UNIX 基础知识

一、 UNIX 体系结构

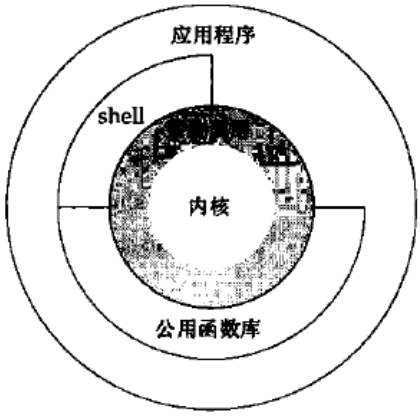

图 1-1　UNIX 操作系统的体系结构

二、 登录

1. 登录名
2. shell：UNIX 系统中常见的 shell

三、 文件和目录

1. 文件系统
2. 文件名
3. 路径名

实例：列出一个目录中的所有文件的函数实现：

```
#include "apue.h"
#include <dirent.h>

int main(int argc,char *argv[]) {
    DIR *dp;
    struct dirent *dirp;

    if(argc != 2)
        err_quit("usage:ls directory_name");

    if((dp = opendir(argv[1])) == NULL)
        err_sys("can't open %s",argv[1]);
```

```
    while((dirp = readdir(dp)) != NULL)
        printf("%s\n",dirp->d_name);

    closedir(dp);
    exit(0);
    }
```

4. 工作目录
5. 起始目录

四、 输入和输出

1. 文件描述符
2. 标准输入、标准输出被标准错误
3. 不带缓冲的 I/O

函数 open、read、write、lseek 以及 close 提供了不带缓冲的 I/O。这些函数都使用文件描述符。

实例：将标准输入复制到标准输出

```
#include "apue.h"

#define BUFFSIZE 4096

int main(void) {
    int n;
    char buf[BUFFSIZE];

    while ((n = read(STDIN_FILENO,buf,BUFFSIZE)) > 0)
        if (write(STDOUT_FILENO,buf,n) != n)
            err_sys("write error");

    if(n < 0)
        err_sys("read error");

    exit(0);

    }
```

4. 标准 I/O

标准 I/O 函数为那些不带缓冲区的 I/O 函数提供了一个带缓冲区的接口。使用标准 I/O 函数无需担心如何选取最佳的缓冲区大小，比如上例中的 BUFFSIZE 常量的大小。使用标准 I/O 函数还简化了对输入行的处理（常常发生在 UNIX 的应用程序中）。例如，fgets 函数读取一个完整的行，而 read 函数读取指定字节数。以后我们会了解到，标准 I/O 函数库提供了使我们能够控制该库所使用的缓冲风格的函数。

我们最熟悉的标准 I/O 函数是 printf。在调用 printf 的程序中，总是包含<stdio.h>（在本书中，盖头文件包含在 apue.h 中），该头文件包含了所有的标准 I/O 函数的原型。

实例：用标准 I/O 将标准输入复制到标准输出

```
    #include "apue.h"

int main(void) {
    int c;
    while ((c = getc(stdin)) != EOF)
        if (putc(c,stdout) == EOF)
            err_sys("output error");

        if (ferror(stdin))
            err_sys("input error");

        exit(0);
}
```

函数 getc 一次读取一个字符，然后函数 putc 将此字符写到标准输出。读到输入的最后一个字节时，getc 返回常量 EOF（该常量在<stdio.h>中定义）。标准 I/O 常量 stdin 和 stdout 也在头文件<stdio.h>中定义，它们分别表示标准输入和标准输出。

五、 程序和进程

1. 程序

程序（program）是一个存储在磁盘上某个文件中的可执行文件。内核使用 exec 函数（7 个 exec 函数之一），将程序读入内核，并执行程序。后面将说明这些 exec 函数。

2. 进程和进程 ID

程序的执行实例被称为进程（process）。某些操作系统用任务（task）表示正在被执行的程序。

UNIX 系统确保每个进程都有一个唯一地数字标识符，称为进程 ID（process ID）。进程 ID 总是一个非负数。

实例：打印进程 ID

```
#include "apue.h"

int main(void) {
    printf("hello world from process ID %ld\n",(long)getpid());
    exit(0);
}
```

