到该信号以后，要调用该函数。而该函数有可能执行很长时间，在这期间所屏蔽的信号不由☆来指定。而是用 sa_mask 来指定。调用完信号处理函数，再恢复为☆。
2. XXX 信号捕捉函数执行期间，XXX 信号自动被屏蔽。
3. 阻塞的常规信号不支持排队，产生多次只记录一次。（后 32 个实时信号支持排队）

练习 1：为某个信号设置捕捉函数　【sigaction1.c】

练习 2：验证在信号处理函数执行期间，该信号多次递送，那么只在处理函数之行结束后，处理一次。【sigaction2.c】

练习 3：验证 sa_mask 在捕捉函数执行期间的屏蔽作用。　【sigaction3.c】

内核实现信号捕捉过程：



SIGCHLD 信号

SIGCHLD 的产生条件

子进程终止时

子进程接收到 SIGSTOP 信号停止时

子进程处在停止态，接受到 SIGCONT 后唤醒时

借助 SIGCHLD 信号回收子进程

子进程结束运行，其父进程会收到 SIGCHLD 信号。该信号的默认处理动作是忽略。可以捕捉该信号，在捕捉函数中完成子进程状态的回收。

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <signal.h>

void sys_err(char *str)
{
	perror(str);
	exit(1);
}
void do_sig_child(int signo)
```

```
{
    int status;     pid_t pid;
    while ((pid = waitpid(0, &status, WNOHANG)) > 0) {
        if (WIFEXITED(status))
            printf("child %d exit %d\n", pid, WEXITSTATUS(status));
        else if (WIFSIGNALED(status))
            printf("child %d cancel signal %d\n", pid, WTERMSIG(status));
    }
}
int main(void)
{
    pid_t pid;      int i;
    for (i = 0; i < 10; i++) {
        if ((pid = fork()) == 0)
            break;
        else if (pid < 0)
            sys_err("fork");
    }
    if (pid == 0) {
        int n = 1;
        while (n--) {
            printf("child ID %d\n", getpid());
            sleep(1);
        }
        return i+1;
    } else if (pid > 0) {
        struct sigaction act;
        act.sa_handler = do_sig_child;
        sigemptyset(&act.sa_mask);
        act.sa_flags = 0;
        sigaction(SIGCHLD, &act, NULL);

