



目 录

常见出错处理.....	1
1、abort	1
2、assert宏	1
3、exit.....	1
4、atexit.....	1
5、errno变量:	1
6、strerror.....	2
7、perror.....	2
系统日志函数syslog	3
常用的I/O与文件、目录操作函数	4
1、open	4
2、close	4
3、read.....	4
4、write.....	4
5、ftruncate.....	4
6、lseek.....	5
7、fsync.....	5
8、fstat.....	5
9、fchmod	6
10、flock和fcntl	6
11、dup和dup2	6
12、select.....	7
13、ioctl	7
文件目录，及其I/O	9
1、打开关闭文件函数	9
2、读写文件.....	9
3、文件状态	9
4、printf族格式化输出	10
5、scanf族格式化输入.....	10
6、字符的I/O	11
7、字符串（不换行）的I/O.....	11
8、文件的定位	11
9、缓冲区控制.....	12
10、删除和重命名文件.....	12
11、临时文件.....	12
12、目录操作.....	12
13、获得目录列表:	13
进程与信号的相关函数.....	14
1、进程信息检测	14
2、进程的创建	15



2.1 system	15
2.2 fork	15
2.3 exec函数族	16
2.4 popen	16
3. 进程控制	17
3.1 wait、waitpid	17
3.2 中止进程的函数	17
3.3 信号	18
3.4 信号的创建和处理	18
4. 进程的调度	19
POSIX线程基本概念	21
1、__clone函数调用	21
2、POSIX线程库的pthread API	22
3、线程属性	23
4、pthread cleanup宏	23
5、互斥mutex	23
6、条件变量	24
内存管理相关函数	26
1、经典的C动态内存管理相关函数	26
2、Linux的内存映像管理函数	28
进程间通信IPC	31
1、管道	31
2、FIFO	31
3、SysV IPC	32
4、共享内存	33
5、消息队列	34
7、信号灯	35
守护进程程序设计的基本要点	38



常见出错处理

1、abort

定义:

```
#include <stdlib.h>
void abort(void);
```

作用: 强行终止程序 (异常终止)。如果当前 shell 不限制 ulimit, 将会 core dump。

2、assert宏

原型:

```
#include <assert.h>
void assert(int expression);
```

作用: 计算 expression 的值, 若其返回 0 (假), 则向 stderr 打印出错信息, 并调用 abort 终止程序。

使用注意事项: assert 一般在开发阶段用于调试。为防止定义 NDEBUG 后 assert 被禁用, 最好不要直接向 assert 进行输入, 而是如下所示:

```
p = malloc (sizeof (char) *100);
assert(p);
```

3、exit

原型:

```
#include <stdlib.h>
void exit(int status);
```

作用: 返回一个状态值给操作系统, status 在 stdlib.h 中定义了 EXIT_SUCCESS 和 EXIT_FAILURE。

4、atexit

原型:

```
#include <stdlib.h>
int atexit(void (*function)(void));
```

作用: 注册一个函数, 这个函数可以定义一些操作, 用来在程序正常退出时执行之。atexit 注册成功则返回 0, 否则返回 1; 可以用“,”隔开注册多个函数, 执行顺序为最左边的最后执行。

5、errno变量:

定义:



```
#include <errno.h>;  
int errno;
```

作用：全局变量，Linux 系统调用与大部分库函数设置该值，errno.h 定义了其值对应的错误。例如 ENOENT 代表 No such file or directory 等。函数 perror 可以打印相应的出错信息。

注意事项：很多函数返回并设置 errno 后并不会把之清 0,如果调用一个可能在出错时设置 errno 的库函数的时候，最好先手动把 errno 清零。

6、strerror

原型：

```
#include <string.h>  
char *strerror(int errnum);
```

作用：把 errno 转换成标准的出错信息。例如：

```
ps = strerror(ENOENT);
```

则指针 ps 指向的字符串为 "No such file or directory "等。

7、perror

原型：

```
#include <stdlib.h>  
#include <errno.h>  
void perror(const char *s);
```

作用：打印 s 所指的字符串和标准出错信息，相当于

```
printf("%s: %s\n", *s, strerror(errno));
```



系统日志函数syslog

原型:

```
#include <syslog.h>
void syslog(int priority, char *format, ...);
```

作用: 向/var/log 下面的某个日志文件添加一条新的日志记录。向哪个文件添加取决于 priority。

priority 为严重性与功能值的逻辑或值, 严重性从高到低包括 LOG_EMERG、LOG_ALERT、LOG_CRIT、LOG_ERR、LOG_WARNING、LOG_NOTICE、LOG_INFO、LOG_DEBUG 等 8 项。功能值提供了这个 log 的作用, 包括了 LOG_CRON, LOG_USER, LOG_KERN, LOG_DAEMON, LOG_KERN 等等。

*format 为记录的字符串, 遵循类似 printf 一般的格式, 且其中的%m 表示由 strerror 返回的 errno 值代表的信息。

一般使用 openlog 函数来定制日志, 其原型是:

```
#include <syslog.h>
void openlog(const char *ident, int option, int facility);
```

其中 ident 指定了要加在日志信息前的字符串, option 代表了 LOG_PID, LOG_CONS, LOG_NDELAY, LOG_PERROR 中的零个到多个的逻辑或值。而 facility 则为 log 的严重性。

例如:

```
openlog("logit", LOG_PID, LOG_USER);
syslog( LOG_INFO, "Hey man\n");
closelog();
```

则往/var/log/messages 产生一行信息形如:

```
Nov 29 23:47:46 ubuntu logit[18229]: Hey man
```

同时, shell 提供了 logger 这个工具直接在 shell 脚本中写 log:

```
logger [-is] [-f file] [-p pri] [-t tag] [-u socket] [ message ... ]
```



常用的I/O与文件、目录操作函数

1、open

原型:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags);
```

作用: 用 flags 指定的操作打开 pathname 指定的文件。成功返回一个文件描述符, 失败返回-1 并设置 errno。

常用的 flags 包括 O_RDONLY, O_WRONLY, O_RDWR, O_CREAT, O_TRUNC, O_APPEND 等等。它们直接可以在合理的前提下进行逻辑或, 例如 O_CREAT | O_TRUNC | O_WRONLY。

2、close

原型:

```
#include <unistd.h>
int close(int fd);
```

作用: 注销描述符 fd, 关闭文件。失败则返回-1 并设置 errno。

3、read

原型:

```
#include <unistd.h>
ssize_t read(int fd, const void *buf, size_t count);
```

作用: 从描述符 fd 引用的文件中读出 count 个字节到 buf 所指向的缓冲区。成功返回实际读出的字节数, 失败返回-1 并设置 errno。

4、write

原型:

```
#include <unistd.h>
ssize_t write(int fd, const void *buf, size_t count);
```

作用: 把 buf 指向的缓冲区中的 count 个字节写入到描述符 fd 引用的文件。成功返回实际写入的字节数, 失败返回-1 并设置 errno。

5、ftruncate

原型:

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



```
int ftruncate(int fd, off_t length);
```

作用：把描述符 fd 引用的文件缩短到 length 指定的长度。成功返回 0，失败返回-1 并设置 errno。

6、lseek

原型：

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence);
```

作用：定位操作文件的指针到相对 whence 偏移 offset 的位置。

whence 代表：

SEEK_SET ——开始于文件头

SEEK_CUR ——开始于当前指针位置，这时 offset 可以为负数

SEEK_END ——从文件结尾往回数。

函数调用成功返回指针的新位置，失败或出错返回-1 并设置 errno。

7、fsync

原型：

```
#include <unistd.h>
int fsync(int fd);
```

作用：把在 fd 上执行的写入操作同步到真正的磁盘或其它下层设备文件中。

成功返回 0，失败返回-1 并设置 errno。

8、fstat

原型：

```
#include <sys/stat.h>
#include <unistd.h>
int fstat(int fd, struct stat *buf);
```

