APUE

- 谷歌的Android采用Linux作为操作系统内核
- 系统调用接口和标准C库提供的许多函数

第一章　UNIX基础知识

1.1

1.2 UNIX体系结构

- 从严格意义上说，将操作系统称为内核，内核的接口被称为系统调用
- 从广义上说，操作系统包括了内核和系统实用程序、应用程序、shell以及公共函数库

1.3登录

- 登录名
- shell

1.4 文件和目录

- 文件系统
  - stat和fstat函数返回包含所有文件属性的一个信息结构
- 文件名
- 路径名
- 工作目录
  - chdir函数更改其工作目录

1.5 输入和输出

- 文件描述符
  - 当内核打开一个现有文件或创建一个新文件时，都会返回一个文件描述符，在读写文件时，可以使用这个文件描述符
- 标准输入、标准输出和标准错误
- 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,都使用文件描述符
  - 头文件<unistd.h>包含了很多UNIX系统服务的函数原型，如read和write
- 标准I/O
  - 提供了一个带缓冲的接口
  - 常量EOF和标准I/O常量stdin和stdout也在<stdio.h>中定义

1.6 程序和进程

- 程序，内核使用exec函数将程序读入内存，并执行程序
- 进程和进程ID
  - 程序的执行实例被称为进程
  - getpid得到进程ID
- 进程控制
  - 3个用于进程控制的主要函数: fork exec waitpid
  - fork对父进程返回新的子进程的进程ID,对子进程则返回0
- 线程和线程ID
  - 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，线程ID只在它所属的进程内起作用

1.7 出错处理

- 当Unix函数出错时，通常会返回一个负值，而且整型变量errno通常被设置为具有特定信息的值
- 文件<errno.h>中定义了errno以及可以赋予它们的各种常量
- 两个出错函数　　strerror perror
- 出错恢复

1.8 用户标识

- 用户ID
  - 用户ID为0的为root
  - 口令文件包含了登录名和用户ID之间的映射关系
- 组ID
  - 组文件通常是/etc/group
  - 组文件则包含了组名和组ID之间的映射关系
- getuid getgid 返回用户ID和组ID

1.9 信号

- 忽略信号
- 按系统默认方式处理
- 提供一个函数，信号发生时调用该函数
- 为了能捕捉到信号，程序需要调用signal函数

1.10 时间值

- 日历时间　time_t
- 进程时间　clock_t
- 时钟时间
- 用户CPU时间

- 系统CPU时间
- time
- cd /usr/include time -p grep _POSIX_SOURCE /.h 展示三种时间

1.11 系统调用和库函数

- 各种版本的UNIX实现都提供良好定义、数量有限、直接进入内核的入口点，这些入口点被称为系统调用
- 通用库函数可能会调用一个或多个内核的系统调用，但是它们并不是内核的入口点
- 应用程序既可以调用系统调用也可以调用库函数，很多库函数则会调用系统调用

第二章　UNIX标准及实现

2.1

2.2 UNIX标准化

- ISO C
  - ISO C头文件依赖于操作系统所配置的C编译器的版本
  - 可将ISO C库分成24个区
- IEEE POSIX
  - POSIX指可移植操作系统接口
  - POSIX.1包含了ISO C标准库函数
- Single UNIX Specification
  - POSIX.1相当于Single UNIX Specification中的基本规范部分
  - 只有遵循XSI的实现才能称为UNIX系统

2.3 UNIX系统实现

- SVR4
- 4.4BSD
- FreeBSD
- Linux
- Mac OS X
- Solaris
- 其他UNIX系统

2.4 标准和实现的关系

- 各个标准定义了任一实际系统的子集

2.5 限制

- 编译时限制，可以在头文件里定义
- 运行时限制，要求进程调用一个函数获得限制值

    - 与文件或目录无关的运行时限制(sysconf函数)
    - 与文件或目录有关的运行时限制(pathconf函数和fpathconf函数)
- ISO C限制
    - 定义的所有编译时限制都定义在头文件<limits.h>
    - 头文件<float.h>也有类似的定义
    - FOPEN_MAX,在<stdio.h>中，具体实现保证可同时打开的标准I/O流的最小个数
    - TMP_MAX
    - FILENAME_MAX
- POSIX限制
    - 定义在<limits.h>中
    - 数值限制
    - 最小值
    - 最大值运行时可以增加的值
    - 运行时不变值
    - 其他不变值
    - 路径名可变值
    - sysconf, pathconf, fpathconf使用这三个函数可以在运行时得到实际的实现值
- XSI限制
- 函数sysconf, pathconf, fpathconf
    - awk
- 不确定的运行时限制
    - 路径名
    - 最大打开文件数

2.6 选项

- 需要一种可移植的方法来判断实现是否支持一个给定的选项
- 三种处理选项的方法
    - 编译时选项定义在<unistd.h>
    - sysconf
    - pathconf
    - fpathconf

2.7 功能测试宏

- 一旦定义了_POSIX_C_SOURCE，所有POSIX.1头文件都使用此常量来排除任何实现专有的定义
- cc -D _POSIX_C_SOURCE=200809L file.c 使得C程序在任何头文件之前，定义了功能测试宏

2.8 基本系统数据类型

- 头文件<sys/types.h>中定义了某些与实现有关的数据类型

2.9 标准之间的冲突

第三章　文件I/O

3.1 引言

- open read write lseek close
- 不带缓冲指的是每个read和write都调用内核中的一个系统调用
- 在多个进程间共享资源

3.2 文件描述符

- 所有打开的文件都通过文件描述符引用
- 常量STDIN_FILENO STDOUT_FILENO STDERR_FILENO 在头文件<unistd.h>中定义

3.3 函数open和openat

- #include<fcnt1.h>
- 由open和openat函数返回的文件描述符一定是最小的未用描述符数值
- 同一进程中的所有线程共享相同的当前工作目录
- TOCTTOU错误
- 文件名和路劲截断

3.4 函数creat

- #include<fcnt1.h>
- 可调用creat函数创建一个新文件

3.5 函数close

- #include<unistd.h>
- 可调用close函数关闭一个打开文件
- 关闭一个文件还会释放该进程加在该文件上的所有记录锁
- 当一个进程终止时，内核自动关闭它所有的打开文件

3.6 函数lseek

- 每个打开文件都有一个与其相关联的当前文件偏移量，用以度量从文件开始处计算的字节数
- 可以调用lseek显式地为一个打开文件设置偏移量
- #include<unistd.h>
- 若lseek成功执行，则返回新的文件偏移量
- 如果文件描述符指向的是一个管道、FIFO或网络套接字，则lseek返回-1,并将errno设置为ESPIPE
- 某些设备也可允许负的偏移量
- 文件偏移量可以大于文件的当前长度，对该文件的下一次写将加长该文件，并在文件中构成一个空洞，空洞不要求在磁盘上占用存储区
- od命令
- od -c filename 以字符方式打印文件内容
- off-t类型，具体实现根据各自特定的平台自行选择大小合适的数据类型

3.7 函数read

- 从打开文件中读数据
- 读操作从文件的当前偏移量处开始，在成功返回之前，该偏移量将增加实际读到的字节数
- #include<unistd.h>

3.8 函数write

- 调用write函数向打开文件写数据
- 其返回值通常与参数nbytes的值相同，否则表示出错

3.9 I/O的效率

3.10 文件共享

- 在不同进程间共享打开文件
- 内核使用3种数据结构表示打开文件
  - 进程表项
  - 文件表项
    - 可能有多个文件描述符指向同一文件表项
  - v节点表项
    - 创建v节点结构的目的是对在一个计算机系统上的多文件系统类型提供支持
    - 打开该文件的每个进程都获得各自的一个文件表项，但对于一个给定的文件只有一个v节点表项

3.11 原子操作

- 追加到一个文件
  - 任何要求多于一个函数调用的操作都不是原子操作，因为在两个函数调用之间，内核有可能会临时挂起进程
- 函数pread和pwrite
  - #include<unistd.h>
  - 调用pread相当于调用lseek后调用read
- 创建一个文件
- 一般而言，原子操作指的是由多步组成的一个操作，如果该操作原子地执行，则要么执行完所有步骤，要么一步也不执行，不可能只执行所有步骤的一个子集

3.12 函数dup和dup2

- 可用来复制一个现有的文件描述符
- 这些函数返回的新文件描述符与参数fd共享同一个文件表项
- 每个文件描述符都有它自己的一套文件描述标志

3.13 函数sync, fsync, fdatasync

- 当我们向文件写入数据时，内核通常先将数据复制到缓冲区，然后排入队列，晚些时候再写入磁盘，这种方式称为延迟写
- #include<unistd.h>

- sync只是将所有修改过的块缓冲区排入写队列，然后就返回，它并不等待实际写磁盘操作结束
- fsync函数只对由文件描述符指定的一个文件起作用，并且等待写磁盘操作结束才返回
- fdatasync函数类似与fsync,但它只影响文件的数据部分

3.14 函数fcnt1

- fcnt1函数可以改变已经打开文件的属性
- #include<fcnt1.h>
- F_DUPFD　复制文件描述符fd
- F_DUPFD_CLOEXEC
- F_GETFD
- F_SETFD
    - 当前只定义了一个文件描述符标志　　FD_CLOEXEC
- F_GETFL
- F_SETFL
- F_GETOWN
- F_SETOWN
- 子句5<>temp.foo表示在文件描述符5上打开文件temp.foo以供读写
- 在UNIX系统中，通常write只是将数据排入队列，而实际的写磁盘操作则可能在以后的某个时刻进行。开启同步写标志，直至数据已写到磁盘上再返回
- 同步写和延迟写

3.15 函数ioct1

- #include<sys/ioct1.h>
- 每个设备驱动程序可以定义自己专用的一组ioct1命令，系统则为不同种类的设备提供通用的ioct1命令

3.16 /dev/fd

- /dev/fd/n
- 打开/dev/fd/n文件等效于复制描述符n , 但在Linux系统中，Linux实现使用指向实际文件的符号链接，不一样

第四章　文件和目录

4.1

4.2 函数stat fstat fstatat lstat

- #include<sys/stat.h>
- struct stat

4.3 文件类型

- 普通文件
- 目录文件，只有内核可以直接写目录文件
- 块设备文件　这种类型的文件提供对设备(如磁盘)带缓冲的访问，每次访问以固定长度为单位进行
- 字符特殊文件，　这种类型的文件提供对设备不带缓冲的访问，每次访问长度可变
- FIFO　用于进程间通信，也称为命令管道
- 套接字　用于进程间的网络通信
- 符号链接　这种类型的文件指向另一个文件
- 文件类型信息包含在stat结构的st_mode成员中
- S_ISxxx宏

4.4 设置用户ID和设置组ID

- ID
    - 实际用户ID
    - 实际组ID
    - 有效用户ID
    - 有效组ID
    - 附属组ID
    - 保存的设置用户ID 保存在文件的st_mode值中
    - 保存的设置组ID　　保存在文件的st_mode值中

4.5 文件访问权限

- st_mode值也包含了对文件的访问权限位，9个访问权限位
- 如果当前目录是/usr/include,那么为了打开文件stdio.h, 需要对当前目录有执行权限，对该目录的执行权限使我们可搜索该目录
- 用exec函数执行任何一个文件，该文件需具有执行权限，该文件还必须是一个普通文件
- 两个所有者ID是文件的性质，而两个有效ID和附属组ID则是进程的性质