        while (1) {
            printf("Parent ID %d\n", getpid());
            sleep(1);
        }
    }
    return 0;
}
```

分析该例子。结合 17)SIGCHLD 信号默认动作，掌握父使用捕捉函数回收子进程的方式。 【sigchild.c】

如果子进程主逻辑中 sleep(1)可以吗？可不可以将程序中，捕捉函数内部的 while 替换为 if？为什么？

if ((pid = waitpid(0, &status, WNOHANG)) > 0) { ... }

思考：信号不支持排队，当正在执行 SIGCHLD 捕捉函数时，再过来一个或多个 SIGCHLD 信号怎么办？

子进程结束 status 处理方式

pid_t waitpid(pid_t pid, int *status, int options)

options：

WNOHANG

没有子进程结束，立即返回

WUNTRACED

如果子进程由于被停止产生的 SIGCHLD，waitpid 则立即返回

WCONTINUED

如果子进程由于被 SIGCONT 唤醒而产生的 SIGCHLD，waitpid 则立即返回

status：

WIFEXITED(status)

子进程正常 exit 终止，返回真

WEXITSTATUS(status)返回子进程正常退出值

WIFSIGNALED(status)

子进程被信号终止，返回真

WTERMSIG(status)返回终止子进程的信号值

WIFSTOPPED(status)

子进程被停止，返回真

WSTOPSIG(status)返回停止子进程的信号值

WIFCONTINUED(status)

SIGCHLD 信号注意问题

1. 子进程继承父进程的信号屏蔽字和信号处理动作，但子进程没有继承未决信号集 spending。
2. 注意注册信号捕捉函数的位置。
3. 应该在 fork 之前，阻塞 SIGCHLD 信号。注册完捕捉函数后解除阻塞。

中断系统调用

系统调用可分为两类：慢速系统调用和其他系统调用。

1. 慢速系统调用：可能会使进程永远阻塞的一类。如果在阻塞期间收到一个信号，该系统调用就被中断,不再继续执行(早期)；也可以设定系统调用是否重启。如，read、write、pause、wait...
2. 其他系统调用：getpid、getppid、fork...

结合 pause，回顾慢速系统调用：

慢速系统调用被中断的相关行为，实际上就是 pause 的行为： 如，read

① 想中断 pause，信号不能被屏蔽。

② 信号的处理方式必须是捕捉 (默认、忽略都不可以)

③ 中断后返回-1， 设置 errno 为 EINTR(表“被信号中断”)

可修改 sa_flags 参数来设置被信号中断后系统调用是否重启。SA_INTERRURT 不重启。 SA_RESTART 重启。

扩展了解：

sa_flags 还有很多可选参数，适用于不同情况。如：捕捉到信号后，在执行捕捉函数期间，不希望自动阻塞该信号，可将 sa_flags 设置为 SA_NODEFER，除非 sa_mask 中包含该信号。

进程组和会话

概念和特性

进程组，也称之为作业。BSD 于 1980 年前后向 Unix 中增加的一个新特性。代表一个或多个进程的集合。每个进程都属于一个进程组。在 waitpid 函数和 kill 函数的参数中都曾使用到。操作系统设计的进程组的概念，是为了简化对多个进程的管理。

当父进程，创建子进程的时候，默认子进程与父进程属于同一进程组。进程组 ID==第一个进程 ID(组长进程)。所以，组长进程标识：其进程组 ID==其进程 ID

可以使用 kill -SIGKILL -进程组 ID(负的)来将整个进程组内的进程全部杀死。 【kill_multprocess.c】

组长进程可以创建一个进程组，创建该进程组中的进程，然后终止。只要进程组中有一个进程存在，进程组就存在，与组长进程是否终止无关。

进程组生存期：进程组创建到最后一个进程离开(终止或转移到另一个进程组)。

一个进程可以为自己或子进程设置进程组 ID

创建会话

创建一个会话需要注意以下 6 点注意事项：

1. 调用进程不能是进程组组长，该进程变成新会话首进程(session header)
2. 该进程成为一个新进程组的组长进程。
3. 需有 root 权限 (ubuntu 不需要)
4. 新会话丢弃原有的控制终端，该会话没有控制终端
5. 该调用进程是组长进程，则出错返回
6. 建立新会话时，先调用 fork, 父进程终止，子进程调用 setsid()

getsid 函数

获取进程所属的会话 ID

pid_t getsid(pid_t pid); 成功：返回调用进程的会话 ID；失败：-1，设置 errno

pid 为 0 表示察看当前进程 session ID

ps ajx 命令查看系统中的进程。参数 a 表示不仅列当前用户的进程，也列出所有其他用户的进程，参数 x 表示不仅列有控制终端的进程，也列出所有无控制终端的进程，参数 j 表示列出与作业控制相关的信息。

组长进程不能成为新会话首进程，新会话首进程必定会成为组长进程。

setsid 函数

创建一个会话，并以自己的 ID 设置进程组 ID，同时也是新会话的 ID。

pid_t setsid(void); 成功：返回调用进程的会话 ID；失败：-1，设置 errno

调用了 setsid 函数的进程，既是新的会长，也是新的组长。

练习：fork 一个子进程，并使其创建一个新会话。查看进程组 ID、会话 ID 前后变化 【session.c】

守护进程

Daemon(精灵)进程，是 Linux 中的后台服务进程，通常独立于控制终端并且周期性地执行某种任务或等待处理某些发生的事件。一般采用以 d 结尾的名字。

Linux 后台的一些系统服务进程，没有控制终端，不能直接和用户交互。不受用户登录、注销的影响，一直在运行着，他们都是守护进程。如：预读入缓输出机制的实现；ftp 服务器；nfs 服务器等。

创建守护进程，最关键的一步是调用 setsid 函数创建一个新的 Session，并成为 Session Leader。

创建守护进程模型

1. 创建子进程，父进程退出

   所有工作在子进程中进行形式上脱离了控制终端

2. 在子进程中创建新会话

   setsid()函数

   使子进程完全独立出来，脱离控制

3. 改变当前目录位置

   chdir()函数

   防止占用可卸载的文件系统

   也可以换成其它路径

4. 重设文件权限掩码

   umask()函数

   防止继承的文件创建屏蔽字拒绝某些权限

   增加守护进程灵活性

5. 关闭文件描述符

   继承的打开文件不会用到，浪费系统资源，无法卸载

6. 开始执行守护进程核心工作守护进程退出处理程序模型

线程概念

什么是线程

LWP：light weight process 轻量级的进程，本质仍是进程(在 Linux 环境下)

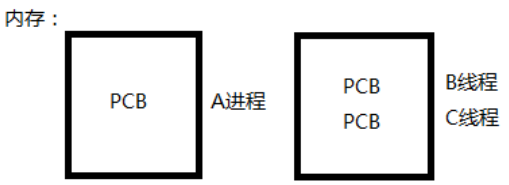