作用：把描述符 fd 引用的文件的相关信息保存到 buf 指向的 stat 结构中。成功返回 0，失败返回-1 并设置 errno。

其中 stat 结构的成员 st_mode 即文件位模式可以用 S_ISLNK(mode)、S_ISREG(mode)、S_ISDIR(mode).....等宏返回的真假判断其文件类型(目录、普通文件、符号链接、字符设备、块设备、FIFO 管道、套接口等)。

另外，st_atime 成员可以用 ctime 这个函数转换为 datetime 格式的字符串。



ctime 的原型为:

```
#include <time.h>
char *ctime(const time_t *timer);
```

9、fchmod

原型:

```
#include <sys/types.h>
#include <sys/stat.h>
int fchmod(int fd, mode_t mode);
```

作用: 把 fd 引用的文件的模式改变为 mode 指定的模式, 类似 shell 里面的 chmod 命令。但要注意八进制的表示方法为以 0 起头。成功返回 0, 失败返回-1 并设置 errno。

10、flock和fcntl

原型:

```
#include <sys/file.h>
int flock(int fd, int operation);
```

```
#include <unistd.h>
#include <fcntl.h>
int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, long arg);
int fcntl(int fd, int cmd, struct flock *lock);
```

作用: 在打开文件前给文件上锁以防止其被其它 IO 进程访问而引起不可预计的后果。

flock 的 operation 包括 LOCK_SH, LOCK_EX, LOCK_UN。fcntl 的 cmd 包括 F_GETLK, F_SETLK, F_SETLKW.....等等。成功返回 0, 失败返回-1。

fcntl 比 flock 更通用。除了符合 POSIX 标准外, fcntl 还同时支持建议性锁(由程序来检查锁)和强制性锁(由操作系统内核检查锁), 另外可以用于读取锁(共享锁)和写入锁(排斥锁)。fcntl 除了可以上锁文件外, 还可以控制文件的进程组、复制文件描述符等。

11、dup和dup2

原型:

```
#include <unistd.h>
int dup(int oldfd);
```




```
int dup2(int oldfd, int newfd);
```

作用: dup 和 dup2 都用于复制文件描述符, 它们调用成功都返回新的描述符, 失败则返回-1 并设置 errno。其中 dup2 可以自定义新的文件描述符, 通常用来重新打开或者重定向文件描述符。例如:

```
dup2(fd, STDOUT_FILENO);
```

重定向文件描述符 fd 到标准输出。

12、select

原型:

```
#include <unistd.h>
int select(int n, fd_set *readfds, fd_set *writefds, fd_set
exceptfds, struct timeval *timeout);
```

作用: 同时读/写 fd_set 结构中的描述符集合, n 代表集合中的文件描述符最大值加 1, 不需要的参数设置为 NULL。timeout 定义 select 的阻塞等待时间, 如果设置为 0 则为非阻塞式的 I/O 调用, 如果设置为 NULL 则一直等到 I/O 操作发生或者出错。

select 成功返回受监视的 fds 集合的文件描述符总数, 如果 timeout 时集合中的文件描述符都没有该表状态则返回 0, 失败则返回-1 并设置 errno。

例如:

```
#include <unistd.h>
#include <sys/time.h>
#include <sys/types.h>
fd_set *writeable_fds;
select(maxfds, NULL, writefds, NULL, 10);
```

有四个宏函数可以对描述符集合 fd_set 进行操作:

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

FD_ZERO(fd_set *set);           /* 清空集合 set */
FD_SET(int fd, fd_set *set);    /* 把描述符 fd 添加到集合 set */
FD_CLR(int fd, fd_set *set);    /* 从集合 set 中删除描述符 fd */
FD_ISSET(int fd, fd_set *set);  /* 判断描述符 fd 是否在集合 set 中 */
```

13、ioctl

原型:

```
#include <sys/ioctl.h>
int ioctl(int d, int request, ...);
```



作用：设置或检索文件的多种有关参数并对文件进行一些其它操作。是否可以 `ioctl` 以及传递什么参数给 `ioctl` 因下层设备而异。



文件目录，及其I/O

1、打开关闭文件函数

```
#include <stdio.h>
FILE *fopen(const char *path, const char mode);
FILE *freopen(const char *path, const char mode, FILE *stream);
int fclose(FILE *stream);
```

作用：fopen 以指定的模式打开一个字符串 path 指定的文件，返回一个文件流指针。如果文件不存在，以 0666 模式创建此文件。打开失败返回 NULL 并设置 errno。

fopen 使用的打开模式包括：r, r+, w, w+, a, a+, rb, rb+, wb, wb+, ab, ab+。

freopen 以指定的模式打开一个字符串 path 指定的文件，同时使文件流指针 stream 指向它。它可以类似 dup2 那样重定向 stdout, stderr 等这些标准的文件流。失败返回 NULL 并设置 errno。

fclose 关闭一个文件流，此后对此文件流指针的访问会导致出现不可预料的结果，必须采取相关处理方式。

失败返回 EOF 并设置 errno。

另有一个非 ANSI C 的 POSIX 调用 fdopen，用于以指定的模式打开一个文件描述符，返回文件流指针。原型为：

```
#include <stdio.h>
FILE *fdopen(int fd, const char *mode);
```

2、读写文件

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fwrite(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

作用：fread 把已经打开的文件流 stream 读取到 ptr 所指定的缓冲区，每次调用依次读取 nmemb 个 size 大小的文件块。

fwrite 则读取 nmemb 个 size 大小的文件块从 ptr 写入到 stream。

3、文件状态

```
#include <stdio.h>
int feof(FILE *stream);
int ferror(FILE *stream);
```



```
void clearerr(FILE *stream);  
int fileno(FILE *stream);
```

作用：feof 用于检测 EOF 结束标志，同时返回非零值，但此时一般已经超出文件末尾。

feror 用于检测文件流的出错标志，同时返回非零值。

clearerr 用于清除文件流的 EOF 标志和出错标志。

fileno 为非标准 ANSI C 调用，返回和文件流 stream 相关的文件描述符，以执行基于文件描述符的 I/O 操作。

4、printf族格式化输出

```
#include <stdio.h>  
int printf(const char *format, ...);  
int fprintf(FILE *stream, const char *format, ...);  
int sprintf(char *str, const char *format, ...);  
int snprintf(char *str, size_t size, const char *format, ...);
```

作用：格式化输出字符串到标准输出(printf)、文件流(fprintf)、缓冲区(sprintf, snprintf)。

ANSI C 的 sprintf 由于不检验缓冲区长度，容易遭受缓冲区溢出的攻击，应以 POSIX 的 snprintf 取代。

```
#include <stdarg.h>  
int vprintf(const char *format, va_list ap);  
int vfprintf(FILE *stream, const char format, va_list ap);  
int vsprintf(char *ptr, const char *format, va_list ap);  
int vsnprintf(char *str, size_t size, const char *format,  
va_list ap);
```

作用：这是一组带可变参数列表的 printf 函数族，但是至少要有一个固定参数，后面必须包含“...”以表明可变参数开始。

5、scanf族格式化输入

```
#include <stdio.h>  
int scanf(const char *format, ...);  
int fscanf(FILE *stream, const char *format, ...);  
int sscanf(const char *str, const char *format, ...);
```

```
#include <stdarg.h>  
int vscanf(const char *format, va_list ap);  
int vfscanf(FILE *stream, const char *format, va_list ap);
```