4.6 新文件和目录的所有权

- 新文件的用户ID设置为进程的有效用户ID
- 新文件的组ID

4.7 函数access和faccessat

- 按实际用户ID和实际组ID进行访问权限测试的
- #include<unistd.h>
- 当用open函数打开一个文件时，内核以进程的有效用户ID和有效组ID为基础执行其访问权限测试

4.8 函数umask

- umask函数为进程设置文件模式创建屏蔽字，并返回之前的值
- 在文件模式创建屏蔽字中为1的位，在文件mode中的相应位一定会被关闭
- 更改进程的文件模式创建屏蔽字并不影响父进程的屏蔽字

4.9 函数chmod fchmod 和fchmodat

- 更改现有文件的访问权限

4.10 粘着位

- 如果可执行程序文件的这一位被设置了。那么当程序第一次被执行，在其终止时，程序正文部分的一个副本仍被保存在交换区
- 如果对一个目录设置了粘着位，只有对该目录具有写权限的用户并满足下列条件之一，才能删除或重命名该目录下的文件

4.11 函数chown fchown fchownat lchown

- 可用于改变文件的用户ID和组ID
- #include<unistd.h>

4.12 文件长度

- stat结构成员st_size表示以字节为单位的文件的长度
- 文件中的空洞
    - wc -c 计算文件中的字节数
    - 文件系统使用了若干块以存放指向实际数据块的各个指针

4.13 文件截断

- truncate
- ftruncate 将一个现有文件长度截断为length

4.14 文件系统

- 硬链接
    - 每个i结点都有一个链接计数，其值是指向该i结点的目录项数
- 符号链接
    - 符号链接文件的实际内容(在数据块中)包含了该符号链接所指向的文件的名字
- i节点包含了文件有关的所有信息，只有文件名和i节点编号存放在目录项中
- 为一个文件重命名时，只需构造一个指向现有i节点的新目录项
- 链接计数大于或等于3表明至少有3个目录项指向它

4.15 函数link linkat unlink unlinkat remove

- 任何一个文件可以有多个目录项指向其i节点，创建一个指向现有文件的链接的方法是使用link函数或linkat函数
- 创建新目录项和增加链接计数应当是一个原子操作
- 很多文件系统实现不允许对于目录的硬链接
- 为了删除一个现有的目录项，可以调用unlink函数
- 关闭一个文件时，内核首先检查打开该文件的进程个数，如果这个计数达到0,内核再去检查其链接计数，如果计数也是０，那么删除该文件的内容
- remove可以解除对一个文件或目录的链接

4.16 函数rename和renameat

- 文件或目录可以用rename函数或者renameat函数进行重命名
- #include<stdio.h>

4.17 符号链接

- 硬链接直接指向文件的i节点
- 对符号链接以及它指向何种对象并无任何文件系统限制
- 使用符号链接可能在文件系统中引入循环
- 各个函数是否处理符号链接?

4.18 创建和读取符号链接

- symlink symlinkat函数创建一个符号链接
- 不需要位于同一文件系统中
- readlink readlinkat 打开该链接本身，并读取该链接中的名字

4.19 文件的时间

- st_atim 文件数据的最后访问时间 ls -u
- st_mtim 文件数据的最后修改时间
- st_ctim i节点状态的最后更改时间 ls -c
- i节点中的所有信息都是与文件的实际内容分开存放的
- 目录是包含目录项(文件名和相关的i节点编号)的文件

4.20 函数futimens utimensat utimes

- 访问和修改时间的更改

4.21 函数mkdir mkdirat rmdir

- 用mkdir mkdirat函数创建目录　　.和..目录自动创建
- 用rmdir函数删除目录，可以删除一个空目录，只包含.和..这两项的目录

4.22 读目录

- opendir fdopendir函数
- readdir
- rewinddir
- closedir
- telldir和seekdir
- #include<dirent.h>

4.23 函数chdir fchdir getcwd

- 每个进程都有一个当前工作目录
- 进程调用chdir或fchdir函数可以更改当前工作目录
- getcwd

4.24 设备特殊文件

- 每个文件系统所在的存储设备都由其主、次设备号表示，主设备号标识设备驱动程序，次设备号标识特定的子设备，同一磁盘驱动器上的各文件系统通常具有相同的主设备号，但是次设备号却不同
- st_dev值是文件系统的设备号
- st_rdev包含实际设备的设备号

第五章　标准I/O库

5.2 流和FILE对象

- 对于标准I/O库，它们的操作都是围绕流进行的
- 流的定向决定了所读、所写的字符是单字节还是多字节的
- fwide函数设置流的定向
- 标准I/O函数返回一个指向FILE对象的指针，该对象通常是一个结构

5.3 标准输入、标准输出和标准错误

- 对一个进程预定义了3个流，这3个标准I/O流通过预定义指针stdin stdout stderr加以引用

5.4 缓冲

- 标准I/O库提供缓冲的目的是尽可能减少使用read和write调用的次数
- 全缓冲，在填满标准I/O缓冲区后才进行实际I/O操作
- 行缓冲，遇到换行字符时，执行I/O操作
- 不带缓冲，标准错误流通常是不带缓冲的
- setbuf 函数打开或关闭缓冲机制
- setvbuf
- fflush 使该流所有未写的数据都被传送到内核

5.5 打开流

- fopen函数打开一个指定的文件
- freopen函数在一个指定的流上打开一个指定的文件
- fdopen函数取一个已有的文件描述符，并使一个标准的I/O流与该描述符相结合

5.6 读和写流

- 输入函数
    - getc
    - fgetc
    - getchar
    - 可用于一次读一个字符
- 出错标志和文件结束标志
    - ferror
    - feof
    - 为每个流在FILE对象中维护了两个标志
    - clearerr可以清除这两个标志
- ungetc将字符再压送回流中

- 输出函数
    - putc
    - fputc
    - putchar

5.7 每次一行I/O

- fgets
- gets
- fputs
- puts

5.8 标准I/O的效率

- exit函数将会冲洗任何未写的数据，然后关闭所有打开的流
- 系统调用与普通的函数调用相比需要花费更多的时间

5.9 二进制I/O

- fread
- fwrite
- 使用二进制I/O的基本问题是: 它只能用于读在同一个系统上已写的数据

5.10 定位流

- ftell
- fseek
- ftello
- fseeko
- fgetpos
- fsetpos

5.11 格式化I/O

- 格式化输出
    - printf
    - fprintf
    - dprintf
    - sprintf
    - snprintf
    - 转换说明以百分号%开始
- 格式化输入
    - scanf
    - fscanf
    - sscanf

5.12 实现细节

- 在UNIX中，标准I/O库最终都要调用第3章说明的I/O例程，每个标准I/O流都有一个与其关联的文件描述符，可以对一个流调用fileno函数获得其描述符
- 标准错误是不带缓冲的，普通文件按系统默认是全缓冲的

5.13 临时文件

- tmpnam 产生一个有效路径名字符串
- tmpfile　创建一个临时二进制文件，在关闭文件或程序结束时将自动删除这种文件
- mkdtemp 和mkstemp 创建目录和文件，　mkstemp创建的临时文件并不会自动删除，必须自己对它解除链接
- 把指针自身驻留在栈上，但指向的常量字符串存放在可执行文件的只读段，当试图修改字符串时，会出现段错误(segment fault)

5.14 内存流

- 标准I/O库把数据缓存在内存中
- fmemopen 用于内存流的创建
- open_memstream 函数创建的流是面向字节的
- open_wmemstream函数创建的流是面向宽字节的
- 避免了缓冲区溢出，内存流非常适用于创建字符串
- 内存流只访问主存，不访问磁盘上的文件

第六章　系统数据文件和信息

6.2 口令文件

- <pwd.h>中定义的passwd结构
- 口令文件 s /etc/passwd ,而且是一个ASCII文件
- root用户ID是0, 加密口令字段包含一个占位符
- finger命令　finger -p leiwang　　The finger displays information about the system users.
- getpwuid
- getpwnam
- passwd结构通常是函数内部的静态变量
- getpwent 返回口令文件中的下一个记录项
- setpwent
- endpwent

第七章　进程环境

7.2 main函数

- main函数的原型　int main(int argc, char* argv[]);
- 在调用main函数前先调用一个特殊的启动例程，启动例程从内核取得命令行参数和环境变量值

7.3 进程终止

- 8种方式使进程终止
    - 从main返回

    - 调用exit
    - 调用_exit或_Exit
    - 最后一个线程从其启动例程返回
    - 从最后一个线程调用pthread_exit
    - 调用abort
    - 接到一个信号
    - 最后一个线程对取消请求做出响应
- 退出函数
    - _exit 和_Exit立即进入内核
    - exit先执行一些清理处理，
    - 然后返回内核
    - 终止状态
    - m**ain**函数返回一个整型值与用该值调用**exit**是等价的
- 函数atexit
    - 一个进程可以登记多至32个函数，这些函数将由exit自动调用，终止处理程序
    - exit首先调用各终止处理程序，然后关闭所有打开流
    - 内核使程序执行的唯一方法是调用一个exec函数，进程自愿终止的唯一方法是显式或隐式地(通过调用exit)调用_exit或_Exit。进程也可以非自愿地由一个信号使其终止

7.4 命令行参数

7.5 环境表

- 每个程序都接收到一张环境表
- 环境表是一个字符指针数组，全局变量environ包含了该指针数组的地址
- extern char **environ

7.6 C程序的存储空间布局

- 正文段，由CPU执行的机器指令部分
- 初始化数据段
- 未初始化数据段
- 栈
- 堆
- a.out还有若干其他类型的段，包含符号表的段、包含调试信息的段以及包含动态共享库链接表的段等，这些部分并不装载到进程执行的程序映像中
- 未初始化数据段的内容并不存放在磁盘程序文件中，内核在程序开始运行前将它们都设置为0,需要存放在磁盘程序文件中的段只有正文段和初始化数据段
- size命令报告正文段、数据段和bss段的长度　　size a.out

7.7 共享库

- 共享库使得可执行文件中不再需要包含公用的库函数，而只需要在所有进程都可引用的存储区中保存这种库例程的一个副本
- 共享库的另一个优点是可以用库函数的新版本代替老版本而无需对使用该库的程序重新链接编译
- gcc -static hello.c 不使用共享库的方式
- gcc hello.c 使用共享库的方式

7.8 存储空间分配

- 3个用于存储空间动态分配的函数
    - malloc,分配指定字节数的存储区，此存储区的初始值不确定
    - calloc 为指定数量指定长度的对象分配存储空间，该空间中的每一位都初始化为0
    - realloc 增加或减少以前分配区的长度，当增加长度时，可能需要将以前分配区的内容移到另一个足够大的区域，新曾区域初始值不确定
    - 这三个函数返回的指针一定是适当对齐的
    - free
- 替代的存储空间分配程序

7.9 环境变量

- getenv 取环境变量值
- putenv　可自由地将传递给它的字符串直接放到环境中
- setenv 必须分配存储空间
- unsetenv
- 环境表和环境字符串通常存放在进程存储空间的顶部(栈之上)

7.11 函数setjmp和longjmp

- 对于处理发生在很深层次嵌套函数调用中的出错情况是非常有用的，在栈上跳过若干调用帧，返回到当前函数调用路径上的某一个函数中
- C语言中，goto语言是不能跨越函数的
- 在希望返回的位置调用setjmp,参数env的类型是一个特殊类型jmp_buf
- 自动变量、寄存器变量和易失变量
    - 定义为volatile属性的自动变量，其值不回滚
    - 全局变量或静态变量的值在执行longjmp时保存不变
    - gcc testjmp.c 不进行任何优化的编译　所有这5个变量都存放在存储器中
    - gcc -O testjmp.c　进行全部优化的编译， autoval和regival都存放在寄存器中，volatile变量仍存放在存储器中
    - 全局变量、静态变量和易失变量不受优化的影响，在longjmp之后，它们的值是最近的值
    - 存放在存储器中的变量将具有longjmp时的值，而在CPU和浮点寄存器中的变量则恢复为调用setjmp时的值