3. 进程控制

有 3 个用于进程控制的主要函数：fork、exec 和 waitpid。（exec 函数有 7 种变体，但经常把它们统称为 exec 函数。）

实例：从标准输入读取命令，然后执行这些命令。它类似于 shell 程序的基本实施部分

```
#include "apue.h"
#include <sys/wait.h>

int main(void) {
    char buf[MAXLINE]; /*from apue.h*/
    pid_t pid;
    int  status;

    printf("%% "); /*print prompt (printf requires %% to print %)*/
    while (fgets(buf,MAXLINE,stdin) != NULL) {
        if (buf[strlen(buf) - 1] == '\n')
            buf[strlen(buf) - 1] = 0; /* replace newline with null */

        if ((pid = fork()) < 0) {
            err_sys("fork error");
        } else if (pid == 0) { /* child */

            execlp(buf,buf,(char*)0);
            err_ret("could't execute: %s",buf);
            exit(127);
        }

        /* parent */
        if ((pid = waitpid(pid,&status,0))<0)
            err_sys("waitpid error");
        printf("%% ");
```

```
    }
    exit(0);
}
```

4. 进程和线程 ID

通常，一个进程只有一个控制线程（thread）---某一时刻执行的一组机器指令。对于某些问题，如果有多个控制线程分别作用于它的不同部分，那么解决起来就容易得多。另外，多个控制线程也可以充分利用多处理器系统的并行能力。

一个进程内的所有线程共享同一地址空间、文件描述符、栈以及与进程相关的属性。因为它们能访问同一存储区，所以各线程在访问共享数据时需要采取同步措施以避免不一致性。

与进程相同，线程也用 ID 标识。但是，线程 ID 只在它所属的进程内起作用。一个进程中的线程 ID 在另一个进程中没有意义。当在一进程中对某个特定线程进行处理时，我们可以使用该线程的 ID 引用它。

控制线程的函数与控制进程的函数类似，但另有一套。线程模型是在进程模型建立很久以后才被引入 UNIX 系统中的，然而这两种模型之间存在复杂的交互，我们将在后面进行说明。