进程：独立地址空间，拥有 PCB

线程：有独立的 PCB，但没有独立的地址空间(共享)

区别：在于是否共享地址空间。　　独居(进程)；合租(线程)。

Linux 下：　　线程：最小的执行单位

　　进程：最小分配资源单位，可看成是只有一个线程的进程。

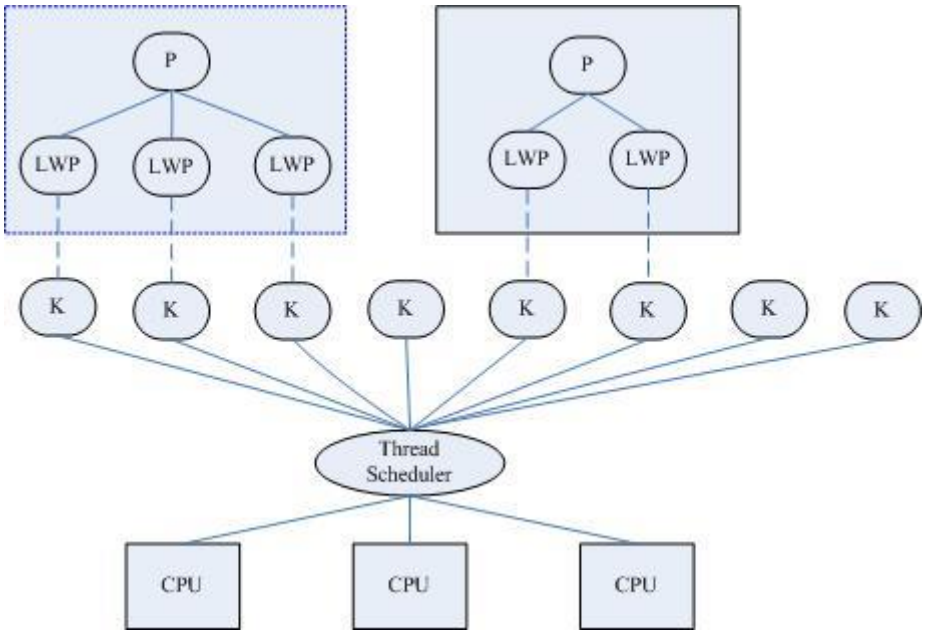


Linux 内核线程实现原理

类 Unix 系统中，早期是没有“线程”概念的，80 年代才引入，借助进程机制实现出了线程的概念。因此在这类系统中，进程和线程关系密切。

1. 轻量级进程(light-weight process)，也有 PCB，创建线程使用的底层函数和进程一样，都是 clone
2. 从内核里看进程和线程是一样的，都有各自不同的 PCB，但是 PCB 中指向内存资源的三级页表是相同的
3. 进程可以蜕变成线程
4. 线程可看做寄存器和栈的集合
5. 在 linux 下，线程最是小的执行单位；进程是最小的分配资源单位

察看 LWP 号：ps –Lf pid 查看指定线程的 lwp 号。



三级映射：进程 PCB --> 页目录(可看成数组，首地址位于 PCB 中) --> 页表 --> 物理页面 --> 内存单元

参考：《Linux 内核源代码情景分析》 ----毛德操

对于进程来说，相同的地址(同一个虚拟地址)在不同的进程中，反复使用而不冲突。原因是他们虽虚拟址一样，但，页目录、页表、物理页面各不相同。相同的虚拟址，映射到不同的物理页面内存单元，最终访问不同的物理页面。

但！线程不同！两个线程具有各自独立的 PCB，但共享同一个页目录，也就共享同一个页表和物理页面。所以两个 PCB 共享一个地址空间。

实际上，无论是创建进程的 fork，还是创建线程的 pthread_create，底层实现都是调用同一个内核函数 clone。

如果复制对方的地址空间，那么就产出一个“进程”；如果共享对方的地址空间，就产生一个“线程”。

因此：Linux 内核是不区分进程和线程的。只在用户层面上进行区分。所以，线程所有操作函数 pthread_* 是库函数，而非系统调用。

线程共享资源

1.文件描述符表

2.每种信号的处理方式

3.当前工作目录

4.用户 ID 和组 ID

5.内存地址空间 (.text/.data/.bss/heap/共享库)

线程非共享资源

1.线程 id

2.处理器现场和栈指针(内核栈)

3.独立的栈空间(用户空间栈)

4.errno 变量

5.信号屏蔽字

6.调度优先级

线程优、缺点

优点： 1. 提高程序并发性 2. 开销小 3. 数据通信、共享数据方便

缺点： 1. 库函数，不稳定 2. 调试、编写困难、gdb 不支持 3. 对信号支持不好

优点相对突出，缺点均不是硬伤。Linux 下由于实现方法导致进程、线程差别不是很大。

线程控制原语

pthread_self 函数

获取线程 ID。其作用对应进程中 getpid() 函数。

pthread_t pthread_self(void);返回值：成功：0； 失败：无！

线程 ID：pthread_t 类型，本质：在 Linux 下为无符号整数(%lu)，其他系统中可能是结构体实现

线程 ID 是进程内部，识别标志。(两个进程间，线程 ID 允许相同)

注意：不应使用全局变量 pthread_t tid，在子线程中通过 pthread_create 传出参数来获取线程 ID，而应使用 pthread_self。

pthread_create 函数

创建一个新线程。 其作用，对应进程中 fork() 函数。

int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg);

返回值：成功：0； 失败：错误号 -----Linux 环境下，所有线程特点，失败均直接返回错误号。

参数：

pthread_t：当前 Linux 中可理解为：typedef unsigned long int pthread_t;

参数 1：传出参数，保存系统为我们分配好的线程 ID

参数 2：通常传 NULL，表示使用线程默认属性。若想使用具体属性也可以修改该参数。

参数 3：函数指针，指向线程主函数(线程体)，该函数运行结束，则线程结束。

参数 4：线程主函数执行期间所使用的参数。

在一个线程中调用 pthread_create()创建新的线程后，当前线程从 pthread_create()返回继续往下执行，而新的线程所执行的代码由我们传给 pthread_create 的函数指针 start_routine 决定。start_routine 函数接收一个参数，是通过 pthread_create 的 arg 参数传递给它的，该参数的类型为 void *，这个指针按什么类型解释由调用者自己定义。start_routine 的返回值类型也是 void *，这个指针的含义同样由调用者自己定义。start_routine 返回时，这个线程就退出了，其它线程可以调用 pthread_join 得到 start_routine 的返回值，类似于父进程调用 wait(2)得到子进程的退出状态，稍后详细介绍 pthread_join。

pthread_create 成功返回后，新创建的线程的 id 被填写到 thread 参数所指向的内存单元。我们知道进程 id 的类型是 pid_t，每个进程的 id 在整个系统中是唯一的，调用 getpid(2)可以获得当前进程的 id，是一个正整数值。线程 id 的类型是 thread_t，它只在当前进程中保证是唯一的，在不同的系统中 thread_t 这个类型有不同的实现，它可能是一个整数值，也可能是一个结构体，也可能是一个地址，所以不能简单地当成整数用 printf 打印，调用 pthread_self(3)可以获得当前线程的 id。

attr 参数表示线程属性，本节不深入讨论线程属性，所有代码例子都传 NULL 给 attr 参数，表示线程属性取缺省值，感兴趣的读者可以参考 APUE。

【练习】：创建一个新线程，打印线程 ID。注意：链接线程库 -lpthread

【pthrd_crt.c】

由于 pthread_create 的错误码不保存在 errno 中，因此不能直接用 perror(3)打印错误信息，可以先用 strerror(3)把错误码转换成错误信息再打印。如果任意一个线程调用了 exit 或_exit，则整个进程的所有线程都终止，由于从 main 函数 return 也相当于调用 exit，为了防止新创建的线程还没有得到执行就终止，我们在 main 函数 return 之前延时 1 秒，这只是一种权宜之计，即使主线程等待 1 秒，内核也不一定会调度新创建的线程执行，下一节我们会看到更好的办法。

【练习】：循环创建多个线程，每个线程打印自己是第几个被创建的线程。(类似于进程循环创建子进程)
【more_pthrd.c】

拓展思考：将 pthread_create 函数参 4 修改为(void *)&i, 将线程主函数内改为 i=*((int *)arg) 是否可以？

线程与共享

线程间共享全局变量！