7.11 函数getrlimit和setrlimit

- 资源限制的查询和更改
- 资源限制影响到调用进程并由其子进程继承
- 字符串创建算符(#) #define doit(name) pr_limits(#name, name)
- warning: ISO C++ forbids converting a string constant to ‘char*’ [-Wwrite-strings] #define doit(name) pr_limits(#name, name) g++编译时
- gcc编译时不会

第八章　进程控制

8.2 进程标识

- ID为0的进程通常是调度进程，称为交换进程，是内核的一部分
- 进程ID1通常是init进程，在自举过程结束时由内核调用，此进程负责在自举内核后启动一个Unix系统，init通常读取与系统有关的初始化文件，并将系统引导到一个状态。init进程决不会终止，它是一个普通的用户进程，但它是以超级用户特权运行
- 每个Unix系统实现都有它自己的一套供操作系统服务的内核进程
- getpid
- getppid
- getuid
- geteuid
- getgid
- getegid

8.3 函数fork

- 一个现有的进程可以调用fork函数创建一个新进程。子进程返回0,父进程返回子进程ID
- 由fork创建的新进程被称为子进程，fork函数被调用一次，但返回两次，两次返回的区别是子进程的返回值是0,而父进程的返回值是新建子进程的进程ID
- 子进程是父进程的副本，子进程获得父进程数据空间，堆和栈的副本，这是子进程所拥有的副本，父进程和子进程并不共享这些存储空间，共享正文段
- 写时复制技术(COW)
- 在fork之后是父进程还是子进程先执行是不确定的，取决于内核所使用的调度算法。如果要求父进程和子进程互相同步，则要求某种形式的进程间通信
- strlen计算不包含终止null字节的字符串长度，而sizeof则计算包括终止null字节的缓冲区长度
- 标准输出缓冲区可以由换行符冲洗
- 文件共享
- fork的特性是父进程的所有打开文件描述符都被复制到子进程中，父进程和子进程共享同一个文件偏移量
- 父进程和子进程的共同与不同点

8.4 函数vfork

- vfork函数的调用序列和返回值与fork相同，但两者的语义不同
- 创建一个子进程，但是它不将父进程的地址空间完全复制到子进程中，因为子进程会立即调用exec或exit
- vfork保证子进程先运行，在它调用exec或exit之后父进程才可能被调度运行

8.5 函数exit

- 不管进程如何终止，最后都会执行内核中的同一段代码，这段代码为相应进程关闭所有打开文件描述符，释放它所使用的存储器
- 对于父进程已经终止的所有进程，它们的父进程都改变为init进程，称这些进程由init进程收养
- 内核为每个终止子进程保存了一定量的信息
- 一个已经终止、但是其父进程尚未对其进行善后处理(获取终止进程的有关信息，释放它所占用的资源)的进程被称为僵死进程
- ps将僵死进程的状态打印为Z
- init被编写成无论何时只要有一个子进程终止，init就会调用一个wait函数取得其终止状态，防止了在系统中塞满僵死进程

8.6 函数wait和waitpid

- 当一个进程正常或异常终止时，内核就向其父进程发送SIGCHLD信号
- 调用wait或waitpid的进程
  - 如果其所有子进程都还在运行，则阻塞
  - 如果一个子进程已终止，正等待父进程获取其终止状态，则取得该子进程的终止状态立即返回
  - 如果它没有任何子进程，则立即出错返回
- wait和waitpid所返回的终止状态的宏
- waitpid等待一个特定进程的函数，wait返回任一终止子进程的状态
- waitpid提供了一个wait的非阻塞版本
- 在fork之后，父进程和子进程都可继续执行，并且我们无法预知哪一个会先执行

8.7 函数waitid

- 类似于waitpid

8.8 函数wait3和wait4

- 允许内核返回由终止进程及其所有子进程使用的资源概况

8.9 竞争条件

- 如果一个进程希望等待一个子进程终止，则它必须调用wait函数中的一个，如果一个进程要等待其父进程终止，则可使用下列形式的循环

```
while(getppid()!=1)
    sleep(1);
```

- 为了避免竞争条件和轮询，在多个进程之间需要有某种形式的信号发送和接收的方法，在Unix中可以使用信号机制，各种形式的进程间通信也可以使用
- 在程序中将标准输出设置为不带缓冲的，于是每个字符输出都需要调用一次write

8.10 函数exec

- 当进程调用一种exec函数时，该进程执行的程序完全替换为新程序，而新程序从main函数开始执行，exec只是用磁盘上的一个新程序替换了当前进程的正文段、数据段、堆段和栈段
- 7个不同的exec函数
  - execl
  - execv
  - execle
  - execve
  - execlp
  - execvp
  - fexecve

- 三个区别
- 字母p表示该函数取filename作为参数，并且用PATH环境变量寻找可执行文件，字母l表示该函数取一个参数表，v表示取一个argv[]矢量，字母e表示取envp[]数组，而不使用当前环境
- 执行exec后，新程序从调用进程继承了一些属性
- 只有execve是内核的系统调用，另外6个只是库函数，它们最终都要调用该系统调用

8.11 更改用户ID和更改组ID

- 最小特权模型
- setuid
- setgid
- setuid
- setgid

8.12 解释器文件

- #! pathname [option-argument]
- 最常见的解释器文件
- /bin/sh
- 对于这种文件的识别是由内核作为exec系统调用处理的一部分完成的，内核使调用exec函数的进程实际执行的并不是该解释器文件，而是该解释器文件第一行中pathname所指定的文件

8.13 函数system

- system在其实现中调用了fork exec waitpid
- int system(const char* cmdstring)
- execl("/bin/sh","sh","-c",cmdstring,(char*)0);
- shell的-c选项告诉shell程序取下一个命令行参数作为命令输入
- 调用_exit而不是exit, 这是为了防止任一标准I/O缓冲(这些缓冲会在fork中由父进程复制到子进程)在子进程被冲洗
- 如　system("date")
- 使用system而不是直接使用fork和exec的优点是：system进行了所需的各种出错处理以及各种信号处理
- 设置用户ID程序
- system调用shell对命令字符串进行语法分析

8.14 进程会计

- Unix系统提供了一个选项以进行进程会计处理，启用该处理后，每当进程结束时内核就写一个会计记录
- 会计记录所需的各个数据都由内核保存在进程表中
- accton命令

8.15 用户标识

- getlogin获取登录名

8.16 进程调度

- 进程可以通过调整nice值选择以更低优先级运行，只有特权进程允许提高调度权限
- nice值越小，优先级越高
- NZERO是系统默认的nice值
- nice函数，可以获取或更改nice值
- getpriority函数获取进程的nice值
- setpriority函数可用于为进程、进程组和属于特定用户ID的所有进程设置优先级

8.17 进程时间

- 墙上时钟时间、用户CPU时间、系统CPU时间
- times函数　返回墙上时钟时间作为其函数值

第九章　进程关系

9.2 终端登录

- BSD终端登录
  - 当系统自举时，内核创建进程ID为1的进程，init进程使系统进入多用户模式，init读取文件/etc/ttys,对每一个允许登录的终端设备，init调用一次fork,生成的子进程则exec getty程序，之后再exec login程序
  - 调用登录shell exec("/bin/sh","-sh",(char*)0);
  - 登录shell读取启动文件(如.bash_profile)
- Linux终端登录
  - 类似于BSD
  - 使用存放在/etc/init目录的*.conf命名的配置文件，例如，运行/dev/tty1上的getty需要的说明可能放在/etc/init/tty1.conf文件中
- Solaris登录

9.3 网络登录

- 伪终端
- BSD网络登录
  - init调用一个shell,使其执行shell脚本/etc/rc,由此shell脚本启动一个守护进程inetd, 一旦此shell脚本终止，inetd的父进程就变成init,inetd等待TCP/IP连接请求，当一个连接到达时，执行一次fork,子进程exec适当的程序
  - telnetd进程打开一个伪终端设备，并用fork分成两个进程，父进程处理通过网络进行的通信，子进程执行login程序

9.4 进程组

- 每个进程除了有一个进程ID,还有一个进程组，其中的进程是在同一作业中结合起来的
- getpgrp 返回调用进程的进程组ID
- getpgid(0)　也返回进程的进程组ID
- 每个进程组有一个组长进程，组长进程的进程组ID等于其进程ID
- setpgid

9.5 会话

- 会话是一个或多个进程组的集合
- 通常是由shell的管道将几个进程编成一组的
  - 例如　proc1 | proc2
- 进程调用setsid 建立一个新会话，会话首进程总是一个进程组的组长进程，getsid函数返回会话首进程的进程组ID

9.6 控制终端

- 一个会话可以有一个控制终端，通常是终端设备或伪终端设备
- 建立与控制终端连接的会话首进程被称为控制进程
- 如果一个会话有一个控制终端，则它有一个前台进程组，其他进程组为后台进程组
- 键入终端的中断键，会将中断信号发送至前台进程组的所有进程
- open文件/dev/tty ,在内核中，此特殊文件是控制终端的同语义，如果程序没有控制终端，则open会失败

9.7 函数tcgetpgrp tcsetpgrp tcgetsid

- tcgetpgrp返回前台进程组ID,它与在fd上打开的终端相关联
- 进程可以调用tcsetpgrp将前台进程组ID设置为pgrpid, fd必须引用该会话的控制终端
- tcgetsid获得会话首进程的进程组ID

9.8 作业控制

- 一个作业只是几个进程的集合，通常是一个进程管道
  - 例如　vi main.c 在前台启动了只有一个进程组成的作业
  - pr*.c | lpr & 在后台启动了一个作业
- 只有前台作业接收终端输入，如果后台作业试图读终端，终端驱动程序检测到这种情况，向后台作业发送一个特定的信号SIGTTIN

9.9 shell执行程序

- ps -o pid,ppid,pgid,sid,tpgid,comm TPGID指示前台进程组
- 注意: 使用的shell不同，创建各个进程的顺序也可能不同

9.10 孤儿进程组

- 孤儿进程组: 该组中每个成员的父进程要么是该组的一个成员，要么不是该组所属会话的成员

9.11 FreeBSD实现

- session结构
- proc结构
- 控制终端vnode结构
- 控制终端tty结构
- pgrp结构
- vnode结构，在打开控制终端设备时分配此结构，进程对/dev/tty的所有访问都通过vnode结构

第10章　信号

- 信号是软件中断，提供了一种处理异步事件的方法

10.2 信号概念

- 信号名都被定义为正整数常量
- 产生信号
    - 用户按终端键时，产生的信号
    - 硬件异常产生的信号
    - kill
    - 软件条件产生的信号
- 信号的处理
    - 忽略此信号
    - 捕捉信号
    - 执行默认动作
- "终止+core"表示在进程当前工作目录的core文件中复制了该进程的内存映像，大多数UNIX系统调试程序都使用core文件检查进程终止时的状态
    - SIGABRT, 调用abort函数产生
    - SIGALRM, 当用alarm函数设置的定时器超时时产生
    - SIGBUS
    - SIGCANCEL
    - SIGCHLD
    - SIGCONT
    - SIGEMT
    - SIGFPE
    - SIGFREEZE
    - SIGHUP
    - SIGILL
    - SIGINFO
    - SIGINT
    - SIGIOSIGIOT
    - SIGKILL
    - SIGLOST
    - SIGLWP
    - SIGPIPE
    - SIGPWR
    - SIGQUIT ctrl+\ 中断进程，并且产生core文件
    - SIGSEGV 指示进程进行了一次无效的内存引用
    - SIGSTOP
    - SIGSYS
    - SIGTERM 由kill命令发送的系统默认终止信号
    - SIGTSTP 交互停止信号，Ctrl+Z, 终端驱动程序产生此信号
    - SIGTTIN
    - SIGTTOU