六、 出错处理

1. 错误码---errno

```
#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 */
#define EDEADLK 35 /* Resource deadlock would occur */
#define ENAMETOOLONG 36 /* File name too long */
#define ENOLCK 37 /* No record locks available */
#define ENOSYS 38 /* Function not implemented */
#define ENOTEMPTY 39 /* Directory not empty */
#define ELOOP 40 /* Too many symbolic links encountered */
#define EWOULDBLOCK EAGAIN /* Operation would block */
#define ENOMSG 42 /* No message of desired type */
#define EIDRM 43 /* Identifier removed */
#define ECHRNG 44 /* Channel number out of range */
#define EL2NSYNC 45 /* Level 2 not synchronized */
#define EL3HLT 46 /* Level 3 halted */
#define EL3RST 47 /* Level 3 reset */
#define ELNRNG 48 /* Link number out of range */
#define EUNATCH 49 /* Protocol driver not attached */
#define ENOCSI 50 /* No CSI structure available */
#define EL2HLT 51 /* Level 2 halted */
#define EBADE 52 /* Invalid exchange */
#define EBADR 53 /* Invalid request descriptor */
#define EXFULL 54 /* Exchange full */
#define ENOANO 55 /* No anode */
#define EBADRQC 56 /* Invalid request code */
#define EBADSLT 57 /* Invalid slot */
#define EDEADLOCK EDEADLK
#define EBFONT 59 /* Bad font file format */
#define ENOSTR 60 /* Device not a stream */
#define ENODATA 61 /* No data available */
#define ETIME 62 /* Timer expired */
#define ENOSR 63 /* Out of streams resources */
```

```
#define ENONET 64 /* Machine is not on the network */
#define ENOPKG 65 /* Package not installed */
#define EREMOTE 66 /* Object is remote */
#define ENOLINK 67 /* Link has been severed */
#define EADV 68 /* Advertise error */
#define ESRMNT 69 /* Srmount error */
#define ECOMM 70 /* Communication error on send */
#define EPROTO 71 /* Protocol error */
#define EMULTIHOP 72 /* Multihop attempted */
#define EDOTDOT 73 /* RFS specific error */
#define EBADMSG 74 /* Not a data message */
#define EOVERFLOW 75 /* Value too large for defined data type */
#define ENOTUNIQ 76 /* Name not unique on network */
#define EBADFD 77 /* File descriptor in bad state */
#define EREMCHG 78 /* Remote address changed */
#define ELIBACC 79 /* Can not access a needed shared library */
#define ELIBBAD 80 /* Accessing a corrupted shared library */
#define ELIBSCN 81 /* .lib section in a.out corrupted */
#define ELIBMAX 82 /* Attempting to link in too many shared libraries
*/
#define ELIBEXEC 83 /* Cannot exec a shared library directly */
#define EILSEQ 84 /* Illegal byte sequence */
#define ERESTART 85 /* Interrupted system call should be restarted */
#define ESTRPIPE 86 /* Streams pipe error */
#define EUSERS 87 /* Too many users */
#define ENOTSOCK 88 /* Socket operation on non-socket */
#define EDESTADDRREQ 89 /* Destination address required */
#define EMSGSIZE 90 /* Message too long */
#define EPROTOTYPE 91 /* Protocol wrong type for socket */
#define ENOPROTOOPT 92 /* Protocol not available */
#define EPROTONOSUPPORT 93 /* Protocol not supported */
#define ESOCKTNOSUPPORT 94 /* Socket type not supported */
#define EOPNOTSUPP 95 /* Operation not supported on transport endpoint
*/
#define EPFNOSUPPORT 96 /* Protocol family not supported */
#define EAFNOSUPPORT 97 /* Address family not supported by protocol */
#define EADDRINUSE 98 /* Address already in use */
#define EADDRNOTAVAIL 99 /* Cannot assign requested address */
#define ENETDOWN 100 /* Network is down */
#define ENETUNREACH 101 /* Network is unreachable */
#define ENETRESET 102 /* Network dropped connection because of reset
*/
#define ECONNABORTED 103 /* Software caused connection abort */
#define ECONNRESET 104 /* Connection reset by peer */
```

```
#define ENOBUFS 105 /* No buffer space available */
#define EISCONN 106 /* Transport endpoint is already connected */
#define ENOTCONN 107 /* Transport endpoint is not connected */
#define ESHUTDOWN 108 /* Cannot send after transport endpoint shutdown
*/
#define ETOOMANYREFS 109 /* Too many references: cannot splice */
#define ETIMEDOUT 110 /* Connection timed out */
#define ECONNREFUSED 111 /* Connection refused */
#define EHOSTDOWN 112 /* Host is down */
#define EHOSTUNREACH 113 /* No route to host */
#define EALREADY 114 /* Operation already in progress */
#define EINPROGRESS 115 /* Operation now in progress */
#define ESTALE 116 /* Stale NFS file handle */
#define EUCLEAN 117 /* Structure needs cleaning */
#define ENOTNAM 118 /* Not a XENIX named type file */
#define ENAVAIL 119 /* No XENIX semaphores available */
#define EISNAM 120 /* Is a named type file */
#define ENOKEY 126 /* Required key not available */
#define EKEYEXPIRED 127 /* Key has expired */
#define EKEYREVOKED 128 /* Key has been revoked */
#define EKEYREJECTED 129 /* Key was rejected by service */
#define EOWNERDEAD 130 /* Owner died */
#define ENOTRECOVERABLE 131 /* State not recoverable */
#define ERFKILL 132 /* Operation not possible due to RF-kill */
#define EHWPOISON 133 /* Memory page has hardware error */
```

2. 打印错误信息

- strerror

```
#include <string.h>
char *strerror(int errnum);
返回值：指向消息字符串的指针
```

- perror

```
#include <stdio.h>
void perror(const char *msg);
```

实例：显示两个出错函数的使用方法

```
#include "apue.h"
#include <errno.h>

int main(int argc, char *argv[]) {
    fprintf(stderr,"EACCES: %s\n",strerror(EACCES));
```

```
    errno = ENOENT;
    perror(argv[0]);
    exit(0);
}
```