【牢记】：线程默认共享数据段、代码段等地址空间，常用的是全局变量。而进程不共享全局变量，只能借助 mmap。

【练习】：设计程序，验证线程之间共享全局数据。 【glb_var_pthrd.c】

pthread_exit 函数

将单个线程退出

void pthread_exit(void *retval); 参数：retval 表示线程退出状态，通常传 NULL

思考：使用 exit 将指定线程退出，可以吗？ 【pthrd_exit.c】

结论：线程中，禁止使用 exit 函数，会导致进程内所有线程全部退出。

在不添加 sleep 控制输出顺序的情况下。pthread_create 在循环中，几乎瞬间创建 5 个线程，但只有第 1 个线程有机会输出（或者第 2 个也有，也可能没有，取决于内核调度）如果第 3 个线程执行了 exit，将整个进程退出了，所以全部线程退出了。

所以，多线程环境中，应尽量少用，或者不使用 exit 函数，取而代之使用 pthread_exit 函数，将单个线程退出。任何线程里 exit 导致进程退出，其他线程未工作结束，主控线程退出时不能 return 或 exit。

另注意，pthread_exit 或者 return 返回的指针所指向的内存单元必须是全局的或者是用 malloc 分配的，不能在线程函数的栈上分配，因为当其它线程得到这个返回指针时线程函数已经退出了。

【练习】：编写多线程程序，总结 exit、return、pthread_exit 各自退出效果。

return：返回到调用者那里去。

pthread_exit()：将调用该函数的线程退出

exit: 将进程退出。

pthread_join 函数

阻塞等待线程退出，获取线程退出状态 其作用，对应进程中 waitpid() 函数。

int pthread_join(pthread_t thread, void **retval); 成功：0；失败：错误号

参数：thread：线程 ID （【注意】：不是指针）；retval：存储线程结束状态。

对比记忆：

进程中：main 返回值、exit 参数-->int；等待子进程结束 wait 函数参数-->int *

线程中：线程主函数返回值、pthread_exit-->void *；等待线程结束 pthread_join 函数参数-->void **

【练习】：参数 retval 非空用法。 【pthrd_exit_join.c】

调用该函数的线程将挂起等待，直到 id 为 thread 的线程终止。thread 线程以不同的方法终止，通过 pthread_join 得到的终止状态是不同的，总结如下：

1. 如果 thread 线程通过 return 返回，retval 所指向的单元里存放的是 thread 线程函数的返回值。
2. 如果 thread 线程被别的线程调用 pthread_cancel 异常终止掉，retval 所指向的单元里存放的是常数 PTHREAD_CANCELED。
3. 如果 thread 线程是自己调用 pthread_exit 终止的，retval 所指向的单元存放的是传给 pthread_exit 的参数。
4. 如果对 thread 线程的终止状态不感兴趣，可以传 NULL 给 retval 参数。

【练习】：使用 pthread_join 函数将循环创建的多个子线程回收。 【pthrd_loop_join.c】

pthread_detach 函数

实现线程分离

int pthread_detach(pthread_t thread); 成功：0；失败：错误号

线程分离状态：指定该状态，线程主动与主控线程断开关系。线程结束后，其退出状态不由其他线程获取，而直接自己自动释放。网络、多线程服务器常用。

进程若有该机制，将不会产生僵尸进程。僵尸进程的产生主要由于进程死后，大部分资源被释放，一点残留资源仍存于系统中，导致内核认为该进程仍存在。

也可使用 pthread_create 函数参 2(线程属性)来设置线程分离。

【练习】：使用 pthread_detach 函数实现线程分离 【pthrd_detach.c】

一般情况下，线程终止后，其终止状态一直保留到其它线程调用 pthread_join 获取它的状态为止。但是线程也可以被置为 detach 状态，**这样的线程一旦终止就立刻回收它占用的所有资源，而不保留终止状态。**

不能对一个已经处于 detach 状态的线程调用 pthread_join，这样的调用将返回 EINVAL 错误。也就是说，如果已经对一个线程调用了 pthread_detach 就不能再调用 pthread_join 了。

pthread_cancel 函数

杀死(取消)线程 其作用，对应进程中 kill() 函数。

int pthread_cancel(pthread_t thread); 成功：0；失败：错误号

【注意】：线程的取消并不是实时的，而有一定的延时。需要等待线程到达某个取消点(检查点)。

类似于玩游戏存档，必须到达指定的场所(存档点，如：客栈、仓库、城里等)才能存储进度。杀死线程也不是

立刻就能完成，必须要到达取消点。

取消点：是线程检查是否被取消，并按请求进行动作的一个位置。通常是一些系统调用 creat，open，pause，close，read，write..... 执行命令 man 7 pthreads 可以查看具备这些取消点的系统调用列表。也可参阅 APUE.12.7 取消选项小节。

可粗略认为一个系统调用(进入内核)即为一个取消点。如线程中没有取消点，可以通过调用 pthread_testcancel 函数自行设置一个取消点。

被取消的线程， 退出值定义在 Linux 的 pthread 库中。常数 PTHREAD_CANCELED 的值是-1。可在头文件 pthread.h 中找到它的定义：**#define PTHREAD_CANCELED ((void *) -1)**。因此当我们对一个已经被取消的线程使用 pthread_join 回收时，得到的返回值为-1。

【练习】：终止线程的三种方法。注意“取消点”的概念。 【pthrd_endof3.c】

终止线程方式

总结：终止某个线程而不终止整个进程，有三种方法：

1. 从线程主函数 return。这种方法对主控线程不适用，从 main 函数 return 相当于调用 exit。
2. 一个线程可以调用 pthread_cancel 终止同一进程中的另一个线程。
3. 线程可以调用 pthread_exit 终止自己。

控制原语对比

进程	线程	
fork	pthread_create	
exit	pthread_exit	
wait	pthread_join	
kill	pthread_cancel	
getpid	pthread_self	命名空间

线程属性

本节作为指引性介绍，linux 下线程的属性是可以根据实际项目需要，进行设置，之前我们讨论的线程都是采用线程的默认属性，默认属性已经可以解决绝大多数开发时遇到的问题。如我们对程序的性能提出更高的要求那么需要设置线程属性，比如可以通过设置线程栈的大小来降低内存的使用，增加最大线程个数。