    - SIGURG
    - SIGUSR1
    - SIGUSR2
    - SIGWINCH
    - SIGXCPU
    - SIGXFSZ

10.3 函数signal

- Unix系统信号机制最简单的接口是signal函数，但是signal的语义与实现有关，所以最好使用sigaction函数代替signal函数
- void(*signal(int signo, void(*func)(int)))(int);　　成功，返回以前的信号处理配置，出错，返回SIG_ERR
    - signo是信号名
    - func的值是常量SIG_IGN SIG_DFL 或接到此信号后要调用的函数的地址
    - typedef void Sigfunc(int);
    - Sigfunc* signal(int, Sigfunc*);
- kill命令和kill函数只是将一个信号发送给一个进程或进程组
    - 用户定义的信号
    - kill -USR1 8249 发送SIGUSR1
    - kill -USR2 8249 发送SIGUSR2
    - kill 8249 向进程发送SIGTERM
- 程序启动
    - 当执行一个程序时，所有信号的状态都是系统默认或忽略
    - exec函数将原先设置为要捕捉的信号都更改为默认动作
- 进程创建
    - 当一个进程调用fork时，其子进程继承父进程的信号处理方式，因为子进程复制了父进程的内存映像

10.4 不可靠的信号

- 信号可能会丢失，但进程可能不知道
- 进程对信号的控制能力很差

10.5 中断的系统调用

- 被中断的系统调用，返回出错，errno设置为EINTR
- 低速系统调用
    - 可能会使系统永远阻塞的一类系统调用
- 其他系统调用
- 对于read, write系统调用，POSIX标准处理为部分成功返回
- POSIX要求只有中断信号的SA_RESTART标志有效时，实现才重启系统调用

10.6 可重入函数

- 信号处理程序中调用的函数可能影响主程序中调用的函数

- 在信号处理程序中保证调用安全的函数，这些函数是可重入的并被称为是异步信号安全的，在信号处理操作期间，它会阻塞任何会引起不一致的信号发送
- 不可重入的
  - 使用静态数据结构
  - 调用malloc或free
  - 是标准I/O函数(很多以不可重入的方式使用全局数据结构)，例如信号处理程序中使用printf函数，可能中断主程序中的printf函数调用
- 由于每个线程只有一个errno变量，信号处理程序应当在调用前保存errno,在调用后恢复errno
- 例: 在主程序中调用getwnam,在信号处理程序中也调用getwnam, malloc和free调用两次，维护的数据结构出现了损坏，程序因产生SIGSEGV信号而终止
- 在信号处理程序中调用一个非可重入函数，其结果是不可预知的

10.7 SIGCLD语义

- System V的一个信号名

10.8 可靠信号术语和语义

- 在信号产生和递送之间的时间间隔内，称信号是未决的
- 进程可以选用阻塞信号递送
- 每个进程都有一个信号屏蔽字，它规定了当前要阻塞递送到该进程的信号集

10.9 函数kill和raise

- kill函数将信号发送给进程或进程组
- raise函数则允许进程向自身发送信号
- 系统进程集包括内核进程和init

10.10 函数alarm和pause

- 使用alarm函数可以设置一个定时器，在将来的某个时刻该定时器会超时，产生SIGALRM信号
- 每个进程只能有一个闹钟时间
- pause函数使调用进程挂起直至捕捉到一个信号
- 只有执行了一个信号处理程序并从其返回时，pause才返回
- alarm和pasue之间的竞争条件问题
- 如果SIGALRM中断了某个其他信号处理程序，则调用longjmp会提早终止该信号处理程序
- 除了用来实现sleep函数外，alarm函数常用于对可能阻塞的操作设置时间上限值
  - 注意：如果系统调用是自动重启动的，则当从SIGALRM信号处理程序返回时，read并不被中断
  - 使用longjmp实现，无需担心一个慢速的系统调用是否被中断
  - 可能会有与其他信号处理程序交互的问题
- 如果要对I/O操作设置时间限制，另一种选择是使用select或poll函数

10.11 信号集

- 数据类型sigset_t以包含一个信号集
- sigemptyset 初始化由set指向的信号集，清除其中的信号

- sigfillset　初始化由set指向的信号集，使其包括所有信号
- sigaddset　将一个信号添加到已有的信号集中
- sigdelset　从信号集中删除一个信号
- sigismember
- c语言的逗号运算符，将逗号运算符后的值作为表达式的值返回
- 如果实现的信号数目少于一个整型量所包含的位数，则可用一位代表一个信号的方法实现信号集

10.12 函数sigprocmask

- 一个进程的信号屏蔽字规定了当前阻塞而不能递送给该进程的信号集
- sigprocmask可以检测或更改进程的信号屏蔽字
- 仅为单线程进程定义的

10.13 函数sigpending

- 返回一个信号集，对于调用进程而言，其中的各信号是阻塞不能递送的

```
#include "apue.h"

static void sig_quit(int);

int
main(void)
{
    sigset_t    newmask, oldmask, pendmask;

    if (signal(SIGQUIT, sig_quit) == SIG_ERR)
        err_sys("can't catch SIGQUIT");

    /*
     * Block SIGQUIT and save current signal mask.
     */
    sigemptyset(&newmask);
    sigaddset(&newmask, SIGQUIT);
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        err_sys("SIG_BLOCK error");

    sleep(5);   /* SIGQUIT here will remain pending */

    if (sigpending(&pendmask) < 0)
        err_sys("sigpending error");
    if (sigismember(&pendmask, SIGQUIT))
        printf("\nSIGQUIT pending\n");

    /*
     * Restore signal mask which unblocks SIGQUIT.
     */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("SIG_SETMASK error");
    printf("SIGQUIT unblocked\n");

    sleep(5);   /* SIGQUIT here will terminate with core file */
    exit(0);
```

```
}

static void
sig_quit(int signo)
{
    printf("caught SIGQUIT\n");
    if (signal(SIGQUIT, SIG_DFL) == SIG_ERR)
        err_sys("can't reset SIGQUIT");
}
```

10.14 函数sigaction

- 检查或修改与指定信号相关联的处理动作
- struct sigaction结构
- 如果在某种信号被阻塞时，它发生了5次，对这种信号解除阻塞后，那么对这种信号处理函数通常只会被调用一次
- 一旦对给定的信号设置了一个动作，那么在调用sigaction显式地改变它之前，该设置就一直有效
- sa_flags字段
- 用sigaction实现signal函数

```
if(signo=SIGALRM){
#ifdef SA_INTERRUPT
    act,sa_flags | = SA_INTERRUPT;
#endif
} else{
    act.sa_flags | = SA_RESTART;
}
//对除SIGALRM以外的所有信号，尝试设置SA_RESTART标志，于是被这些信号中断的系统调用能够自动重启，不希望重启由SIGALRM信号中断的系统调用，希望对I/O操作可以设置时间限制
```

10.15 函数sigsetjmp和siglongjmp

- POSIX.1并没有指定setjmp和longjmp对信号屏蔽字的作用，而是定义了两个新函数sigsetjmp和siglongjmp,在信号处理程序中进行非局部转移时应当使用这两个函数
- sigsetjmp增加了一个参数，如果savemask非0, 则sigsetjmp在env中保存进程的当前信号屏蔽字，调用siglongjmp时，如果带非0 savemask的sigsetjmp调用保存了env,则siglongjmp从其中恢复保存的信号屏蔽字
- 当调用一个信号处理程序时，被捕捉到的信号加到进程的当前信号屏蔽字中，当从信号处理程序返回时，恢复原来的屏蔽字，siglongjmp恢复了由sigsetjmp所保存的信号屏蔽字
- 而如果采用setjmp和longjmp，则可能不会恢复信号屏蔽字

10.16 函数sigsuspend

- 如果在解除阻塞时刻和pause之间确实发生了信号，那么就会产生问题，因为可能永远不会再见到该信号，pause永远阻塞
- 为了纠正此问题，需要在一个原子操作中先恢复信号屏蔽字，然后使进程休眠，由sigsuspend提供此功能

- 进程的信号屏蔽字设置为由sigmask指向的值，在捕捉到一个信号或发生了一个会终止该进程的信号之前，该进程被挂起。如果捕捉到一个信号而且从该信号处理程序返回，则sigsuspend返回，并且该进程的信号屏蔽字设置为调用sigsuspend之前的值。捕捉到该信号进入到该信号的信号处理程序中时，该信号被阻塞。
- 数据类型　sig_atomic_t, 这是右ISO C标准定义的变量类型，在写这种类型变量是不会被中断，这种变量不会跨越边界，可以用一条机器指令对其进行访问，总是包括ISO类型修饰符volatile，　该变量将由两个不同的控制线程访问-main函数和异步执行的信号处理程序
- 可以用信号实现父子进程的同步

10.17 函数abort

- abort函数的功能是使程序异常终止，将SIGABRT信号发送给调用进程
- 调用kill使其为调用者产生信号，如果该信号是不被阻塞的，则在kill返回前该信号就被传送给了该进程

10.18 函数system

- POSIX.1要求system忽略SIGINT和SIGQUIT, 阻塞SIGCHLD
- 当用system运行另一个程序时，不应使父子进程两者都捕捉终端产生的信号: 中断和退出，这两个信号只应发送给正在运行的程序:子进程
- 当system创建的子进程结束时，system的调用者可能错误地认为，它自己的一个子进程结束了，于是，调用者将会调用一种wait函数以获得子进程的终止状态，阻止了system函数获得子进程的终止状态，并将其作为它的返回值
- 若从shell调用ed,并键入中断字符，则它捕捉中断信号并打印问号，ed程序对退出信号的处理方式设置为忽略
- 调用sigprocmask函数解除SIGCHLD信号的调用是在调用waitpid获取子进程的终止状态之后
- 在Linux中，在system函数调用了waitpid后，SIGCHLD保持为未决，当解除了对此信号的阻塞后，他被递送到调用者
- POSIX.1说明，在SIGCHLD未决期间，如若wait或waitpid返回了子进程的状态，那么SIGCHLD信号不应该递送给该父进程

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

int
system(const char *cmdstring)   /* with appropriate signal handling */
{
    pid_t               pid;
    int                 status;
    struct sigaction    ignore, saveintr, savequit;
    sigset_t            chldmask, savemask;

    if (cmdstring == NULL)
        return(1);      /* always a command processor with UNIX */

    ignore.sa_handler = SIG_IGN;    /* ignore SIGINT and SIGQUIT */
    sigemptyset(&ignore.sa_mask);
    ignore.sa_flags = 0;
    if (sigaction(SIGINT, &ignore, &saveintr) < 0)
        return(-1);
```