3. 出错恢复

可将在<errno.h>中定义的各种出错分成两类：致命性的和非致命性的。对于致命性的错误，无法执行恢复动作。最多能做的是在用户屏幕上打印出一条出错消息或者将一条出错消息写入日志文件中，然后退出。对于非致命的出错，优势可以叫妥善地进行处理。大多数非致命性出错是暂时的（如资源短缺），当系统中的活动较少时，这种出错很可能不会发生。

与资源相关的非致命性出错包括：EAGAIN、ENFILE、ENOBUFS、ENOLCK、ENOSPC、EWOULDBLOCK，优势 ENOMEM 也是非致命性出错。当 EBUSY 指明共享资源正在使用时，也可将它作为非致命性出错处理。当 ENTER 中断一个慢速系统调用时，可将它作为非致命性出错处理。

对于资源相关的非致命性出错的典型恢复操作是延迟一段时间，然后重试。这种技术可应用于其他情况。例如，假设出错表明一个网络连接不再起作用，那么应用程序可以采用这种方法，在短时间延迟后，然后重新建立该连接。一些应用使用指数补偿算法，在每次迭代中等待更长时间。

最终，由应用的开发者决定在那些情况下应用程序可以从出错中恢复。如果能够采用一种合理的恢复策略，那么可以避免应用程序异常终止，进而就能改善应用程序的健壮性。

七、 用户标识

1. 用户 ID（user ID）
2. 组 ID（group ID）

实例：打印用户 ID 和组 ID

```
#include "apue.h"

int main() {
    printf("uid = %d,gid = %d\n",getuid(),getgid());
    exit(0);
}
```

3. 附属组 ID（supplementary group ID）

八、 信号

信号（signal）用于通知进程发生了某种情况。进程有以下三种处理信号的方式：

- 忽略信号。有些信号表示硬件异常，例如，除以 0 或访问进程地址空间以外的存储单元等，因为这些异常产生的后果不确定，所以不推荐使用这种处理方式。
- 按系统默认方式处理。对于除数为 0，系统默认方式是终止该进程。
- 提供一个函数，信号发生时调用该函数，这被称为捕捉该信号。通过提供自编的函数，我们就能知道什么时候产生了信号，并按期望的方式处理它。

很多情况都会产生信号。终端键盘上有两种方式产生信号，分别称为中断键（interrupt key，通常是 Delete 键或 Ctrl+C）和退出键（quit key，通常是 Ctrl+\），它们被用于中断当前运行的进程。另一种产生信号的方法是调用 kill 函数。在一个进程中调用此函数就可向另一个进程发送一个信号。当然这样做也有一些限制：当想一个进程发送信号时，我们必须是哪个进程的所有者或者是超级用户。

实例：捕捉信号并处理

```
#include "apue.h"
#include <sys/wait.h>

static void sig_int(int); /*our singal_catching function*/

int main(void) {
    char buf[MAXLINE]; /* from apue.h */
    pid_t pid;
    int status;

    if (signal(SIGINT,sig_int) == SIG_ERR)
        err_sys("signal error");


    printf("%% "); /* print prompt (print requires %% to print %)*/
    while (fgets(buf,MAXLINE,stdin) != NULL) {
        if (buf[strlen(buf) - 1] == '\n')
            buf[strlen(buf) - 1] = 0; /* replace newline with null */

        if ((pid = fork()) < 0) {
            err_sys("fork error");
        } else if (pid == 0) { /* child */
```

```
            execlp(buf,buf,(char*)0);
            err_ret("could't execute: %s",buf);
            exit(127);
        }

        /* parent */
        if ((pid = waitpid(pid,&status,0))<0)
            err_sys("waitpid error");
        printf("%% ");
    }
    exit(0);
}

void sig_int(int signo) {
    printf("interrupt\n%% ");
}
```