```
typedef struct
{
    int                 etachstate;     //线程的分离状态
    int                 schedpolicy;    //线程调度策略
    struct sched_param  schedparam;     //线程的调度参数
    int                 inheritsched;   //线程的继承性
    int                 scope;          //线程的作用域
```

```
    size_t          guardsize;      //线程栈末尾的警戒缓冲区大小
    int             stackaddr_set; //线程的栈设置
    void*           stackaddr;      //线程栈的位置
    size_t          stacksize;      //线程栈的大小
} pthread_attr_t;
```

主要结构体成员：

1. 线程分离状态
2. 线程栈大小（默认平均分配）
3. 线程栈警戒缓冲区大小（位于栈末尾）　　参 APUE.12.3 线程属性

属性值不能直接设置，须使用相关函数进行操作，初始化的函数为 pthread_attr_init，这个函数必须在 pthread_create 函数之前调用。之后须用 pthread_attr_destroy 函数来释放资源。

线程属性主要包括如下属性：作用域（scope）、栈尺寸（stack size）、栈地址（stack address）、优先级（priority）、分离的状态（detached state）、调度策略和参数（scheduling policy and parameters）。默认的属性为非绑定、非分离、缺省的堆栈、与父进程同样级别的优先级。

线程属性初始化

注意：应先初始化线程属性，再 pthread_create 创建线程

初始化线程属性

int pthread_attr_init(pthread_attr_t *attr); 成功：0；失败：错误号

销毁线程属性所占用的资源

int pthread_attr_destroy(pthread_attr_t *attr); 成功：0；失败：错误号

线程的分离状态

线程的分离状态决定一个线程以什么样的方式来终止自己。

非分离状态：线程的默认属性是非分离状态，这种情况下，原有的线程等待创建的线程结束。只有当 pthread_join() 函数返回时，创建的线程才算终止，才能释放自己占用的系统资源。

分离状态：分离线程没有被其他的线程所等待，自己运行结束了，线程也就终止了，马上释放系统资源。应该根据自己的需要，选择适当的分离状态。

线程分离状态的函数：

设置线程属性，分离 or 非分离

int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);

获取程属性，分离 or 非分离

int pthread_attr_getdetachstate(pthread_attr_t *attr, int *detachstate);

参数：　attr：已初始化的线程属性

detachstate： PTHREAD_CREATE_DETACHED（分离线程）

PTHREAD _CREATE_JOINABLE（非分离线程）

这里要注意的一点是，如果设置一个线程为分离线程，而这个线程运行又非常快，它很可能在 pthread_create 函数返回之前就终止了，它终止以后就可能将线程号和系统资源移交给其他的线程使用，这样调用 pthread_create 的线程就得到了错误的线程号。要避免这种情况可以采取一定的同步措施，最简单的方法之一是可以在被创建的线程里调用 pthread_cond_timedwait 函数，让这个线程等待一会儿，留出足够的时间让函数 pthread_create 返回。设置一段等待时间，是在多线程编程里常用的方法。但是注意不要使用诸如 wait()之类的函数，它们是使整个进程睡眠，并不能解决线程同步的问题。

线程属性控制示例