```
    if (sigaction(SIGQUIT, &ignore, &savequit) < 0)
        return(-1);
    sigemptyset(&chldmask);         /* now block SIGCHLD */
    sigaddset(&chldmask, SIGCHLD);
    if (sigprocmask(SIG_BLOCK, &chldmask, &savemask) < 0)
        return(-1);

    if ((pid = fork()) < 0) {
        status = -1;    /* probably out of processes */
    } else if (pid == 0) {          /* child */
        /* restore previous signal actions & reset signal mask */
        sigaction(SIGINT, &saveintr, NULL);
        sigaction(SIGQUIT, &savequit, NULL);
        sigprocmask(SIG_SETMASK, &savemask, NULL);

        execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
        _exit(127);     /* exec error */
    } else {                        /* parent */
        while (waitpid(pid, &status, 0) < 0)
            if (errno != EINTR) {
                status = -1; /* error other than EINTR from waitpid() */
                break;
            }
    }

    /* restore previous signal actions & reset signal mask */
    if (sigaction(SIGINT, &saveintr, NULL) < 0)
        return(-1);
    if (sigaction(SIGQUIT, &savequit, NULL) < 0)
        return(-1);
    if (sigprocmask(SIG_SETMASK, &savemask, NULL) < 0)
        return(-1);

    return(status);
}
```

- system的返回值
  - system的返回值是shell的终止状态
  - 在Bourne shell中，终止状态是128加上一个信号编号
  - 直接调用fork exec wait，则终止状态与调用system是不同的

10.19 函数sleep nanosleep clock_nanosleep

- sleep, 当由于捕捉到某个信号，提前返回时，返回的是未休眠完的秒数
- 用nanosleep函数实现sleep
- nanosleep函数不涉及产生任何信号，不需要担心与其他函数的交互
- clock_nanosleep

10.20 函数sigqueue

- 增加对信号排队的支持

10.21 作业控制信号

- 交互式shell
- 由init将3个作业控制信号SIGTSTP SIGTTIN SIGTTOU设置为SIG_IGN ，这种配置由所有登录shell继承，只有作业控制shell才应将这3个信号重新设置为SIG_DFL

```
#include "apue.h"

#define BUFFSIZE    1024

static void
sig_tstp(int signo) /* signal handler for SIGTSTP */
{
    sigset_t    mask;

    /* ... move cursor to lower left corner, reset tty mode ... */

    /*
     * Unblock SIGTSTP, since it's blocked while we're handling it.
     */
    sigemptyset(&mask);
    sigaddset(&mask, SIGTSTP);
    sigprocmask(SIG_UNBLOCK, &mask, NULL);

    signal(SIGTSTP, SIG_DFL);   /* reset disposition to default */

    kill(getpid(), SIGTSTP);    /* and send the signal to ourself */

    /* we won't return from the kill until we're continued */

    signal(SIGTSTP, sig_tstp);  /* reestablish signal handler */

    /* ... reset tty mode, redraw screen ... */
}

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

    /*
     * Only catch SIGTSTP if we're running with a job-control shell.
     */
    if (signal(SIGTSTP, SIG_IGN) == SIG_DFL)
        signal(SIGTSTP, sig_tstp);

    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);
}
```

10.22 信号名和编号

- 信号名数组
- 可利用psignal函数打印与信号编号对应的字符串
- psiginfo函数
- strsignal

第11章　线程

- 一个进程中的所有线程都可以访问该进程的组成部件，如文件描述符和内存

11.2 线程概念

- 多线程的优点
  - 每个线程在进行事件处理时可以采用同步编程方式
  - 多个进程必须使用操作系统提供的复杂机制才能实现内存和文件描述符的共享，而多个线程自动地可以访问相同的存储地址空间和文件描述符
  - 多个控制线程时，相互独立的任务的处理就可以交叉进行
  - 多线程可以把程序中处理用户输入输出的部分与其他部分分开
- Unix进程可以看成只有一个控制线程
- 不管处理器个数的多少，程序都可以通过使用线程得以简化
- 线程包含: 线程ID 一组寄存器值　栈　调度优先级　策略　信号屏蔽字　errno变量　以及线程私有数据
- 一个进程的所有信息对该进程的所有线程都是共享的，包括可执行程序的代码，程序的全局内存，堆内存，栈，以及文件描述符
- 线程的测试宏是_POSIX_THREADS可以在编译时确定是否支持线程

11.3 线程标识

- 线程ID只在它所属的进程上下文中才有意义，用pthread_t的数据类型来表示，用一个结构来代表
- pthread_euqal 两个线程ID进行比较
- 线程可以通过调用pthread_self函数获得自身的线程ID

11.4 线程创建

- 在传统Unix进程模型中，每个进程只有一个控制线程，这与基于线程的模型中每个进程只包含一个线程是相同的
- 新增的线程可以通过调用pthread_create函数创建
- int pthread_create(pthread_t *restrict tidp, const pthread_attr_t* restrict attr, void*(*star_rtn)(void*), void *restrict arg);
- 新创建的线程从start_rtn函数的地址开始运行
- 线程创建不能保证哪个线程会先运行，新创建的线程可以访问进程的地址空间，并且继承调用线程的浮点环境和信号屏蔽字，但是该线程的挂起信号集会被清除

- 每个线程都提供errno的副本
- 如果新线程在主线程调用pthread_create返回之前就运行了，那么新线程看到的是未经初始化的ntid的内容，可能并不是正确的线程ID
- 主线程和新线程之间的竞争
- pthread 库不是 Linux 系统默认的库，连接时需要使用静态库 libpthread.a，所以在使用pthread_create()创建线程，以及调用 pthread_atfork()函数建立fork处理程序时，需要链接该库。

  问题解决： 在编译中要加 -lpthread参数 gcc mult-thread-tcp-server.c -o mult-thread-tcp-server -lpthread mult-thread-tcp-server.c 为你的源文件，不要忘了加上头文件#include<pthread.h>
- Linux线程ID是用无符号长整型来表示的，但是看起来像指针
- Linux 2.4中，是用单独的进程实现每个线程的
- Linux 2.6中，采用了一个称为Native POSIX线程库的新线程实现，支持单个进程中有多个线程的模型

11.5 线程终止

- 如果进程中的任意线程调用了exit _exit _Exit，那么整个进程就会终止
- 如果默认的动作是终止进程，那么发送到线程的信号就会终止整个进程
- 线程的退出方式
  - 简单地从启动例程中返回，返回值是线程的退出码
  - 可以被同一进程中的其他线程取消
  - 调用pthread_exit
- pthread_join
  - 调用线程将一直阻塞，直到指定的线程调用pthread_exit,从启动例程中返回或者被取消，rval_ptr包含相应的信息
  - 可以通过pthread_join自动地把线程置于分离状态

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

void *
thr_fn1(void *arg)
{
    printf("thread 1 returning\n");
    return((void *)1);
}

void *
thr_fn2(void *arg)
{
    printf("thread 2 exiting\n");
    pthread_exit((void *)2);
}

int
main(void)
{
    int         err;
    pthread_t   tid1, tid2;
    void        *tret;
```

```
    err = pthread_create(&tid1, NULL, thr_fn1, NULL);
    if (err != 0)
        err_exit(err, "can't create thread 1");
    err = pthread_create(&tid2, NULL, thr_fn2, NULL);
    if (err != 0)
        err_exit(err, "can't create thread 2");
    err = pthread_join(tid1, &tret);
    if (err != 0)
        err_exit(err, "can't join with thread 1");
    printf("thread 1 exit code %ld\n", (long)tret);
    err = pthread_join(tid2, &tret);
    if (err != 0)
        err_exit(err, "can't join with thread 2");
    printf("thread 2 exit code %ld\n", (long)tret);
    exit(0);
}
```

- pthread_create和pthread_exit函数的无类型指针参数可以传递的值不止一个，这个指针可以传递包含复杂信息的结构的地址
- pthread_cancel函数来请求取消同一进程中的其他线程，并不等待线程终止，仅仅提出请求
- 线程可以安排它退出时需要调用的函数，即线程清理处理程序，处理记录在栈中
- pthread_cleanup_push
- pthread_cleanup_pop 可以实现为宏，必须在与线程相同的作用域中以匹配对的形式使用
- 线程如果通过从它的启动例程中返回而终止的话，它的清理处理程序就不会被调用
- 需要把pthread_cleanup_pop和pthread_cleanup_push调用匹配起来

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

void
cleanup(void *arg)
{
    printf("cleanup: %s\n", (char *)arg);
}

void *
thr_fn1(void *arg)
{
    printf("thread 1 start\n");
    pthread_cleanup_push(cleanup, "thread 1 first handler");
    pthread_cleanup_push(cleanup, "thread 1 second handler");
    printf("thread 1 push complete\n");
    if (arg)
        return((void *)1);
    pthread_cleanup_pop(0);
    pthread_cleanup_pop(0);
    return((void *)2);
}
```

```
void *
thr_fn2(void *arg)
{
    printf("thread 2 start\n");
    pthread_cleanup_push(cleanup, "thread 2 first handler");
    pthread_cleanup_push(cleanup, "thread 2 second handler");
    printf("thread 2 push complete\n");
    if (arg)
        pthread_exit((void *)3);
    pthread_cleanup_pop(0);
    pthread_cleanup_pop(0);
    pthread_exit((void *)4);
}

int
main(void)
{
    int         err;
    pthread_t   tid1, tid2;
    void        *tret;

    err = pthread_create(&tid1, NULL, thr_fn1, NULL);  //处理程序未运行
    //  err = pthread_create(&tid1, NULL, thr_fn1, (void*)1);  //运行了
    if (err != 0)
        err_exit(err, "can't create thread 1");
    err = pthread_create(&tid2, NULL, thr_fn2, NULL);
    //err = pthread_create(&tid2, NULL, thr_fn2, (void*)1);
    if (err != 0)
        err_exit(err, "can't create thread 2");
    err = pthread_join(tid1, &tret);
    if (err != 0)
        err_exit(err, "can't join with thread 1");
    printf("thread 1 exit code %ld\n", (long)tret);
    err = pthread_join(tid2, &tret);
    if (err != 0)
        err_exit(err, "can't join with thread 2");
    printf("thread 2 exit code %ld\n", (long)tret);
    exit(0);
}
```

- 宏把某些上下文存放在栈上
- 在默认情况下，线程的终止状态会保存直到对该线程调用pthread_join, 如果线程已经被分离，线程的底层存储资源可以在线程终止时立即被收回。在线程被分离后，我们不能用pthread_join函数等待它的终止状态，因为对分离状态的线程调用pthread_join会产生未定义行为
- 可以调用pthread_detach分离线程

11.6 线程同步

- 当多个控制线程共享相同的内存时，需要确保每个线程看到一致的数据视图
- 当一个线程可以修改的变量，其他线程也可以读取或修改时，就需要对这些线程进行同步
- 线程不得不使用锁，同一时间只允许一个线程访问该变量，如果线程B希望读取该变量，它首先要获得锁。当线程A更新变量，也需要获得同样的这把锁。

- 两个或多个线程试图在同一时间修改同一变量时，也需要进行同步
- 如果增1只需要一个存储器周期，那么就没有竞争存在，如果数据总是以顺序一致出现的，就不需要额外的同步。在现代操作系统中，存储访问需要多个总线周期，处理器的总线周期通常在多个处理器上是交叉的，所以我们不能保证数据是一致的

互斥量

- 互斥量本质是一把锁，在访问共享资源前对互斥量进行加锁，访问完成后释放互斥量
- 对互斥量进行加锁以后，其他试图再次对互斥量加锁的线程都会被阻塞直到当前线程释放该互斥锁。如果释放互斥量时有一个以上的线程阻塞，那么所有该锁上的阻塞线程都会变成可运行状态，第一个变成运行的线程就可以对互斥量加锁，其他线程就会看到互斥量依然是锁着的。
- 互斥变量是用pthread_mutex_t数据类型表示的
- pthread_mutex_init函数进行初始化
- 如果动态分配互斥量(malloc)，在释放内存前需要调用pthread_mutex_destory
- 对互斥量进行加锁，需要调用pthread_mutex_lock, 如果互斥量已经上锁，调用线程将阻塞直到互斥量被解锁。
- 对互斥量进行解锁，需要调用pthread_mutex_unlock
- pthread_mutex_trylock 尝试对互斥量进行加锁，失败时返回EBUSY

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

struct foo {
    int             f_count;
    pthread_mutex_t f_lock;
    int             f_id;
    /* ... more stuff here ... */
};

struct foo *
foo_alloc(int id) /* allocate the object */
{
    struct foo *fp;

    if ((fp = malloc(sizeof(struct foo))) != NULL) {
        fp->f_count = 1;
        fp->f_id = id;
        if (pthread_mutex_init(&fp->f_lock, NULL) != 0) {
            free(fp);
            return(NULL);
        }
        /* ... continue initialization ... */
    }
    return(fp);
}

void
foo_hold(struct foo *fp) /* add a reference to the object */
{
    pthread_mutex_lock(&fp->f_lock);
```

```
    fp->f_count++;
    pthread_mutex_unlock(&fp->f_lock);
}

void
foo_rele(struct foo *fp) /* release a reference to the object */
{
    pthread_mutex_lock(&fp->f_lock);
    if (--fp->f_count == 0) { /* last reference */
        pthread_mutex_unlock(&fp->f_lock);
        pthread_mutex_destroy(&fp->f_lock);
        free(fp);
    } else {
        pthread_mutex_unlock(&fp->f_lock);
    }
}
```