```
int vsscanf(const char *ptr, const char format, va_list ap);
```

作用，和 printf 相反，scanf 族从标准输入/文件流/缓冲区读取参数到格式化的相关变量中，同时注意变量要加“&”地址运算符，除了字符串变量通常不需要加“&”。另外，为了防止缓冲区溢出，参数格式应该指定长度如“%80s”之类。

6、字符的I/O

```
#include <stdio.h>
int getc(FILE *stream);
int fgetc(FILE *stream);
int getchar(void);
int ungetc(int c, FILE *stream);
int fputc(int c, FILE *stream);
int putc(int c, FILE *stream);
int putchar(int c);
```

注意：getchar 和 putchar 通常是由宏实现的，使用时可能会产生副作用。

7、字符串（不换行）的I/O

```
#include <stdio.h>
char *fgets(char *s, int size, FILE *stream);
char *gets(char *s);
int fputs(const char *s, FILE *stream);
int puts(const char *s);
```

注意 gets 可能会引起缓冲区溢出，最好使用 fgets 来代替，fgets 代替 gets 的最主要区别是 fgets 保留行终止符而 gets 不会保留。

8、文件的定位

```
#include <stdio.h>
int fseek(FILE *stream, long offset, int whence);
long ftell(FILE *stream);
int fgetpos(FILE *stream, fpos_t *pos);
int fsetpos(FILE *stream, fpos_t *pos);
void rewind(FILE *stream);
```

作用：fseek 以 whence 模式把当前位置定位到流 stream 的 offset 处，whence 包括 SEEK_SET, SEEK_CUR, SEEK_END。

ftell 返回当前位置(相对于文件起始位置)。

fgetpos 和 fsetpos 则是 ftell 和 fseek 的变体。

rewind 把 stream 的当前位置重置到起始处。



9、缓冲区控制

```
#include <stdio.h>
int fflush(FILE *stream);
int setbuf(FILE *stream, char *buf);
int setbuffer(FILE *stream, char *buf, size_t size);
int setlinebuf(FILE *stream);
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

作用：fflush 强制把缓冲区中剩余的输入到 stream 流上。

setbuf、setbuffer 和 setvbuf 都用于设置流所使用的缓冲区。其中 setbuf 功能比较弱，且不安全，可能会导致缓冲区溢出。setvbuf 除了可以实现 setbuffer 的功能外还能设置缓冲区的模式：_IONBUF（无缓冲），_IOLBF（行缓冲），_IOFBF（完全缓冲）。改变流的缓冲模式只需要调用 setvbuf 并缓冲地址为 NULL 即可。

10、删除和重命名文件

```
#include <stdio.h>
int remove(const char *pathname);
int rename(const char *oldpath, const char *newpath);
```

执行成功返回 0，失败返回-1 并设置 errno。

11、临时文件

ANSI C 函数：

```
#include <stdio.h>
FILE *tmpfile(void);
char *tmpnam(char *s);
```

作用：tmpfile 打开一个临时文件并返回其文件流指针。tmpname 则为所给的缓冲区生成一个临时的文件名。它们只能在/tmp、/var/tmp 之类下面创建临时文件。

POSIX C 函数：

```
#include <unistd.h>
int mkstemp(char *template);
char *mktemp(char *template);
```

作用：mkstemp 用模板指定路径，创建一个临时文件，并返回一个文件描述符。

12、目录操作

getcwd：返回当前目录



```
#include <unistd.h>
char *getcwd(char *buf, size_t size);
```

作用：把当前目录的绝对地址保存到 buf 中，buf 的大小为 size。如果 size 太小无法保存该地址，返回 NULL 并设置 errno 为 ERANGE。可以采取零 buf 为 NULL 并使 size 为负值来使 getcwd 调用 malloc 动态给 buf 分配，但是这种情况要特别注意使用后释放缓冲以防止内存泄漏。

chdir、fchdir：改变当前目录

```
#include <unistd.h>
int chdir(const char *path);
int fchdir(int fd);
```

mkdir：创建目录

```
#include <fcntl.h>
#include <unistd.h>
int mkdir(const char *pathname, mode_t mode);
```

作用：以 mode 为模式创建 pathname 目录。成功返回 0，失败返回-1 并设置 errno。

rmdir：删除空目录

```
#include <unistd.h>
int rmdir(const char *pathname);
```

删除空目录 pathname，成功返回 0，失败返回-1 并设置 errno。

13、获得目录列表：

```
#include <dirent.h>
#include <sys/types.h>
DIR *opendir(const char *pathname);
struct dirent *readdir(DIR *dir);
int rewinddir(DIR *dir);
int closedir(DIR *dir);
```

作用：opendir 打开一个目录文件并返回 DIR 流指针。指针指向 DIR 的初始位置。出错返回 NULL 并设置 errno。

readdir 通过移动 DIR 指针读取目录内容到 dirent 结构，出错或者读取到末尾时返回 NULL。其中文件名保存在 dirent.d_name[] 中。

rewinddir 把 DIR 指针重新定位到初始位置。

closedir 关闭 DIR 流指针。



进程与信号的相关函数

1、进程信息检测

1.1 getid 族函数:

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
pid_t getppid(void);
pid_t getuid(void);
pid_t geteuid(void);
pid_t getgid(void);
pid_t getegid(void);
```

作用:

getpid ——返回当前进程 pid;

getppid ——返回当前父进程 pid;

getuid ——返回运行当前进程用户的 UID;

geteuid ——返回运行当前进程用户的有效 UID(一般跟 uid 相同,但是在例如在 setUID 之后就有效 UID 就变成 root 超级用户的 UID)

getgid ——返回运行当前进程用户所在组的 GID;

getegid ——返回运行当前进程用户所在组有效 GID (setGID 之后和 GID 不同);

1.2 getlogin

返回运行当前进程的用户登录名。

```
#include <unistd.h>
char *getlogin(void);
```

1.3 getpwnam

输入一个有效的用户名, 返回一个记录了用户各种信息的 passwd 结构;

```
#include <pwd.h>
struct passwd *getpwnam(const char *name);
```

1.4 进程计时

```
#include <sys/times.h>
clock_t times(struct tms *buf);
```

times: 返回系统启动后流逝至今的 CPU 时间, 并设置输入的 tms 结构。tms 结构包括: tms_utime (用户模式下流逝的 CPU 时间), tms_stime (内核/系统模式



下流逝的 CPU 时间), `tms_cutime` (用户模式下流逝的子进程 CPU 时间), `tms_cstime` (内核/系统模式下流逝的子进程 CPU 时间)。

另有 `rusage` 结构有更多的资源利用信息。它可以用 `getrusage` 这个函数设置。它们都在头文件 `<sys/resource.h>` 中定义。

```
#include <sys/times.h>
#include <sys/resource.h>
#include <unistd.h>
int getrusage(int who, struct rusage *usage);
```

其中 `who` 为 `RUSAGE_SELF` 或者 `RUSAGE_CHILDREN`, 决定用当前进程还是子进程来设置 `rusage` 结构。

`rusage` 结构包括 `timeval` 结构的 `ru_utime` 和 `ru_stime`, 以及记录存储 IO 情况的 `ru_minflt` (引起 RAM 访问的次数)、`ru_majflt` (引起磁盘交换分区访问的次数), `ru_nswap` (读取的交换分区页数) 等。

`times` 调用返回的时间比 `getrusage` 要精确得多, 但 `getrusage` 给出的资源利用信息要更详细。

2、进程的创建

2.1 system

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

`system` 函数执行 `string` 所指的字符串。字符串一般为 `shell` 命令, 可以包括选项和参数。例如