```
#include <pthread.h>

#define SIZE 0x100000
void *th_fun(void *arg)
{
	while (1)
		sleep(1);
}
int main(void)
{
	pthread_t tid;
	int err, detachstate, i = 1;
	pthread_attr_t attr;
	size_t stacksize;
	void *stackaddr;

	pthread_attr_init(&attr);
	pthread_attr_getstack(&attr, &stackaddr, &stacksize);
	pthread_attr_getdetachstate(&attr, &detachstate);

	if (detachstate == PTHREAD_CREATE_DETACHED)
		printf("thread detached\n");
	else if (detachstate == PTHREAD_CREATE_JOINABLE)
		printf("thread join\n");
	else
		printf("thread unknown\n");

	pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

	while (1) {
		stackaddr = malloc(SIZE);
		if (stackaddr == NULL) {
			perror("malloc");
			exit(1);
		}
```

```
        stacksize = SIZE;
        pthread_attr_setstack(&attr, stackaddr, stacksize);
        err = pthread_create(&tid, &attr, th_fun, NULL);
        if (err != 0) {
            printf("%s\n", strerror(err));
            exit(1);
        }
        printf("%d\n", i++);
    }
    pthread_attr_destroy(&attr);
    return 0;
}
```

【pthrd_attr_change.c】

线程使用注意事项

1. 主线程退出其他线程不退出，主线程应调用 pthread_exit
2. 避免僵尸线程

   pthread_join

   pthread_detach

   pthread_create 指定分离属性

   被 join 线程可能在 join 函数返回前就释放完自己的所有内存资源，所以不应当返回被回收线程栈中的值;
3. malloc 和 mmap 申请的内存可以被其他线程释放
4. 应避免在多线程模型中调用 fork 除非，马上 exec，子进程中只有调用 fork 的线程存在，其他线程在子进程中均 pthread_exit
5. 信号的复杂语义很难和多线程共存，应避免在多线程引入信号机制

同步概念

所谓同步，即同时起步，协调一致。不同的对象，对“同步”的理解方式略有不同。如，设备同步，是指在两个设备之间规定一个共同的时间参考；数据库同步，是指让两个或多个数据库内容保持一致，或者按需要部分保持一致；文件同步，是指让两个或多个文件夹里的文件保持一致。等等

而，编程中、通信中所说的同步与生活中大家印象中的同步概念略有差异。“同”字应是指协同、协助、互相配合。主旨在协同步调，按预定的先后次序运行。

线程同步

同步即协同步调，按预定的先后次序运行。

线程同步，指一个线程发出某一功能调用时，在没有得到结果之前，该调用不返回。同时其它线程为保证数据一致性，不能调用该功能。

举例 1： 银行存款 5000。柜台，折：取 3000；提款机，卡：取 3000。剩余：2000

举例 2： 内存中 100 字节，线程 T1 欲填入全 1， 线程 T2 欲填入全 0。但如果 T1 执行了 50 个字节失去 cpu，T2 执行，会将 T1 写过的内容覆盖。当 T1 再次获得 cpu 继续 从失去 cpu 的位置向后写入 1，当执行结束，内存中的 100 字节，既不是全 1，也不是全 0。

产生的现象叫做“与时间有关的错误”(time related)。为了避免这种数据混乱，线程需要同步。

“同步”的目的，是为了避免数据混乱，解决与时间有关的错误。实际上，不仅线程间需要同步，进程间、信号间等等都需要同步机制。

因此，**所有“多个控制流，共同操作一个共享资源”的情况，都需要同步。**

数据混乱原因：

1. 资源共享（独享资源则不会）
2. 调度随机（意味着数据访问会出现竞争）
3. 线程间缺乏必要的同步机制。

以上 3 点中，前两点不能改变，欲提高效率，传递数据，资源必须共享。只要共享资源，就一定会出现竞争。只要存在竞争关系，数据就很容易出现混乱。

所以只能从第三点着手解决。使多个线程在访问共享资源的时候，出现互斥。

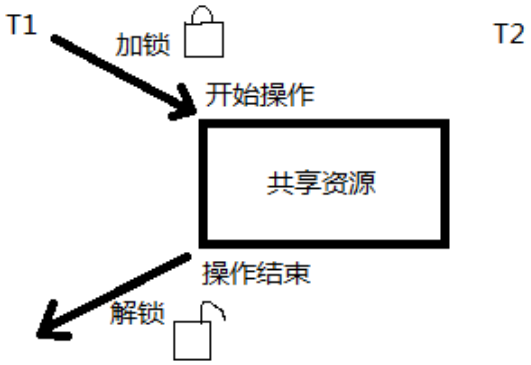
互斥量 mutex

Linux 中提供一把互斥锁 mutex（也称之为互斥量）。

每个线程在对资源操作前都尝试先加锁，成功加锁才能操作，操作结束解锁。

资源还是共享的，线程间也还是竞争的，

但通过“锁”就将资源的访问变成互斥操作，而后与时间有关的错误也不会再产生了。



但，应注意：同一时刻，只能有一个线程持有该锁。

当 A 线程对某个全局变量加锁访问，B 在访问前尝试加锁，拿不到锁，B 阻塞。C 线程不去加锁，而直接访问该全局变量，依然能够访问，但会出现数据混乱。

所以，互斥锁实质上是操作系统提供的一把“建议锁”（又称“协同锁”），建议程序中有多线程访问共享资源的时候使用该机制。但，并没有强制限定。

因此，即使有了 mutex，如果有线程不按规则来访问数据，依然会造成数据混乱。

主要应用函数：

pthread_mutex_init 函数

pthread_mutex_destroy 函数

pthread_mutex_lock 函数

pthread_mutex_trylock 函数

pthread_mutex_unlock 函数

以上 5 个函数的返回值都是：成功返回 0， 失败返回错误号。

pthread_mutex_t 类型，其本质是一个结构体。为简化理解，应用时可忽略其实现细节，**简单当成整数看待。**

pthread_mutex_t mutex; 变量 mutex 只有两种取值 1、0。

pthread_mutex_init 函数

初始化一个互斥锁(互斥量) ---> 初值可看作 1

int pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restrict attr);

参 1：传出参数，调用时应传 &mutex

restrict 关键字：只用于限制指针，告诉编译器，所有修改该指针指向内存中内容的操作，只能通过本指针完成。不能通过除本指针以外的其他变量或指针修改

参 2：互斥量属性。是一个传入参数，通常传 NULL，选用默认属性(线程间共享)。 参 APUE.12.4 同步属性

1. 静态初始化：如果互斥锁 mutex 是静态分配的（定义在全局，或加了 static 关键字修饰），可以直接使用宏进行初始化。e.g. pthead_mutex_t muetx = PTHREAD_MUTEX_INITIALIZER;

2. 动态初始化：局部变量应采用动态初始化。e.g. pthread_mutex_init(&mutex, NULL)

pthread_mutex_destroy 函数

销毁一个互斥锁

int pthread_mutex_destroy(pthread_mutex_t *mutex);

pthread_mutex_lock 函数

加锁。可理解为将 **mutex--**（或 -1），操作后 mutex 的值为 0。

int pthread_mutex_lock(pthread_mutex_t *mutex);

pthread_mutex_unlock 函数

解锁。可理解为将 **mutex ++**（或 +1），操作后 mutex 的值为 1。

int pthread_mutex_unlock(pthread_mutex_t *mutex);

pthread_mutex_trylock 函数

尝试加锁

int pthread_mutex_trylock(pthread_mutex_t *mutex);

加锁与解锁

lock 与 unlock：

lock 尝试加锁，如果加锁不成功，线程阻塞，阻塞到持有该互斥量的其他线程解锁为止。

unlock 主动解锁函数，**同时将阻塞在该锁上的所有线程全部唤醒**，至于哪个线程先被唤醒，取决于优先级、调度。默认：先阻塞、先唤醒。

例如：T1 T2 T3 T4 使用一把 mutex 锁。T1 加锁成功，其他线程均阻塞，直至 T1 解锁。T1 解锁后，T2 T3 T4 均被唤醒，并自动再次尝试加锁。

可假想 mutex 锁 init 成功初值为 1。lock 功能是将 mutex--。而 unlock 则将 mutex++。

lock 与 trylock：

lock 加锁失败会阻塞，等待锁释放。

trylock 加锁失败直接返回错误号（如：EBUSY），不阻塞。

加锁步骤测试：

看如下程序：该程序是非常典型的，由于共享、竞争而没有加任何同步机制，导致产生于时间有关的错误，造成数据混乱：

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

void *tfn(void *arg)
{
	srand(time(NULL));
	while (1) {

		printf("hello ");
		sleep(rand() % 3);  /*模拟长时间操作共享资源，导致 cpu 易主，产生与时间有关的错误*/
		printf("world\n");
		sleep(rand() % 3);
	}
	return NULL;
}
int main(void)
{
	pthread_t tid;
	srand(time(NULL));
	pthread_create(&tid, NULL, tfn, NULL);
	while (1) {
		printf("HELLO ");
		sleep(rand() % 3);
		printf("WORLD\n");
		sleep(rand() % 3);
	}
	pthread_join(tid, NULL);
	return 0;
}
```

【mutex.c】

【练习】：修改该程序，使用 mutex 互斥锁进行同步。

1. 定义全局互斥量，初始化 init(&m, NULL)互斥量，添加对应的 destry
2. 两个线程 while 中，两次 printf 前后，分别加 lock 和 unlock
3. 将 unlock 挪至第二个 sleep 后，发现交替现象很难出现。

   线程在操作完共享资源后本应该立即解锁，但修改后，线程抱着锁睡眠。睡醒解锁后又立即加锁，这两个库函数本身不会阻塞。

   所以在这两行代码之间失去 cpu 的概率很小。因此，另外一个线程很难得到加锁的机会。
4. main 中加 flag = 5 将 flg 在 while 中-- 这时，主线程输出 5 次后试图销毁锁，但子线程未将锁释放，无法完成。

5. main 中加 pthread_cancel()将子线程取消。 【pthrd_mutex.c】

结论：

在访问共享资源前加锁，访问结束后立即解锁。锁的“粒度”应越小越好。

死锁

1. 线程试图对同一个互斥量 A 加锁两次。
2. 线程 1 拥有 A 锁，请求获得 B 锁；线程 2 拥有 B 锁，请求获得 A 锁

【作业】：编写程序，实现上述两种死锁现象。

读写锁

与互斥量类似，但读写锁允许更高的并行性。其特性为：**写独占，读共享。**

读写锁状态：

特别强调：读写锁只有一把，但其具备两种状态：

1. 读模式下加锁状态 (读锁)
2. 写模式下加锁状态 (写锁)

读写锁特性：

1. 读写锁是“写模式加锁”时， 解锁前，所有对该锁加锁的线程都会被阻塞。
2. 读写锁是“读模式加锁”时， 如果线程以读模式对其加锁会成功；如果线程以写模式加锁会阻塞。
3. 读写锁是“读模式加锁”时， 既有试图以写模式加锁的线程，也有试图以读模式加锁的线程。那么读写锁会阻塞随后的读模式锁请求。优先满足写模式锁。**读锁、写锁并行阻塞，写锁优先级高**