避免死锁

- 如果线程试图对同一个互斥量加锁两次，那么它自身会陷入死锁状态
- 程序中使用一个以上的互斥量时，如果允许线程一直占有第一个互斥量，并且试图锁住第二个互斥量时处于阻塞状态，但是拥有第二个互斥量的线程也在试图锁住第一个互斥量，两个线程都无法向前运行，产生了死锁
- 可能出现的死锁只会发生在一个线程试图锁住另一个线程以相反的顺序锁住的互斥量
- 可以使用pthread_mutex_trylock接口避免死锁，如果已经占有某些锁而且pthread_mutex_trylock接口返回成功，那么就可以前进。如果不能获得锁，可以先释放已经占有的锁，过一段时间再重试
- 在同时需要两个互斥量时，总是让它们以相同的顺序加锁，可以避免死锁
- 如果锁的粒度太粗，就会出现很多线程阻塞等待相同的锁，这可能并不能改善并发性。如果锁的粒度太细，那么过多的锁开销使得系统性能受到影响，而且代码变得复杂

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

#define NHASH 29
#define HASH(id) (((unsigned long)id)%NHASH)

struct foo *fh[NHASH];
pthread_mutex_t hashlock = PTHREAD_MUTEX_INITIALIZER;

struct foo {
    int             f_count; /* protected by hashlock */
    pthread_mutex_t f_lock;
    int             f_id;
    struct foo     *f_next; /* protected by hashlock */
    /* ... more stuff here ... */
};

struct foo *
foo_alloc(int id) /* allocate the object */
{
    struct foo  *fp;
    int         idx;
```

```
    if ((fp = malloc(sizeof(struct foo))) != NULL) {
        fp->f_count = 1;
        fp->f_id = id;
        if (pthread_mutex_init(&fp->f_lock, NULL) != 0) {
            free(fp);
            return(NULL);
        }
        idx = HASH(id);
        pthread_mutex_lock(&hashlock);
        fp->f_next = fh[idx];
        fh[idx] = fp;
        pthread_mutex_lock(&fp->f_lock);
        pthread_mutex_unlock(&hashlock);
        /* ... continue initialization ... */
        pthread_mutex_unlock(&fp->f_lock);
    }
    return(fp);
}

void
foo_hold(struct foo *fp) /* add a reference to the object */
{
    pthread_mutex_lock(&hashlock);
    fp->f_count++;
    pthread_mutex_unlock(&hashlock);
}

struct foo *
foo_find(int id) /* find an existing object */
{
    struct foo  *fp;

    pthread_mutex_lock(&hashlock);
    for (fp = fh[HASH(id)]; fp != NULL; fp = fp->f_next) {
        if (fp->f_id == id) {
            fp->f_count++;
            break;
        }
    }
    pthread_mutex_unlock(&hashlock);
    return(fp);
}

void
foo_rele(struct foo *fp) /* release a reference to the object */
{
    struct foo  *tfp;
    int         idx;

    pthread_mutex_lock(&hashlock);
    if (--fp->f_count == 0) { /* last reference, remove from list */
        idx = HASH(fp->f_id);
```

```
        tfp = fh[idx];
        if (tfp == fp) {
            fh[idx] = fp->f_next;
        } else {
            while (tfp->f_next != fp)
                tfp = tfp->f_next;
            tfp->f_next = fp->f_next;
        }
        pthread_mutex_unlock(&hashlock);
        pthread_mutex_destroy(&fp->f_lock);
        free(fp);
    } else {
        pthread_mutex_unlock(&hashlock);
    }
}
```

函数**pthread_mutex_timelock**

- pthread_mutex_timelock函数与pthread_mutex_lock是基本等价的，但是在达到超时时间值时，pthread_mutex_timelock不会对互斥量进行加锁，而是返回错误码ETIMEDOUT
- 使用pthread_mutex_timelock避免永久阻塞
- ```
  #include "apue.h"
  #include <pthread.h>

  int
  main(void)
  {
      int err;
      struct timespec tout;
      struct tm *tmp;
      char buf[64];
      pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

      pthread_mutex_lock(&lock);
      printf("mutex is locked\n");
      clock_gettime(CLOCK_REALTIME, &tout);
      tmp = localtime(&tout.tv_sec);
      strftime(buf, sizeof(buf), "%r", tmp);
      printf("current time is %s\n", buf);
      tout.tv_sec += 10;  /* 10 seconds from now */
      /* caution: this could lead to deadlock */
      err = pthread_mutex_timedlock(&lock, &tout);
      clock_gettime(CLOCK_REALTIME, &tout);
      tmp = localtime(&tout.tv_sec);
      strftime(buf, sizeof(buf), "%r", tmp);
      printf("the time is now %s\n", buf);
      if (err == 0)
          printf("mutex locked again!\n");
      else
          printf("can't lock mutex again: %s\n", strerror(err));
      exit(0);
  }
  ```
  ```

读写锁

- 读写锁与互斥量类似，允许更高的并行性，互斥量要么是锁住状态，要么就是不加锁状态，而且一次只有一个线程可以对齐加锁
- 读写锁有三种状态
  - 读模式下加锁状态
  - 写模式下加锁状态
  - 不加锁状态
  - 一次只有一个线程可以占有写模式的读写锁，但是多个线程可以同时占有读模式的读写锁
- 当读写锁是写加锁状态时，在这个锁被解锁之前，所有试图对这个锁加锁的线程都会被阻塞。
- 当读写锁是在读加锁状态时，所有试图以读模式对它进行加锁的线程都可以得到访问权，但是任何以写模式对此锁加锁的线程都会阻塞，直到所有的线程释放它们的读锁
- 当读写锁处于读模式锁住时，一个线程试图以写模式获取锁时，读写锁通常会阻塞随后的读模式锁请求
- 读写锁也叫共享互斥锁
- pthread_rwlock_init进行初始化
- pthread_rwlock_destory释放资源
- pthread_rwlock_rdlock　读模式下锁住读写锁
- pthread_rwlock_wrlock 写模式下锁定读写锁
- pthread_rwlock_unlock解锁
- pthread_rwlock_trylock
- pthread_rwlock_trylock 可以获取锁时，返回0,否则返回错误EBUSY
- ```
  #include <stdlib.h>
  #include <pthread.h>

  struct job {
      struct job *j_next;
      struct job *j_prev;
      pthread_t   j_id;   /* tells which thread handles this job */
      /* ... more stuff here ... */
  };

  struct queue {
      struct job      *q_head;
      struct job      *q_tail;
      pthread_rwlock_t q_lock;
  };

  /*
   * Initialize a queue.
   */
  int
  queue_init(struct queue *qp)
  {
      int err;
  ```
  ```

```
    qp->q_head = NULL;
    qp->q_tail = NULL;
    err = pthread_rwlock_init(&qp->q_lock, NULL);
    if (err != 0)
        return(err);
    /* ... continue initialization ... */
    return(0);
}

/*
 * Insert a job at the head of the queue.
 */
void
job_insert(struct queue *qp, struct job *jp)
{
    pthread_rwlock_wrlock(&qp->q_lock);
    jp->j_next = qp->q_head;
    jp->j_prev = NULL;
    if (qp->q_head != NULL)
        qp->q_head->j_prev = jp;
    else
        qp->q_tail = jp;    /* list was empty */
    qp->q_head = jp;
    pthread_rwlock_unlock(&qp->q_lock);
}

/*
 * Append a job on the tail of the queue.
 */
void
job_append(struct queue *qp, struct job *jp)
{
    pthread_rwlock_wrlock(&qp->q_lock);
    jp->j_next = NULL;
    jp->j_prev = qp->q_tail;
    if (qp->q_tail != NULL)
        qp->q_tail->j_next = jp;
    else
        qp->q_head = jp;    /* list was empty */
    qp->q_tail = jp;
    pthread_rwlock_unlock(&qp->q_lock);
}

/*
 * Remove the given job from a queue.
 */
void
job_remove(struct queue *qp, struct job *jp)
{
    pthread_rwlock_wrlock(&qp->q_lock);
    if (jp == qp->q_head) {
        qp->q_head = jp->j_next;
        if (qp->q_tail == jp)
```

```
            qp->q_tail = NULL;
        else
            jp->j_next->j_prev = jp->j_prev;
    } else if (jp == qp->q_tail) {
        qp->q_tail = jp->j_prev;
        jp->j_prev->j_next = jp->j_next;
    } else {
        jp->j_prev->j_next = jp->j_next;
        jp->j_next->j_prev = jp->j_prev;
    }
    pthread_rwlock_unlock(&qp->q_lock);
}