```
system("ls -l hello.c");
```

如果 `string` 为 `NULL` 该函数返回非零值, 否则返回 0;

2.2 fork

```
#include <unistd.h>
pid_t fork(void);
```

`fork` 调用创建父进程的一个准确副本, 包括相同的 `UID`、`EUID`、`GID`、`EGID`、进程组 (例如用|执行的 `shell` 命令管道就是一个进程组, 用进程组 `PGID` 标识)、会话 `ID` (会话由一个或多个进程或进程组构成, 以惟一的 `session ID` 标识, 创建一个 `shell` 就是创建一个 `shell` 会话)、环境变量、资源、打开的文件和共享内存段等。但是没有继承父进程的文件锁和 `pending` 未决信号。



fork 执行成功后，在父进程中返回子进程的 PID，在子进程中返回 0。

fork 调用要注意进程代码中不应有依赖父进程或子进程的代码，否则可能会引起竞态乃至系统死锁。

2.3 exec函数族

exec 函数族包括 6 个函数：

```
#include <unistd.h>
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execlxe(const char *path, const char *arg,
const char *envp[]);
int execv(const char *path, const char *argv[]);
int execve(const char *path, const char *argv[],
const char *envp[]);
int execvp(const char *file, const char *argv[]);
```

execl 的第一个参数是包括路径的可执行文件，后面是列表参数，列表的第一个为命令 path，接着为参数列表，最后必须以 NULL 结束。

execlp 的第一个参数可以使用相对路径或者绝对路径。

execlxe，最后包括指向一个自定义环境变量列表的指针，此列表必须以 NULL 结束。

execv，v 表示 path 后面接收的是一个向量，即指向一个参数列表的指针，注意这个列表的最后一项必须为 NULL。

execve，path 后面接收一个参数列表向量，并可以指定一个环境变量列表向量。

execvp，第一个参数可以使用相对路径或者绝对路径，v 表示后面接收一个参数列表向量。

exec 被调用时会替换调用它的进程，直接返回到调用它的进程的父进程，如果出错，返回-1 并设置 errno。

2.4 popen

popen 使用 FIFO 管道执行外部程序。

```
#include <stdio.h>
FILE *popen(const char *command, const char *type);
int pclose(FILE *stream);
```

popen 通过 type 是 r 还是 w 确定 command 的输入/输出方向，r 和 w 是相对



command 的管道而言的。r 表示 command 从管道中读入，w 表示 command 通过管道输出到它的 stdout，popen 返回 FIFO 管道的文件流指针。pclose 则用于使用结束后关闭这个指针。

3. 进程控制

3.1 wait、waitpid

```
#include <sys/wait.h>
#include <sys/types.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

这两个函数收集子进程的退出状态，以避免它成为僵尸进程。status 为子进程的返回状态，0 或-1。pid 为要等待的子进程 pid，可能的值有-1(等待任何 PGID 为 PID 的绝对值的子进程)、1(等待任何子进程)、0(等待任何 PGID 等于调用进程的子进程)、>0(pid 等于 PID 的子进程)。option 包括 WNOHANG(没有子进程要退出时返回)和 WUNTRACED(子进程没有要报告的状态而返回)。

3.2 中止进程的函数

进程被中止的原因包括：在 main 函数里面执行了 return；执行了 exit；执行了 _exit；执行了 abort；被一个信号中止。

```
#include <stdlib.h>
int exit(int status);
```

exit 以 status 状态正常中止进程，如果有使用 atexit 登记了相关的处理程序，也可以执行。

_exit 和 exit 不同的地方是它是在 unistd.h 中生命，会立刻中止调用它的进程，而且不会执行 atexit 登记的程序。

abort 则立刻中止进程，如果系统允许还将发生 core dump 生成 core 文件，作为严重情况下使用。

使用 kill 函数是利用信号中止程序的一个例子。

```
#include <signal.h>
#include <sys/types.h>
int kill(pid_t pid, int sig);
```

pid 为要杀死的进程，sig 是要发送给该进程的信号，如果要杀死它，可以发送 SIGKILL、SIGTERM、SIGQUIT。



3.3 信号

信号是硬件中断的软模拟，用信号量标识，为正整数，其宏定义在 `signal.h`，均为 `SIG` 开头。

`kill` 命令或者 `kill` 函数可以发送。与发送相对的则是捕获和处理信号。此外还有产生 `generate`、递送 `deliver`(信号将要被处理)、未决 `pending`(信号已产生而未被递送的时间间隔)、忽略 `ignore`、部署 `disposition`(如何处理这个信号)等。多个信号可以用信号集合 `signal_set` 结构(定义在 `signal.h`)表示。使用 `mask` 掩码可以阻塞信号以禁止它被递送。而 `SIGKILL` 和 `SIGSTOP` 是唯一两个不能被进程捕获和忽略的信号。

`SIGALRM` 超时信号，通过 `alarm` 函数发送。

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

调用 `alarm` 的进程在 `seconds` 时间到后会捕获到一个 `SIGALRM` 信号。

`pause` 函数把调用它的进程挂起，直到捕获到一个信号。如果调用 `pause` 进程不能处理递送到的信号，则发生默认部署。`pause` 在捕获到信号后返回，而信号被递送后的处理会在 `pause` 返回前执行。`pause` 总返回-1 并设置 `errno`。

3.4 信号的创建和处理

```
#include <signal.h>
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
int sigismember(const sigset_t *set, int signum);
```

`sigemptyset`: 创建一个信号集合，该集合为空；

`sigfillset`: 创建一个信号集合，该集合为满；

`sigaddset`: 把信号 `signum` 添加到信号集合 `set` 中；

`sigdelset`: 把信号 `signum` 从信号集合 `set` 中删除；

`sigismember`: 测试信号 `signum` 是否在信号集合 `set` 中，如果是则返回 1，否则返回 0。

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

`sigprocmask` 设置或者修改信号掩码，`how` 为对 `set` 的可选处理，包括



SIG_BLOCK, SIG_UNBLOCK, SIG_SETMASK, NULL。

```
#include <signal.h>
int sigaction(int signum, const struct sigaction *act,
struct sigaction *oldact);
```

sigaction 为指定的信号 signum 设置一个信号处理器，结构 sigaction 描述了对该信号的部署。

```
#include <signal.h>
struct sigaction{
    void (*sa_handler)(int);
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restorer)(void);
}
```

sa_handler 规定了 signum 中的信号产生后要调用的处理器或者函数，输入为一个 int 类型参数，返回一个 void，或者也可以为 SIG_DFL 而引起 signum 的默认动作，SIG_IGN 忽略 sinnum；

sa_mask 为要阻塞的信号掩码的集合；

sa_flags 掩码修正 sa_handler 的行为，包括 SA_NOCLDSTOP(忽略子进程的 SIGSTOP 等信号)，SA_ONESHOT(登记的自定义信号处理器只执行一次，然后恢复信号的默认动作)，SA_RESTART(让可重启的系统调用起作用)，SA_NOMASK，SA_NODEFER。

sa_restorer: 已经废弃不用。

```
#include <signal.h>
int sigpending(sigset_t *set);
```

sigpending 检查是否有未决信号并设置到 set 中，如果挂起信号是为了执行某个操作，操作完成后可以用 sigpending 检查未决信号并处理之。否则简单接触阻塞就可以。

4. 进程的调度

```
#include <sched.h>
int sched_setscheduler(pid_t pid, int policy,
const struct sched_param *p);
int sched_getscheduler(pid_t pid);
int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);
int getpriority(int which, int who);
```



```
int setpriority(int which, int who, int prio);  
int nice(int inc);
```