读写锁也叫共享-独占锁。当读写锁以读模式锁住时，它是以共享模式锁住的；当它以写模式锁住时，它是以独占模式锁住的。**写独占、读共享。**

读写锁非常适合于对数据结构读的次数远大于写的情况。

主要应用函数：

pthread_rwlock_init 函数

pthread_rwlock_destroy 函数

pthread_rwlock_rdlock 函数

pthread_rwlock_wrlock 函数

pthread_rwlock_tryrdlock 函数

pthread_rwlock_trywrlock 函数

pthread_rwlock_unlock 函数

以上 7 个函数的返回值都是：成功返回 0， 失败直接返回错误号。

pthread_rwlock_t 类型 用于定义一个读写锁变量。

pthread_rwlock_t rwlock;

pthread_rwlock_init 函数

初始化一把读写锁

int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock, const pthread_rwlockattr_t *restrict attr);

参 2：attr 表读写锁属性，通常使用默认属性，传 NULL 即可。

pthread_rwlock_destroy 函数

销毁一把读写锁

int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);

pthread_rwlock_rdlock 函数

以读方式请求读写锁。（常简称为：请求读锁）

int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);

pthread_rwlock_wrlock 函数

以写方式请求读写锁。（常简称为：请求写锁）

int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);

pthread_rwlock_unlock 函数

解锁

int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);

pthread_rwlock_tryrdlock 函数

非阻塞以读方式请求读写锁（非阻塞请求读锁）

int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);

pthread_rwlock_trywrlock 函数

非阻塞以写方式请求读写锁（非阻塞请求写锁）

int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);

读写锁示例

看如下示例，同时有多个线程对同一全局数据读、写操作。

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

int counter;
pthread_rwlock_t rwlock;

/* 3 个线程不定时写同一全局资源，5 个线程不定时读同一全局资源 */
void *th_write(void *arg)
{
	int t, i = (int)arg;
	while (1) {
		pthread_rwlock_wrlock(&rwlock);
		t = counter;
		usleep(1000);
		printf("=======write %d: %lu: counter=%d ++counter=%d\n", i, pthread_self(), t, ++counter);
		pthread_rwlock_unlock(&rwlock);
		usleep(10000);
	}
	return NULL;
}
void *th_read(void *arg)
{
	int i = (int)arg;

	while (1) {
		pthread_rwlock_rdlock(&rwlock);
		printf("----------------------------read %d: %lu: %d\n", i, pthread_self(), counter);
		pthread_rwlock_unlock(&rwlock);
		usleep(2000);
	}
	return NULL;
}
int main(void)
{
	int i;
	pthread_t tid[8];
```

```
    pthread_rwlock_init(&rwlock, NULL);

    for (i = 0; i < 3; i++)
        pthread_create(&tid[i], NULL, th_write, (void *)i);
    for (i = 0; i < 5; i++)
        pthread_create(&tid[i+3], NULL, th_read, (void *)i);
    for (i = 0; i < 8; i++)
        pthread_join(tid[i], NULL);

    pthread_rwlock_destroy(&rwlock);
    return 0;
}
```

【rwlock.c】

条件变量：

条件变量本身不是锁！但它也可以造成线程阻塞。通常与互斥锁配合使用。给多线程提供一个会合的场所。

主要应用函数：

pthread_cond_init 函数

pthread_cond_destroy 函数

pthread_cond_wait 函数

pthread_cond_timedwait 函数

pthread_cond_signal 函数

pthread_cond_broadcast 函数

以上 6 个函数的返回值都是：成功返回 0， 失败直接返回错误号。

pthread_cond_t 类型　用于定义条件变量

pthread_cond_t cond;

pthread_cond_init 函数

初始化一个条件变量

int pthread_cond_init(pthread_cond_t *restrict cond, const pthread_condattr_t *restrict attr);

参 2：attr 表条件变量属性，通常为默认值，传 NULL 即可

也可以使用静态初始化的方法，初始化条件变量：

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

pthread_cond_destroy 函数

销毁一个条件变量

int pthread_cond_destroy(pthread_cond_t *cond);

pthread_cond_wait 函数

阻塞等待一个条件变量

int pthread_cond_wait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex);

函数作用：

1. 阻塞等待条件变量 cond（参 1）满足
2. 释放已掌握的互斥锁（解锁互斥量）相当于 pthread_mutex_unlock(&mutex);

   1.2.两步为一个原子操作。
3. 当被唤醒，pthread_cond_wait 函数返回时，解除阻塞并重新申请获取互斥锁 pthread_mutex_lock(&mutex);

pthread_cond_timedwait 函数

限时等待一个条件变量

int pthread_cond_timedwait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex, const struct timespec *restrict abstime);

参 3： 参看 man sem_timedwait 函数，查看 struct timespec 结构体。

```
struct timespec {
    time_t tv_sec;      /* seconds */  秒
    long   tv_nsec;     /* nanosecondes*/  纳秒
}
```

形参 abstime：绝对时间。

如：time(NULL)返回的就是绝对时间。而 alarm(1)是相对时间，相对当前时间定时 1 秒钟。

struct timespec t = {1, 0};

pthread_cond_timedwait (&cond, &mutex, &t); 只能定时到 1970 年 1 月 1 日 00:00:01 秒(早已经过去)

正确用法：

time_t cur = time(NULL); 获取当前时间。

struct timespec t; 定义 timespec 结构体变量 t

t.tv_sec = cur+1; 定时 1 秒

pthread_cond_timedwait (&cond, &mutex, &t); 传参 参 APUE.11.6 线程同步条件变量小节

在讲解 setitimer 函数时我们还提到另外一种时间类型：

```
struct timeval {
    time_t      tv_sec;   /* seconds */  秒
    suseconds_t tv_usec;  /* microseconds */  微秒
};
```

pthread_cond_signal 函数

唤醒至少一个阻塞在条件变量上的线程