九、 时间值

1. 日历时间。数据类型 time_t
2. 进程时间。数据类型 clock_t

- 时钟时间，又称墙上时钟时间（wall clock time）
- 用户 CPU 时间
- 系统 CPU 时间

时钟时间(墙上时钟时间 wall clock time)：从进程从开始运行到结束，时钟走过的时间，这其中包含了进程在阻塞和等待状态的时间。

用户 CPU 时间：就是用户的进程获得了 CPU 资源以后，在用户态执行的时间。

系统 CPU 时间：用户进程获得了 CPU 资源以后，在内核态的执行时间。

进程的三种状态为阻塞、就绪、运行。

时钟时间 ＝ 阻塞时间 ＋ 就绪时间 ＋运行时间

用户 CPU 时间 ＝ 运行状态下用户空间的时间

系统 CPU 时间 ＝ 运行状态下系统空间的时间。

用户 CPU 时间+系统 CPU 时间=运行时间。

十、 系统调用和库函数

系统调用（system call），指运行在用户空间的应用程序向操作系统内核请求某些服务的调用过程。 系统调用提供了用户程序与操作系统之间的接口。一

般来说，系统调用都在内核态执行。由于系统调用不考虑平台差异性，由内核直接提供，因而移植性较差（几乎无移植性）。

库函数（library function），是由用户或组织自己开发的，具有一定功能的函数集合，一般具有较好平台移植性，通过库文件（静态库或动态库）向程序员提供功能性调用。程序员无需关心平台差异，由库来屏蔽平台差异性。

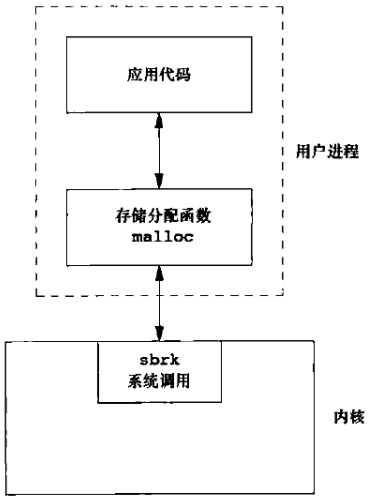

图 1-11 malloc 函数和 sbrk 系统调用

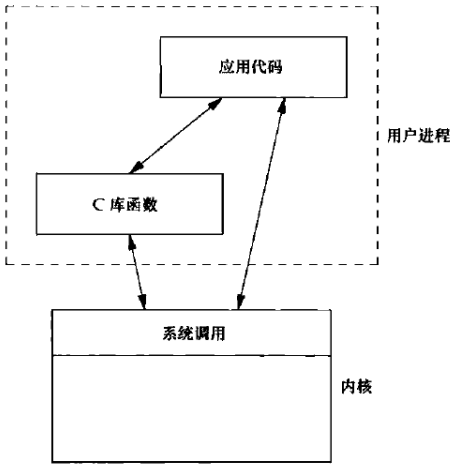

图 1-12 C 库函数和系统调用之间的差别

库函数调用和系统调用的比较：

函数库调用	系统调用
平台移植性好	依赖于内核，不保证移植性
调用函数库中的一段程序（或函数）	调用系统内核的服务
一个普通功能函数的调用	是操作系统的一个入口点
在用户空间执行	在内核空间执行
它的运行时间属于“用户时间”	它的运行时间属于“系统”时间
属于过程调用，调用开销较小	在用户空间和内核上下文环境间切换，开销较大
库函数数量较多	UNIX 中大约有 90 个系统调用，较少
典型的 C 函数库调用:printf scanf malloc	典型的系统调用：fork open write

库函数调用和系统调用的联系：

一般而言，跟内核功能与操作系统特性紧密相关的服务，由系统调用提供；具有共通特性的功能一般需要较好的平台移植性，故而由库函数提供。

库函数与系统调用在功能上是相互补充的，如进程间通信资源的管理，进程控制等功能与平台特性和内核息息相关，必须由系统调用来实现。

文件 I/O 操作等各平台都具有的共通功能一般采用库函数，也便于跨平台移植。

某些情况下，库函数与系统调用也有交集，如库函数中的 I/O 操作的内部实现依然需要调用系统的 I/O 方能实现。

十一、 小结

快速楼兰了 UNIX 系统。说明了某些以后会多次用到的基本术语，介绍了一些小的 UNIX 程序实例。