`sched_setscheduler` 和 `sched_getscheduler` 分别设置和取得与某个特定进程相关的策略和参数。策略 `policy` 包括 `SCHED_OTHER`、`SCHED_FIFO`、`SCHED_RR`。后两者是用于特别看重时间的策略，会抢先于使用默认策略 `SCHED_OTHER` 的进行执行；

`sched_get_priority_max` 和 `sched_get_priority_min` 返回对于策略 `policy` 来说最大或最小的优先级。注意优先级的值越小，其级别越高；

`setpriority` 设置进程(`which=PRIO_PROCESS`)、进程组(`which=PRIO_PGRP`)、用户(`which=PRIO_USER`)的动态优先级；

`getpriority` 则返回匹配进程的最高优先级(最小值)；

`nice` 通过为当前的进程优先级增加一个 `inc` 而降低其优先级。



POSIX线程基本概念

线程编程在 `smp` 体系结构处理并发时会被提及的比较多。它可以实现并发多道操作，常被称为轻量级的进程，因为它可以共享进程资源，省去了多进程切换时内核的上下文切换所用的花销。

多数 Linux 编程教材里面对 POSIX 线程的介绍占的篇幅并不多。Linux 的 `fork` 对多进程的有较好的优化技术，而 `__clone` 系统调用相当于使用进程的方法实现线程。进程的另一个好处是稳定性，进程在退出时操作系统自动回收资源，而共享资源的线程需要程序员自己设计释放资源，积累的错误可能会使得程序崩溃。

线程和进程如何使用的关系，在不少 unix 尤其是 linux 编程的相关社区依然是经久不息的话题。[这篇文章](#)大致说了一下 linux 编程为什么多用进程而少用线程。而 Gentoo 创始人 Daniel Robbins 的[这一篇](#)相当好的线程入门文章。还有[这篇](#)。

我只是初学者，下面是我的学习笔记。

1、__clone函数调用

```
#include <sched.h>

int __clone(int (*fn)(void *fnarg), void *child_stack, int flags,
void *arg);
```

Linux 可以通过 `__clone` 函数调用创建子进程来实现线程，`__clone` 同时也可以实现 `fork`。

`__clone` 的第一个参数是执行子进程时的调用函数指针，这个函数用一个 `void` 类型的指针作为参数，返回 `int` 整型；

第二个参数为为子进程分配的堆栈指针，如果想让操作系统为它来分配，则设置为 `NULL`；

第三个为 `CLONE` 标志，作为调用选项，`flags` 包括：`CLONE_VM`（是否设置则共享内存镜像，否则就类似于 `fork`，子进程拷贝一份父进程的副本）、`CLONE_FS`（是否共享相同的根文件系统，当前动作目录和 `umask`）、`CLONE_FILES`（是否共享文件描述符，否则子进程依然会继承父进程的文件描述符表，但它们的读/写/打开/关闭操作互不影响）、`CLONE_SIGHAND`（是否共享设置在父进程上的信号处理器，否则子进程自己拷贝一份副本，它们的 `sigaction` 互不影响）、`CLONE_PID`（子进程是否继续使用父进程的 `pid`）；

最后一个参数 `arg` 可以设置为传递给 `*fn` 子函数的参数，注意类型是 `void`，



这意味着它是什么类型可以自己选择，包括 NULL。

2、POSIX线程库的pthread API

pthread_create: 创建一个线程

```
#include <pthread.h>
int pthread_create(pthread_t *thread, pthread_attr_t *attr,
void *(*start_routine)(void *), void *arg);
```

第一个参数为要创建的线程结构 pthread_t，第二个参数为线程的相关属性，第三个参数是该线程要执行的函数的指针，第四个则是传递给*start_routine 的参数，可以自行选择它的取值和用法。

pthread_create 执行成功返回 0，并把 tid 线程标识符存放到 pthread_t 结构中；否则返回一个非零值并设置 errno。

pthread_exit: 终止当前线程

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

pthread_exit 退出当前线程，退出之前将调用 pthread_cleanup_push。它在线程的最上层函数的最后是被隐式调用的，这时可以加一个 retval 参数显式调用之，以供 pthread_join 参考。

pthread_join: 挂起当前线程

```
#include <pthread.h>
int pthread_join(pthread_t th, void **thread_return);
```

pthread_join 把当前线程挂起直到指定的线程 th 终止，thread_return 为 th 终止时的参数，如果没有显式指定，则为 NULL。

pthread_cancel: 撤消一个线程

```
#include <pthread.h>
int pthread_cancel(pthread_t thread);
int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
void pthread_testcancel(void);
```

pthread_cancel 用于撤消线程 thread。

pthread_setcancelstate 可以用于设置当前线程的撤消状态，包括 PTHREAD_CANCEL_ENABLE 和 PTHREAD_CANCEL_DISABLE，允许或者忽略撤消。在一些关键操作中可以设置它以避免被另外一个线程用 pthread_cancel 来撤消。



`pthread_setcanceltype` 设置当前线程的撤消类型，包括：

`PTHREAD_CANCEL_ASYNCHRONOUS` 立即撤消

`PTHREAD_CANCEL_DEFERRED` 延迟至撤消点。

可以通过调用 `pthread_testcancel` 或者使用条件变量来设置撤消点。

3、线程属性

线程属性包括：

`detachstate` —— 分离或者切入状态，它的值有 `PTHREAD_CREATE_JOINABLE`(默认值)、`PTHREAD_CREATE_DETACHED`；

`schedpolicy` —— 调度策略，取值有 `SCHED_OTHER`(默认值)，`SCHED_FIFO`，`SCHED_FIFO`；

`schedparam` —— 跟调度策略有关；

`inheritsched` —— `PTHREAD_EXPLICIT_SCHED`(默认值)，
`PTHREAD_INHERIT_SCHED`；

`scope` —— 时间片，取值有 `PTHREAD_SCOPE_SYSTEM`(默认值)，每个线程一个系统时间片，`PTHREAD_SCOPE_PROCESS` 线程共享系统时间片。

线程属性对象的类型为 `pthread_attr_t`，通过 `pthread_attr_t` 函数族操作线程属性对象。

4、pthread cleanup宏

`pthread_cleanup` 宏主要用于处理线程的退出状态，`pthread_exit` 和 `pthread_join` 等可以用它来作参数，包括

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

这些宏主要用于线程结束时释放有关资源，用 `pthread_exit` 调用 `pthread_cleanup_push` 进行处理时，要注意用完后再用对应的 `pthread_cleanup_pop` 把它从堆栈弹出。

5、互斥mutex

由于线程是共享资源的，所以互斥就显得相当重要。互斥提供了对互斥对象



上锁的方法以获得对此资源的独占，其它企图对此互斥加锁的线程则会被阻塞而挂起，直到对资源加锁的线程解锁为止。

在使用线程编程时，不必处处都使用互斥，否则会失去并发线程的意义，例如，在对使用只能串行单独访问的资源下方使用互斥。

互斥对象在 `pthread.h` 中定义为 `pthread_mutex_t`。它的系统调用主要包括：

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *mutex,
const pthread_mutex_attr_t *mutexattr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

`pthread_mutex_init` 创建一个互斥对象，并用指定的属性初始化这个互斥对象，初始化属性包括：`PTHREAD_MUTEX_INITIALIZER` 创建快速互斥，这种互斥执行简单的加锁和解锁，在加锁后阻塞另一个要给它上锁的线程；`PTHREAD_RECURSIVE_MUTEX_INITIALIZER` 创建递归互斥，这种互斥将给加锁计数，解锁时需要调用同样次数的 `pthread_mutex_unlock`；