/*
 * Find a job for the given thread ID.
 */
struct job *
job_find(struct queue *qp, pthread_t id)
{
    struct job *jp;

    if (pthread_rwlock_rdlock(&qp->q_lock) != 0)
        return(NULL);

    for (jp = qp->q_head; jp != NULL; jp = jp->j_next)
        if (pthread_equal(jp->j_id, id))
            break;

    pthread_rwlock_unlock(&qp->q_lock);
    return(jp);
}
```

带有超时的读写锁

- pthred_rwlock_timerdlock
- pthread_rwlock_timedwrloc

条件变量

- 条件变量是线程可用的另一种同步机制，条件变量给多个线程提供了一个会和的场合。条件变量与互斥量一起使用时，允许线程以无竞争方式等待特定条件的发生
- 条件本身是由互斥量保护的
- pthread_cond_t数据类型的条件变量
- 初始化　pthread_cond_init()
- pthread_cond_destory()
- pthread_cond_wait 等待条件变量为真　　互斥量对条件进行保护　调用者把锁住的互斥量传给函数，函数然后自动把调用线程放到等待条件的线程列表上，对互斥量进行解锁。pthread_cond_wait返回时，互斥量再次被锁住
- pthread_cond_timewait　如果超时到期条件还是没有出现，pthread_cond_timewait将重新获取互斥量，然后返回错误ETIMEDOUT

- 从pthread_cond_wait或pthread_cond_timewait调用成功返回时，线程需要重新计算条件，因为另一个线程可能已经在运行并改变了条件
- pthread_cond_signal函数至少能唤醒一个等待该条件的线程
- pthread_cond_broadcast函数则能唤醒等待该条件的所有线程
- ```
  #include <pthread.h>

  struct msg {
      struct msg *m_next;
      /* ... more stuff here ... */
  };

  struct msg *workq;

  pthread_cond_t qready = PTHREAD_COND_INITIALIZER;

  pthread_mutex_t qlock = PTHREAD_MUTEX_INITIALIZER;

  void
  process_msg(void)
  {
      struct msg *mp;

      for (;;) {
          pthread_mutex_lock(&qlock);
          while (workq == NULL)
              pthread_cond_wait(&qready, &qlock);
          mp = workq;
          workq = mp->m_next;
          pthread_mutex_unlock(&qlock);
          /* now process the message mp */
      }
  }

  void
  enqueue_msg(struct msg *mp)
  {
      pthread_mutex_lock(&qlock);
      mp->m_next = workq;
      workq = mp;
      pthread_mutex_unlock(&qlock);
      pthread_cond_signal(&qready);
  }
  ```

自旋锁

- 自旋锁与互斥量类似，但它不是通过休眠使进程阻塞，而是在获取锁之前一直处于忙等(自旋)阻塞状态
- 自旋锁可用于以下情况:锁被持有的时间短，而且线程并不希望在重新调度上花费太多成本
- 自旋锁通常作为底层原语用于实现其他类型的锁
- 当线程自旋锁变为可用时，CPU不能做其他的事情
- pthread_spin_init
- pthread_spin_destroy
  ```

- pthread_spin_lock
- pthread_spin_trylock
- pthread_spin_unlock

屏障

- 用户协调多个线程并行工作的同步机制，屏障允许每个线程等待，直到所有的合作线程都到达某一点，然后从该点继续执行， pthread_join就是一种屏障
- pthread_barrier_init对屏障进行初始化
- pthread_barrier_destroy进行反初始化
- 可以使用pthread_barrier_wait函数来表明，线程已经完成工作，准备等所有其他线程赶上来，调用pthread_barrier_wait的线程在屏障计数未满足条件时，会进入休眠状态，如果该线程是最后一个调用的线程，就满足了屏障计数，所有的线程都被唤醒
- 使用8个线程分解了800万个数的排序工作，每个线程用堆排序算法对100万个进行排序，然后主线程调用一个函数对这些结果进行合并
- ```
  #include "apue.h"
  #include <pthread.h>
  #include <limits.h>
  #include <sys/time.h>

  #define NTHR   8                /* number of threads */
  #define NUMNUM 8000000L         /* number of numbers to sort */
  #define TNUM   (NUMNUM/NTHR)    /* number to sort per thread */

  long nums[NUMNUM];
  long snums[NUMNUM];

  pthread_barrier_t b;

  #ifdef SOLARIS
  #define heapsort qsort
  #else
  extern int heapsort(void *, size_t, size_t,
                      int (*)(const void *, const void *));
  #endif

  /*
   * Compare two long integers (helper function for heapsort)
   */
  int
  complong(const void *arg1, const void *arg2)
  {
      long l1 = *(long *)arg1;
      long l2 = *(long *)arg2;

      if (l1 == l2)
          return 0;
      else if (l1 < l2)
          return -1;
      else
  ```
  ```

```
        return 1;
}

/*
 * Worker thread to sort a portion of the set of numbers.
 */
void *
thr_fn(void *arg)
{
    long    idx = (long)arg;

    heapsort(&nums[idx], TNUM, sizeof(long), complong);
    pthread_barrier_wait(&b);

    /*
     * Go off and perform more work ...
     */
    return((void *)0);
}

/*
 * Merge the results of the individual sorted ranges.
 */
void
merge()
{
    long    idx[NTHR];
    long    i, minidx, sidx, num;

    for (i = 0; i < NTHR; i++)
        idx[i] = i * TNUM;
    for (sidx = 0; sidx < NUMNUM; sidx++) {
        num = LONG_MAX;
        for (i = 0; i < NTHR; i++) {
            if ((idx[i] < (i+1)*TNUM) && (nums[idx[i]] < num)) {
                num = nums[idx[i]];
                minidx = i;
            }
        }
        snums[sidx] = nums[idx[minidx]];
        idx[minidx]++;
    }
}

int
main()
{
    unsigned long   i;
    struct timeval  start, end;
    long long       startusec, endusec;
    double          elapsed;
    int             err;
    pthread_t       tid;
```

```
    /*
     * Create the initial set of numbers to sort.
     */
    srandom(1);
    for (i = 0; i < NUMNUM; i++)
        nums[i] = random();

    /*
     * Create 8 threads to sort the numbers.
     */
    gettimeofday(&start, NULL);
    pthread_barrier_init(&b, NULL, NTHR+1);
    for (i = 0; i < NTHR; i++) {
        err = pthread_create(&tid, NULL, thr_fn, (void *)(i * TNUM));
        if (err != 0)
            err_exit(err, "can't create thread");
    }
    pthread_barrier_wait(&b);
    merge();
    gettimeofday(&end, NULL);

    /*
     * Print the sorted list.
     */
    startusec = start.tv_sec * 1000000 + start.tv_usec;
    endusec = end.tv_sec * 1000000 + end.tv_usec;
    elapsed = (double)(endusec - startusec) / 1000000.0;
    printf("sort took %.4f seconds\n", elapsed);
    for (i = 0; i < NUMNUM; i++)
        printf("%ld\n", snums[i]);
    exit(0);
}
```

第12章　线程控制

12.2 线程限制

- 通过使用sysconf函数查询

12.3 线程属性

- 每个对象关联的不同属性
- 可以使用pthread_attr_t结构修改线程默认属性，并把这些属性与创建的线程联系起来
- 可以使用pthread_attr_init函数初始化pthread_attr_t结构
- pthread_attr_destory
- 线程属性结构对应用程序来说是不透明的
- 线程的分离状态属性　　detachstat
    - 可以调用pthread_attr_getdetachstate函数获取当前的detachstat线程属性

- 可以使用pthread_attr_setdetachstat设置线程的detachstat属性
- ```
  #include "apue.h"
  #include <pthread.h>

  int
  makethread(void *(*fn)(void *), void *arg)
  {
      int             err;
      pthread_t       tid;
      pthread_attr_t  attr;

      err = pthread_attr_init(&attr);
      if (err != 0)
          return(err);
      err = pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
      if (err == 0)
          err = pthread_create(&tid, &attr, fn, arg);
      pthread_attr_destroy(&attr);
      return(err);
  }
  ```

- 线程栈末尾的警戒缓冲区大小 guardsize
  - 控制着线程栈末尾之后用以避免栈溢出的扩展内存的大小
- 线程栈的最低地址 stackaddr
- 线程栈的最小长度 stacksize
  - 可以使用pthread_attr_getstack和pthread_attr_setstack对线程栈属性进行管理
  - 进程中只有一个栈，大小不是问题。对于线程，同样大小的虚地址空间必须被所有的线程栈共享，如果应用程序使用了许多线程，以致这些线程栈的累计大小超过了可用的虚地址空间，就需要减少默认的线程栈大小
  - 如果线程栈的虚地址空间都用完了，那么可用使用malloc或mmap来为可替代的栈分配空间，并用pthread_attr_setstack函数来改变新建线程的栈位置
  - 应用程序可通过pthread_attr_getstacksize和pthread_attr_setstacksize函数读取或设置线程属性stacksize

12.4 同步属性

- 互斥量属性
  - 用pthread_mutexattr_init初始化pthread_mutexattr_t结构　　将用默认的互斥量属性初始化
  - 用pthread_mutexattr_destroy来反初始化
  - 进程共享属性
    - 在进程中，多个线程可以访问同一个同步对象。进程共享互斥量属性设置为PTHREAD_PROCESS_PRIVATE
    - 设置为PTHREAD_PROCESS_SHARED时，多个进程之间共享的内存数据块中分配的互斥量可用于这些进程的同步
    - pthread_mutexattr_getpshared和pthread_mutexattr_setpshared 查看或修改进程共享属性
  - 健壮属性
  ```

        - pthread_mutexattr_getrobust和pthread_mutexattr_setrobust 查看或修改健壮的互斥量属性的值
        - 默认值是PTHREAD_MUTEX_STALLED持有互斥量的进程终止时不需要采取特别的动作
        - 另一个值是PTHREAD_MUTEX_ROBUST
        - 线程可以调用pthread_mutex_consistent函数，指明与该互斥量相关的状态在互斥量解锁之前是一致的
    - 类型属性
        - PTHREAD_MUTEX_NORMAL
        - PTHREAD_MUTEX_ERRORCHECK
        - PTHREAD_MUTEX_RECURSIVE　允许没有解锁时重新加锁
        - PTHREAD_MUTEX_DEFAULT
        - 可以用pthread_mutexattr_gettype函数获得互斥量类型属性
        - pthread_mutexattr_settype修改互斥量类型属性
- 读写锁属性
    - pthread_rwlockattr_init初始化pthread_rwlockattr_t结构，pthread_rwlockattr_destroy反初始化
    - 读写锁支持的唯一属性是进程共享属性，与互斥量的进程共享属性是相同的
    - pthread_rwlockattr_getpshared
    - pthread_rwlockattr_setpshared
- 条件变量属性
    - pthread_condattr_init pthread_condattr_destroy初始化和反初始化条件变量属性
    - 进程共享属性
    - 时钟属性　控制计算pthread_cond_timedwait函数的超时参数采用的是哪个时钟
- 屏障属性
    - 只有进程共享属性

12.5 重入

- 如果一个函数在相同的时间点可以被多个线程安全地调用，就称该函数是线程安全的
- 很多函数并不是线程安全的，因为它们返回的数据存放在静态的内存缓冲区中，通过修改接口，要求调用者自己提供缓冲区可以使函数变为线程安全
- 如果一个函数对多个线程来说是可重入的，就说这个函数是线程安全的，但不能说明对信号处理程序来说该函数也是可重入的
- 提供了以线程安全的方式管理FILE对象的方法: 可以使用flockfile和ftrylockfile获取给定FILE对象关联的锁

12.6 线程特定数据

- 线程私有数据
    - 防止某个线程的数据与其他线程的数据相混淆
    - 提供了基于进程的接口适应多线程环境的机制
- 一个进程中的所有线程都可以访问这个进程的整个地址空间，除了使用寄存器以外，一个线程没办法阻止另一个线程访问它的数据
- 创建与数据关联的键　pthread_key_create
- 调用pthread_key_delete来取消键与线程特定数据值之间的关联关系
- pthread_once 如果每个线程都调用pthread_once,系统就能保证初始化例程initfn只被调用一次
- pthread_setspecific函数把键和线程特定数据关联起来
- pthread_getspecific函数获得线程特定数据的地址

- 可以使用线程特定数据来维护每个线程的数据缓冲副本
- 当线程调用pthread_exit或者线程执行返回，正常退出时，析构函数就会被调用
- malloc函数本身并不是异步信号安全的