```
int pthread_cond_signal(pthread_cond_t *cond);
```

pthread_cond_broadcast 函数

唤醒全部阻塞在条件变量上的线程

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

生产者消费者条件变量模型

线程同步典型的案例即为生产者消费者模型，而借助条件变量来实现这一模型，是比较常见的一种方法。假定有两个线程，一个模拟生产者行为，一个模拟消费者行为。两个线程同时操作一个共享资源（一般称之为汇聚），生产向其中添加产品，消费者从中消费掉产品。

看如下示例，使用条件变量模拟生产者、消费者问题：

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

struct msg {
	struct msg *next;
	int num;
};
struct msg *head;

pthread_cond_t has_product = PTHREAD_COND_INITIALIZER;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void *consumer(void *p)
{
	struct msg *mp;
	for (;;) {
		pthread_mutex_lock(&lock);
		while (head == NULL) {		//头指针为空,说明没有节点	可以为 if 吗
			pthread_cond_wait(&has_product, &lock);
		}
		mp = head;
		head = mp->next;		//模拟消费掉一个产品
		pthread_mutex_unlock(&lock);

		printf("-Consume ---%d\n", mp->num);
		free(mp);
```

```
            sleep(rand() % 5);
        }
    }
    void *producer(void *p)
    {
        struct msg *mp;
        while (1) {
            mp = malloc(sizeof(struct msg));
            mp->num = rand() % 1000 + 1;            //模拟生产一个产品
            printf("-Produce ---%d\n", mp->num);

            pthread_mutex_lock(&lock);
            mp->next = head;
            head = mp;
            pthread_mutex_unlock(&lock);

            pthread_cond_signal(&has_product);   //将等待在该条件变量上的一个线程唤醒
            sleep(rand() % 5);
        }
    }
    int main(int argc, char *argv[])
    {
        pthread_t pid, cid;
        srand(time(NULL));

        pthread_create(&pid, NULL, producer, NULL);
        pthread_create(&cid, NULL, consumer, NULL);

        pthread_join(pid, NULL);
        pthread_join(cid, NULL);
        return 0;
    }
```

【conditionVar_product_consumer.c】

条件变量的优点：

相较于 mutex 而言，条件变量可以减少竞争。

如直接使用 mutex，除了生产者、消费者之间要竞争互斥量以外，消费者之间也需要竞争互斥量，但如果汇聚（链表）中没有数据，消费者之间竞争互斥锁是无意义的。有了条件变量机制以后，只有生产者完成生产，才会引起消费者之间的竞争。提高了程序效率。

信号量

进化版的互斥锁（1 --> N）

由于互斥锁的粒度比较大，如果我们希望在多个线程间对某一对象的部分数据进行共享，使用互斥锁是没有办法实现的，只能将整个数据对象锁住。这样虽然达到了多线程操作共享数据时保证数据正确性的目的，却无形中导

致线程的并发性下降。线程从并行执行，变成了串行执行。与直接使用单进程无异。

信号量，是相对折中的一种处理方式，既能保证同步，数据不混乱，又能提高线程并发。

主要应用函数：

sem_init 函数

sem_destroy 函数

sem_wait 函数

sem_trywait 函数

sem_timedwait 函数

sem_post 函数

以上 6 个函数的返回值都是：成功返回 0， 失败返回-1，同时设置 errno。(注意，它们没有 pthread 前缀)

sem_t 类型，本质仍是结构体。但应用期间可简单看作为整数，忽略实现细节（类似于使用文件描述符）。

sem_t sem; 规定信号量 sem 不能 < 0。头文件 <semaphore.h>

信号量基本操作：

sem_wait: 1. 信号量大于 0，则信号量-- （类比 pthread_mutex_lock）
| 2. 信号量等于 0，造成线程阻塞
对应
|
sem_post： 将信号量++，同时唤醒阻塞在信号量上的线程 （类比 pthread_mutex_unlock）

但，由于 sem_t 的实现对用户隐藏，所以所谓的++、--操作只能通过函数来实现，而不能直接++、--符号。

信号量的初值，决定了占用信号量的线程的个数。

sem_init 函数

初始化一个信号量

int sem_init(sem_t *sem, int pshared, unsigned int value);

参 1：sem 信号量

参 2：pshared 取 0 用于线程间；取非 0（一般为 1）用于进程间

参 3：value 指定信号量初值

sem_destroy 函数

销毁一个信号量

int sem_destroy(sem_t *sem);

sem_wait 函数

给信号量加锁 --

int sem_wait(sem_t *sem);

sem_post 函数

给信号量解锁 ++

int sem_post(sem_t *sem);

sem_trywait 函数

尝试对信号量加锁 -- (与 sem_wait 的区别类比 lock 和 trylock)

int sem_trywait(sem_t *sem);

sem_timedwait 函数

限时尝试对信号量加锁 --

int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);

参 2：abs_timeout 采用的是绝对时间。

定时 1 秒：

```
time_t cur = time(NULL); 获取当前时间。
struct timespec t;  定义 timespec 结构体变量 t
t.tv_sec = cur+1; 定时 1 秒
t.tv_nsec = t.tv_sec +100;
sem_timedwait(&sem, &t); 传参
```

生产者消费者信号量模型

【练习】：使用信号量完成线程间同步，模拟生产者，消费者问题。

【sem_product_consumer.c】

分析：

规定： 如果□中有数据，生产者不能生产，只能阻塞。

如果□中没有数据，消费者不能消费，只能等待数据。

定义两个信号量：S 满 = 0， S 空 = 1 （S 满代表满格的信号量，S 空表示空格的信号量，程序起始，格子一定为空）

所以有：

```
T 生产者主函数 {                    T 消费者主函数 {
    sem_wait(S 空);                     sem_wait(S 满);
    生产....                            消费....
    sem_post(S 满);                     sem_post(S 空);
}                                   }
```

假设： 线程到达的顺序是:T 生、T 生、T 消。

那么： T 生 1 到达，将 S 空-1，生产，将 S 满+1
T 生 2 到达，S 空已经为 0， 阻塞
T 消 到达，将 S 满-1，消费，将 S 空+1

三个线程到达的顺序是：T 生 1、T 生 2、T 消。而执行的顺序是 T 生 1、T 消、T 生 2

这里，S 空 表示空格子的总数，代表可占用信号量的线程总数-->1。其实这样的话，信号量就等同于互斥锁。

但，如果 S 空=2、3、4……就不一样了，该信号量同时可以由多个线程占用，不再是互斥的形式。因此我们说信号量是互斥锁的加强版。

【推演练习】： 理解上述模型，推演，如果是两个消费者，一个生产者，是怎么样的情况。

【作业】：结合生产者消费者信号量模型，揣摩 sem_timedwait 函数作用。编程实现，一个线程读用户输入， 另一个线程打印“hello world”。如果用户无输入，则每隔 5 秒向屏幕打印一个“hello world”；如果用户有输入，立刻打印“hello world”到屏幕。