`PTHREAD_ERRORCHECK_MUTEX_INITIALIZER` 创建检错互斥，这种互斥在被锁上后会向试图给它加锁的线程返回一个 `EDEADLK` 错误代码而不阻塞。

`pthread_mutex_lock` 和 `pthread_mutex_unlock` 则为加锁和解锁指定的互斥对象，`pthread_mutex_trylock` 和 `pthread_mutex_lock` 的不同是如果互斥对象已上锁不会被阻塞而是返回一个 `EBUSY` 错误代码，`pthread_mutex_destroy` 析构指针 `mutex` 并释放相关资源。这几个调用成功都返回 0，失败返回一个非零的错误代码。

6、条件变量

线程用条件变量对象来阻塞自己以等待某个指定条件发生，条件对象在 `<pthread.h>` 中定义为 `pthread_cond_t`。

```
#include <pthread.h>
pthread_cond_t cond = pthread_cond_initializer;
int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t
*cond_attr);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```



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

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

int pthread_cond_destroy(pthread_cond_t *cond);
```

`pthread_cond_init` 创建一个指向条件变量的指针，并用一个参数初始化该变量，这个参数在 Linux 下被忽略而使用 `PTHREAD_COND_INITIALIZER`，而 `pthread_cond_t` 则析构该指针并释放相关系统资源。

线程在锁定一个互斥后，如果需要某个条件到来，则应调用等待所需条件，`pthread_cond_wait` 将检查指定的条件，没有则把当前线程挂起，并解锁指定的互斥，在条件到达后重新锁定该互斥，并唤醒被挂起的线程。`pthread_cond_timedwait` 则为计时等待，使用的 `abstime` 这个参数为和兼容 `time()` 返回值的绝对时间，即以经典的 UNIX 纪元时间 1970-01-01 起至今的秒数。如果这个时间前等待条件未发生，则结束等待并返回 `ETIMEDOUT` 代码。这两个函数都将被设置为撤消点。

`pthread_cond_signal` 和 `pthread_cond_broadcast` 用在一个线程解锁互斥后唤醒等待条件 `cond` 的线程，不同之处在于前者按顺序唤醒第一个进入等待队列的线程，后者唤醒等待该条件的所有线程。



内存管理相关函数

1、经典的C动态内存管理相关函数

标准 C 提供了 malloc, calloc, realloc, free 等基于内存堆的管理函数，负责分配可用内存以及释放用过的内存。这些函数本身只负责告诉调用程序，当前有你想要 size 大小的可用内存块，它的首地址是 xxxxx，并不会检查指针的使用是否越过了这个 size 或者说是 offset，这个需要程序员自己去检查。

```
#include <stdlib.h>
void *malloc(size_t size);
```

malloc 的作用分配一块大小为 size 个字节的可用内存块，并返回首地址。不能分配的时候返回 NULL。

```
#include <stdlib.h>
void calloc(size_t nmemb, size_t size);
```

calloc 的作用是分配并初始化内存块，返回一个指向 nmemb 块数组的指针，每块大小为 size 个字节。它和 malloc 的主要不同之处是会初始化（清零）分配到的内存。

```
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
```

realloc 以 ptr 所指地址为首址，分配 size 个字节的内存，并返回 ptr 所指地址。realloc 不会初始化分配到的内存块，如果 ptr 为 NULL 则相当于 malloc，如果 size 为 NULL 则相当于 free(ptr)。不能分配返回 NULL。

```
#include <stdlib.h>
void free(void *ptr);
```

free 清除 ptr 所指向的地址，它只作清除的工作，并告诉系统，这块地址已经被释放和清除，可以重新被分配。

一个带 bug 的程序：

```
/*
 * mallocfree.c - try to access illegally an address has wrong
 * allocated
 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(void)
```



```
{  
    char *ptr = malloc(sizeof(char) * 5);  
    strcpy(ptr, "123456789");  
    printf("ptr : %s\n", ptr);  
    printf("freeing ptr\n");  
    free(ptr);  
    sleep(1);  
    printf("ptr : %s\n", ptr);  
    printf("trying to write the address has freed\n");  
    strcpy(ptr, "54321");  
    sleep(1);  
    printf("ptr : %s\n", ptr);  
    return 0;  
}
```

sleep 程序在里面是为了让肉眼更直观的分析程序的运行。执行 make mallocfree, 生成可执行文件, 并运行之:

```
# ./mallocfree  
ptr : 123456789  
freeing ptr  
ptr :  
trying to write the address has freed  
ptr : 54321
```

在这里, ptr 使用的地址已经越过了 malloc 分配给它的界限, 而且指针在释放以后继续使用。未经申请便使用的存储区域其内容是未知的, 将存在缓冲区溢出的隐患。

该程序在 Cygwin 下运行的结果:

```
# ./test  
ptr : 123456789  
freeing ptr  
ptr :  
    a  
    a  
trying to write the address has freed  
ptr : 54321  
Segmentation fault (core dumped)
```



另一个程序：

```
/*
 * reallocit.c - realloc a pointer and free it
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char *str1 = malloc(sizeof(char) * 4), *str2;
    strcpy(str1, "abc");
    printf("str1: %s\n", str1);
    str2 = realloc(str1, sizeof(char) * 7);
    printf("str2: %s\n", str2);
    free(str1);
    return 0;
}
```

这个程序先给 str1 分配一段内存，然后给重新分配这段内存，并分配给 str2，最后释放 str1 所指的内存。注意此时释放的是 realloc 后的 sizeof(char) * 7 个字节的空间，而不是原来 malloc 给 str1 的 sizeof(char) * 4 个字节的空间。注意在此代码中，如果再次 free(str2)，编译运行后程序将报错，因为此地址已被释放；如果在重新分配内存之前继续使用 str2 指针，可能会造成缓冲区溢出。上面的函数都是在堆中分配内存，而 alloca 是在进程栈中获得内存，它的功能也是分配一块未经初始化的内存。

```
#include <stdlib.h>
void *alloca(size_t size);
```

2、Linux的内存映像管理函数

内存映像的意思是把磁盘文件映像到内存中，以加速 I/O 操作和便于共享数据，而在共享数据时，为了避免多个进程同时读写数据引起不可预料的后果，通常使用锁或者信号灯等机制实现对共享对象的序列化访问。Linux 提供了在 sys/mman.h 中定义的一系列内存映像管理的函数。

```
#include <sys/mman.h>
void *mmap(void *start, size_t length, int prot, int flags,
int fd, off_t offset);
```



`mmap` 把打开的磁盘文件 `fd` 从 `offset` 开始, 映像到内存 `start` 处, 大小为 `length`。成功返回该映像的指针, 失败返回 -1 并设置相应的 `errno`。

映像可选的保护模式 `prot` 包括: `PROT_NONE`, `PROT_READ`, `PROT_WRITE`, `PROT_EXEC` 等。

映像的可选属性 `flags` 包括: `MAP_FIXED`, `MAP_PRIVATE`, `MAP_SHARED`, `MAP_ANON`, `MAP_DENYWRITE`, `MAP_GROWSDOWN`, `MAP_LOCKED` 等。

其中 `MAP_PRIVATE` 或者 `MAP_SHARED` 两者必须且只能选择一个, 其它都是可选值, 用逻辑或添加。`MAP_FIXED` 强制使用 `start` 指定的地址, 否则执行失败, 如果没有使用这个选项, 则 `mmap` 在 `start` 不可用时会尝试把 `mmap` 放到其它地方。`MAP_LOCKED` 只用在 `root` 权限的进程才能使用, 以防止锁定所有可用内存的恶意攻击。

```
#include <sys/mman.h>
int munmap(void *start, size_t length);
```

`munmap` 解除从 `start` 开始大小为 `length` 个字节的内存映像并释放内存, 如果在 `munmap` 之后试图继续访问 `start`, 将会产生段错误。在进程终止运行时, 所有的内存映像会被自动解除。

```
#include <sys/mman.h>
int msync(const void *start, size_t length, int flags);
```

`msync` 把从 `start` 开始的大小为 `length` 个字节的内存映像同步到磁盘, `flags` 包括: `MS_ASYNC`, `MS_SYNC`, `MS_INVALIDATE`。成功返回 0, 失败返回 -1 并设置 `errno`。

```
#include <sys/mman.h>
int mprotect(const void *start, size_t len, int prot);
```

`mprotect` 修改 `start` 开始的大小为 `len` 个字节的内存映像的保护模式为 `prot`。成功返回 0, 失败返回 -1 并设置 `errno`。

```
#include <sys/mman.h>
int mlock(const void *start, size_t len);
int munlock(void *start, size_t len);
int mlockall(int flags);
int munlockall(void);
```

以上函数对指定的内存映像加锁和解锁, 其中 `mlockall` 的 `flags` 包括 `MCL_CURRENT` 和 `MCL_FUTURE`。只有 `root` 权限才能使用它们。



```
#include <sys/mman.h>
void *mremap(void *old_addr, size_t old_len, size_t new_len,
unsigned long flags);
```

mremap 用指定的 flags 把地址在 old_addr 的内存映像大小从 old_len 调整为 new_len, flags 如果为 MREMAP_MAYMOVE 则调整此内存映像的地址。成功返回新地址, 失败返回 NULL。



进程间通信IPC

1、管道

管道特指无名管道，如 shell 中的管道操作符“|”就是一个无名管道。它只有一个传输方向。

```
#include <unistd.h>
int pipe(int filedes[2]);
```

pipe 系统调用用于建立一个无名管道。管道是一个 linux 文件，所以对 pipe 创建的管道可以用 open, close, write, read, close 等函数来访问。它使用两个文件描述符 filedes[0] 和 filedes[1] 分别对其进行读出和写入，且 filedes[0] 只读(O_RDONLY)，filedes[1] 只写(O_WRONLY)。而且读或者写的时候，必须关掉另外一个文件描述符。pipe 成功返回 0，出错返回 -1 并设置 errno。

```
#include <stdio.h>
FILE *popen(const char *command, const char *mode);
int pclose(FILE *stream);
```

popen 和 pclose 是标准 C 中更简易的管道实现。popen 将创建一个管道并 fork 一个用 exec 执行 command 的子进程，mode 决定了对这个 command 的标准输入输出的读(r)或者写(w)。

popen 成功返回一个文件流指针供读取 command 的输出使用。失败返回-1 并设置 errno。

pclose 则关闭该文件流。

2、FIFO

FIFO 也称为有名管道，使用前者为称呼的更多。

在 shell 里面可以通过 mkfifo 创建一个有名管道，对有名管道的处理在 shell 里面通常可以使用“>”和“<”这两个重定向操作符。例如

```
$ mkfifo -m 644 fifo1
$ cat < fifo1&
$ ls > fifo1
```

-m 表示 fifo 的访问模式即权限位。cat 和 ls 的重定向可以在两个 shell 窗口中分别执行以使对管道的作用更直观一些。例如在 x 下的终端执行 ls > fifo1，然后用 ctrl+alt+f1 进文本控制台执行 cat < fifo1 接收。

在 C 中创建 fifo 的系统调用和 shell 中很类似：



```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
```

pathname 即要创建 FIFO 的路径和文件名，mode 即访问模式。mode 会被当前进程的 umask 修改。对 FIFO 的打开、关闭、读取、写入、删除的使用分别通过使用 open, close, read, write, unlink 这些操作文件的系统调用来完成。注意 FIFO 的两端在读写前都要先打开，且 O_NONBLOCK 标志在 FIFO 的另一端没有对应的输出/输入时会导致 write/read 调用立即返回。

3、SysV IPC

众所周知，70 年代后主流的 UNIX 主要分为 System V 和 BSD 两大分支，现代的 UNIX 以及类 UNIX 操作系统基本上都是这两大支流的综合以及进化。BSD 最有名的是它的套接字 socket 及其 TCP/IP 协议栈，而 SysV 最早让人瞩目的特性估计就是它的 IPC 对象机制了。

IPC 不属于 Linux 下的文件概念范畴，它只存在于内核。通过使用 ipcs 和 ipcrm 这两个专门的 shell 命令来查看和管理。故 IPC 对象也需要专门的系统调用而不能使用 open, read, write 等操作文件的函数来使用。

SysV 的 IPC 对象包括共享内存、消息队列和信号灯三种。通过 ipcget 函数 (shmget, msgget, semget) 创建它们，IPC 对象一个特性是使用了关键字对象 key (类型为 key_t，显然定义于 sys/types.h)，get 函数可以通过识别关键字而判断是创建新的 IPC 对象还是链接到一个已有的 IPC 对象。

如何使用 key 有三个传统方法：

用 IPC_PRIVATE 作为关键字，保证创建新的 IPC 结构。

把关键字保存在公共的头文件中。这可以保证包含这个头文件的程序都能访问到相同的关键字，但进程一般无法判断要创建还是要使用一个已有的 IPC 结构，而且可能出现关键字已经被占用的情况。

使用 ftok 函数。安装 manpages-posix manpages-posix-dev 这些包后可以用 man 3 ftok 来查看 ftok 的用法。

```
#include <sys/types.h>
#include <sys/ipc.h>
key_t ftok(const char *pathname, int proj_id);
```

ftok 通过输入一个路径和一个项目标识符来生成关键字，可以把这个路径和



项目标识符放到公共的头文件或者预定义的配置文件中。

ftok 不能保证生成唯一的關鍵字，存在着潜在隐患，建议尽量不要使用并忽略它。

除了关键字外，IPC 还接受一个 flag 标志控制 get 的行为，其中 IPC_CREAT 这个标志在指定关键字还没有 IPC 对象时将创建一个新的 IPC 对象。

IPC 使用模式(mode_t mode)描述 IPC 对象的读或写的属性(信号灯是读 SEM_R 和修改 SEM_A)，但没有执行这个属性。

使用 ipcget 创建队列后，将产生一个 ipc_perms 结构记录这个 ipc 的属性。并可以通过 ipcctl (shmctl, msgctl, semctl) 用 IPC_SET 标志去修改它们。IPC_SET 和 IPC_RMID 两个标志只能在进程 id 等于 uid 或者 cuid 或者为超级用户时才能使用。

```
struct ipc_perms{
    uid_t uid;
    gid_t gid;
    uid_t cuid;
    gid_t cgid;
    mode_t mode;
    ulong seq;
    key_t key;
}
```

ipc 使用 WAIT 等待这个术语对应 linux 文件的 BLOCK 阻塞状态。

4、共享内存

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, int size, int flags);
```

shmget 创建一个共享内存 IPC，size 为内存段大小，最大不超过处理器本身的页大小(BUFSZ)，flags 为 IPC_CREAT, IPC_EXCL,以及权限位(属性模式)逻辑或的结果。shmget 执行成功，则返回共享内存 IPC 的标识符。否则返回-1 并设置 errno。

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
char *shmat(int shmid, char *shmaddr, int flags);
```



```
int shmdt(char *shmaddr);
```

shmat 把共享内存 IPC 附加映像到当前进程的地址空间 shmaddr 中去，shmaddr 为了让系统自动分配可以用的地址，一般都设置为 0，flags 设置为 SHM_RDONLY 则限制该段为只读，否则默认是可读写的。shmat 成功返回该映像的地址，失败返回-1 并设置 errno。

shmdt 在共享内存 IPC 使用完毕后，把映像地址 shmaddr 分离出进程的地址空间。

5、消息队列

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgget(key_t key, int flags);
```

msgget 仅仅使用 key 和 flags 来创建消息队列 IPC，并返回该消息的标识符。

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgsnd(int msqid, const void *ptr, size_t nbytes, int flags);
```

mgsnd 把 ptr 所指向的 msgbuf 结构的消息以 nbytes 添加到队列 msqid 的末尾，并使用 flags 控制其行为。

msgbuf 在<sys/msg.h>中的定义为

```
struct msgbuf{
    long mtype;
    char mtext[BUFSZ];
}
```

msgbuf 结构只是一个模板，它的 mtext 成员的长度并没有被限定而应程序员根据情况自己定。如果 msgbuf 以 NULL 结尾则送到队列时 NULL 要被截掉。

flags 只能是 0 或者 IPC_NOWAIT。后者说明 msgsnd 在队列满时立即返回，否则将等待直到：1) 队列有了可以容纳此消息的空间；2) 队列被删除；3) 捕捉到一个信号，处理后返回。

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgrcv(int msqid, void *ptr, size_t nbytes, long type,
int flags);
```



`msgrcv` 从队列 `msqid` 中读取一条 `nbytes` 长的消息并把它和消息类型一起写到 `ptr` 所指向的结构中。`type` 决定了从队列的什么地方取出消息，为 0 则按先进先出返回队列的第一条消息；大于 0 则与消息类型匹配，返回匹配的第一条消息，可以用来设置优先级；小于 0 则取绝对值，返回一个消息类型匹配为小于等于这个绝对值的第一条消息。如果 `flags` 设置了 `MSG_NOERROR` 位，在返回的消息比 `nbytes` 多时将被截短；如果 `flags` 为 `IPC_NOWAIT`，则在没有收到消息时立即返回，否则将等待直到：

1) 有了指定 `type` 的消息；2) 队列被删除；3) 捕捉到一个信号并处理后返回。

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

`msgctl` 用于对 `msqid` 的控制，`cmd` 包括了：

`IPC_RMID` —— 删除队列；

`IPC_STAT` —— 把队列的 `msqid_ds` 结构填充到 `buf`；

`IPC_SET` —— 修改 `msg_perms` 设置队列的属性(`UID`, `GID`, 访问模式, 队列最大字节数等)。

7、信号灯

信号灯也可以说是一种锁，但它可以用来控制除了文件以外的更多资源。信号灯的初始值一般为一个正数，决定了可以分配的资源数，为进程分配一个资源后自减，减到 0 后被锁住。`SysV IPC` 要求信号灯必须定义为一个集合。创建信号量时则指定此集合中的值。

双态信号灯是最简单的一种，0 表示锁定，无资源；1 表示解锁，有一个可用资源。

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget(key_t key, int nsems, int flags);
```

`semget` 通过 `key`(`IPC_PRIVATE` 等)创建 `nsems` 个（集合）新的或者访问一个（`nsems` 为 0）已经存在的信号灯，并使用 `flags`(`IPC_CREAT`、8 进制访问模式等)控制它的行为。成功在返回信号灯描述符，失败返回 -1 并设置 `errno`。



操作信号灯的常用函数为 `semop`。

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semop(int semid, struct sembuf *semops, unsigned nops);
```

`nops` 为 `sembuf` 结构数组集合的元素个数。`sembuf` 结构的定义为

```
struct sembuf{
    short sem_num;
    short sem_op;
    short sem_flg;
}
```

这里 `sem_num` 是信号灯在集合中的编号，从 0 开始。

`sem_op` 为正数则释放信号灯控制的资源并增加信号灯的值；`sem_op` 为负数则先考察信号灯减去 `sem_op` 绝对值的结果：大于 0 则使用信号灯，等于 0 则等待资源被释放，小于 0 则等待到信号灯的值增加到大于等于 `sem_op` 绝对值为止；`sem_op` 为 0 则进程等待信号灯的值 0 然后返回。

上面说的等待都是在 `sem_flg` 没有 `IPC_NOWAIT` 的前提的，`sem_flg` 标志包括 `IPC_NOWAIT` 和 `SEM_UNDO` 等。

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semctl(int semid, int semnum, int cmd, union semun arg);
```

`semctl` 使用 `cmd` 控制 `semid` 中的 `semnum` 信号灯，并以联合 `arg` 中的成员作为参数。

联合 `semun` 的定义为：

```
union semun{
    int val;
    struct semid_ds *buf;
    unsigned short int *array;
}
```

`cmd` 包括了

- GETVAL —— 返回信号灯当前解锁或者加锁的状态；
- SETVAL —— 以联合 `arg` 的成员 `val`(即 `arg.val`)设置信号灯的当前状态；
- GETPID —— 返回上次调用 `semop` 的进程的 PID；



- GETNCNT —— 返回正在信号灯上等待的进程数;
- GETZCNT —— 返回等待信号灯的值为 0 的进程数;
- GETALL —— 返回和 `semid` 关联的集合中所有信号灯的值;
- SETALL —— 以联合 `arg` 的成员 `array`(即 `arg.array`)设置和 `semid` 关联的集合中所有信号灯的值;
- IPC_RMID —— 删除含 `semid` 的信号灯;
- IPC_SET —— 设置信号灯的访问模式(权限位);
- IPC_STAT —— 复制信号灯的 `semid_ds` 到 `arg.buf` 中。



守护进程程序设计的基本要点

守护进程 `daemon` 是系统里面一种特殊的进程。它没有与标准 IO 交互的 `shell` 接口，父进程是 `init`，它将在系统中永久驻留直到被强行中止，例如收到 `kill` 的信号。典型的守护进程随系统的启动而启动，随系统的停止而停止。

创建一个工作良好的守护进程需要做以下工作：

1、执行 `fork()`，并使父进程退出，使得子进程成为一个被 `init` 接管的孤儿进程；

```
pid = fork();
if (pid < 0)
{
    perror("fork");
    exit(EXIT_FAILURE);
}
if (pid > 0)
    exit(EXIT_SUCCESS);
```

2、使用 `setsid()` 创建新的会话；

```
if (sid = setsid() < 0)
    perror("setsid");
```

3、使用 `chdir()` 把根目录设置为当前的工作目录；

```
if (chdir("/") < 0)
    perror("chdir");
```

4、清除 `umask`，避免守护进程受到继承的 `umask` 的权限的干扰：

```
umask(0);
```

5、关闭不需要的文件描述符；

```
close(STDOUT_FILENO);
close(STDIN_FILENO);
close(STDERR_FILENO);
```

6、可能还需要读取配置文件。一般使用一个宏来定义配置文件的路径。然后使用一个单独的函数对配置文件进行处理。

例如

```
#define RCFILE "/etc/myprocd.conf"
```

完成上面工作，需要包含的头文件包括 `stdio.h`, `stdlib.h`, `errno.h`, `unistd.h` 等。

做完上述工作，这个守护进程就可以开始做其它相关处理了。

由于守护进程没有标准输入输出，运行产生的相关有用信息可以使用一系列



的 `syslog` 调用，设置 `LOG_DAEMON` 功能标志可以将守护进程 `myprocd` 写入到 `/var/log/myprocd.log` 里。

要注意对 `kill` 信号的处理，如 `SIGHUP`, `SIGCHLD` 等。例如，最好在上面的第二步的时候阻塞掉 `SIGNU` 以避免收到此信号导致进程重启。相关工作处理完毕再重新部署相关的未决信号。