12.7 取消选项

- 可取消状态　PTHREAD_CANCEL_ENABLE PTHREAD_CANCEL_DISABLE
- pthread_setcancelstate函数修改可取消状态
- 线程在取消请求发出以后还是继续运行，直到线程到达某个取消点
- 调用pthread_testcancel函数在程序中添加自己的取消点
- 默认的取消类似是推迟取消，即在调用pthread_cancel后，在线程到达取消点前，并不会出现真的取消
- 可以调用pthread_setcanceltype来修改取消类型

12.8 线程和信号

- 信号的处理是进程中所有线程共享的
- 进程中的信号是递送到单个线程的
- pthread_sigmask,工作在线程中，与sigprocmask基本相同
- 线程可以通过调用sigwait等待一个或多个信号的出现
- 要把信号发送给线程，可以调用pthread_kill
- 可以使用独立的控制线程进行信号处理，新建线程继承了现有的信号屏蔽字
- 为了避免错误行为发生，线程在调用sigwait之前，必须阻塞那些它正在等待的信号，sigwai会原子地取消信号集的阻塞状态，直到有新的信号被递送，在返回之前，sigwait将恢复线程的信号屏蔽字
- ```
  #include "apue.h"
  #include <pthread.h>

  int         quitflag;   /* set nonzero by thread */
  sigset_t    mask;

  pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
  pthread_cond_t waitloc = PTHREAD_COND_INITIALIZER;

  void *
  thr_fn(void *arg)
  {
      int err, signo;

      for (;;) {
          err = sigwait(&mask, &signo);
          if (err != 0)
              err_exit(err, "sigwait failed");
          switch (signo) {
          case SIGINT:
              printf("\ninterrupt\n");
              break;

          case SIGQUIT:
              pthread_mutex_lock(&lock);
  ```
  ```

```
            quitflag = 1;
            pthread_mutex_unlock(&lock);
            pthread_cond_signal(&waitloc);
            return(0);

        default:
            printf("unexpected signal %d\n", signo);
            exit(1);
        }
    }
}

int
main(void)
{
    int         err;
    sigset_t    oldmask;
    pthread_t   tid;

    sigemptyset(&mask);
    sigaddset(&mask, SIGINT);
    sigaddset(&mask, SIGQUIT);
    if ((err = pthread_sigmask(SIG_BLOCK, &mask, &oldmask)) != 0)
        err_exit(err, "SIG_BLOCK error");

    err = pthread_create(&tid, NULL, thr_fn, 0);
    if (err != 0)
        err_exit(err, "can't create thread");

    pthread_mutex_lock(&lock);
    while (quitflag == 0)
        pthread_cond_wait(&waitloc, &lock);
    pthread_mutex_unlock(&lock);

    /* SIGQUIT has been caught and is now blocked; do whatever */
    quitflag = 0;

    /* reset signal mask which unblocks SIGQUIT */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("SIG_SETMASK error");
    exit(0);
}
```

12.9 线程和fork

- 当线程调用fork时，就为子进程创建了整个进程地址空间的副本
- 在子进程内部，只存在一个线程，它是由父进程中调用fork的线程的副本构成
- POSIX.1声明，在fork返回和子进程调用其中一个exec函数之间，子进程只能调用异步信号安全的函数
- pthread_atfork函数最多可以安装3个帮助清理锁的函数

```c
#include "apue.h"
#include <pthread.h>

pthread_mutex_t lock1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t lock2 = PTHREAD_MUTEX_INITIALIZER;

void
prepare(void)
{
    int err;

    printf("preparing locks...\n");
    if ((err = pthread_mutex_lock(&lock1)) != 0)
        err_cont(err, "can't lock lock1 in prepare handler");
    if ((err = pthread_mutex_lock(&lock2)) != 0)
        err_cont(err, "can't lock lock2 in prepare handler");
}

void
parent(void)
{
    int err;

    printf("parent unlocking locks...\n");
    if ((err = pthread_mutex_unlock(&lock1)) != 0)
        err_cont(err, "can't unlock lock1 in parent handler");
    if ((err = pthread_mutex_unlock(&lock2)) != 0)
        err_cont(err, "can't unlock lock2 in parent handler");
}

void
child(void)
{
    int err;

    printf("child unlocking locks...\n");
    if ((err = pthread_mutex_unlock(&lock1)) != 0)
        err_cont(err, "can't unlock lock1 in child handler");
    if ((err = pthread_mutex_unlock(&lock2)) != 0)
        err_cont(err, "can't unlock lock2 in child handler");
}

void *
thr_fn(void *arg)
{
    printf("thread started...\n");
    pause();
    return(0);
}

int
main(void)
{
```

```
    int         err;
    pid_t       pid;
    pthread_t   tid;

    if ((err = pthread_atfork(prepare, parent, child)) != 0)
        err_exit(err, "can't install fork handlers");
    if ((err = pthread_create(&tid, NULL, thr_fn, 0)) != 0)
        err_exit(err, "can't create thread");

    sleep(2);
    printf("parent about to fork...\n");

    if ((pid = fork()) < 0)
        err_quit("fork failed");
    else if (pid == 0)  /* child */
        printf("child returned from fork\n");
    else        /* parent */
        printf("parent returned from fork\n");
    exit(0);
}
```

12.10 线程和I/O

- 使用pread,使偏移量的设定和数据的读取成为一个原子操作

第13章 守护进程

- 守护进程是生存期长的一种进程，常常在系统引导装入时启动，仅在系统关闭时才终止。因为它们没有控制终端，所以是在后台运行的

13.2 守护进程的特征

- ps -efj　内核守护进程的名字出现在方括号中
- 父进程ID为0的各进程通常是内核进程，作为系统引导装入过程的一部分而启动，内核进程是特殊的，通常存在于系统的整个生命期，以超级用户特权运行，无控制终端，无命令行
- init是例外，它是一个由内核在引导装入时启动的用户层次的命令
- 使用一个kthread的特殊内核进程来创建其他内核进程
- kswapd flush sync_supers jbd
- 进程1通常是init,是一个内核守护进程
- rpcbind
- rsyslogd
- inetd 帧听系统网络接口
- 内核守护进程　用户层守护进程
- 用户层守护进程的父进程是init进程

13.3 编程规则

- 首先要做的是调用umask将文件模式创建屏蔽字设置为一个已知值
- 调用fork,然后使父进程exit

- 调用setsid创建一个新会话
- 将当前工作目录改为根目录
- 关闭不再需要的文件描述符
- 某些守护进程打开/dev/null使其具有文件描述符0 1 2

13.4 出错记录

- 内核例程可以调用log函数，任何一个用户进程都可以通过打开并读取 /dev/klog设备来读取这些消息
- 大多数用户进程(守护进程)可以调用syslog函数来产生日志消息，被发送到UNIX域数据报套接字/dev/log
- 可将日志消息发送到UDP端口514
- syslogd守护进程读取这三种日志消息，守护进程在启动时读取配置文件/etc/syslog.conf
- openlog
- syslog产生一个日志消息
- setlogmask函数用于设置进程的记录优先级屏蔽字
- closelog
- logger程序可作为向syslog设施发送日志消息的方法，logger命令是专门为以非交互式运行的需要产生日志消息的shell脚本设计的
- vsyslog
- 大多数syslog实现将使消息短时间处于队列中

13.5 单实例守护进程

- 为了正常运行，某些守护进程会实现为，在任一时刻只运行该守护进程的一个副本
- 文件和记录锁方式，保证一个守护进程只有一个副本在运行，如果每一个守护进程创建一个有固定名字的文件，并在该文件的整体上加一把写锁，那么只允许创建一把这样的写锁。
- ```
  #include <unistd.h>
  #include <stdlib.h>
  #include <fcntl.h>
  #include <syslog.h>
  #include <string.h>
  #include <errno.h>
  #include <stdio.h>
  #include <sys/stat.h>

  #define LOCKFILE "/var/run/daemon.pid"
  #define LOCKMODE (S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH)

  extern int lockfile(int);

  int
  already_running(void)
  {
      int     fd;
      char    buf[16];

      fd = open(LOCKFILE, O_RDWR|O_CREAT, LOCKMODE);
      if (fd < 0) {
          syslog(LOG_ERR, "can't open %s: %s", LOCKFILE, strerror(errno));
          exit(1);
      }
  ```
  ```

```
    if (lockfile(fd) < 0) {
        if (errno == EACCES || errno == EAGAIN) {
            close(fd);
            return(1);
        }
        syslog(LOG_ERR, "can't lock %s: %s", LOCKFILE, strerror(errno));
        exit(1);
    }
    ftruncate(fd, 0);
    sprintf(buf, "%ld", (long)getpid());
    write(fd, buf, strlen(buf)+1);
    return(0);
}
```

- 如果该文件已经加了锁，那么lockfile函数将失败

13.5 守护进程的惯例

- 若守护进程使用锁文件，该文件通常存储在/var/run目录中，锁文件的名字是name.pid
- 配置文件naem.conf在/etc目录中
- 守护进程可用命令行启动，通常是由系统初始化脚本之一(/etc/rc* 或/etc/init.d/*)启动的。在/etc/inittab中为该守护进程包括respawn记录项，这样，init将重新启动终止的守护进程
- 守护进程将读取SIGHUP信号，重新读取配置文件
- ```
  #include "apue.h"
  #include <pthread.h>
  #include <syslog.h>

  sigset_t    mask;

  extern int already_running(void);

  void
  reread(void)
  {
      /* ... */
  }

  void *
  thr_fn(void *arg)
  {
      int err, signo;

      for (;;) {
          err = sigwait(&mask, &signo);
          if (err != 0) {
              syslog(LOG_ERR, "sigwait failed");
              exit(1);
          }
  ```
  ```

```
        switch (signo) {
        case SIGHUP:
            syslog(LOG_INFO, "Re-reading configuration file");
            reread();
            break;

        case SIGTERM:
            syslog(LOG_INFO, "got SIGTERM; exiting");
            exit(0);

        default:
            syslog(LOG_INFO, "unexpected signal %d\n", signo);
        }
    }
    return(0);
}

int
main(int argc, char *argv[])
{
    int                 err;
    pthread_t           tid;
    char                *cmd;
    struct sigaction    sa;

    if ((cmd = strrchr(argv[0], '/')) == NULL)
        cmd = argv[0];
    else
        cmd++;

    /*
     * Become a daemon.
     */
    daemonize(cmd);

    /*
     * Make sure only one copy of the daemon is running.
     */
    if (already_running()) {
        syslog(LOG_ERR, "daemon already running");
        exit(1);
    }

    /*
     * Restore SIGHUP default and block all signals.
     */
    sa.sa_handler = SIG_DFL;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    if (sigaction(SIGHUP, &sa, NULL) < 0)
        err_quit("%s: can't restore SIGHUP default");
    sigfillset(&mask);
    if ((err = pthread_sigmask(SIG_BLOCK, &mask, NULL)) != 0)
```

```
        err_exit(err, "SIG_BLOCK error");

    /*
     * Create a thread to handle SIGHUP and SIGTERM.
     */
    err = pthread_create(&tid, NULL, thr_fn, 0);
    if (err != 0)
        err_exit(err, "can't create thread");

    /*
     * Proceed with the rest of the daemon.
     */
    /* ... */
    exit(0);
}
```

13.7 客户进程- 服务器进程模型

- 守护进程常常用作服务器进程

第14章　高级I/O

14.2 非阻塞I/O

- 低速系统调用是可能会使进程永远阻塞的一类系统调用
- 非阻塞I/O可以发出open read write这样的I/O操作，并使这些操作不会永远阻塞，如果这样的操作不能完成，则调用立即出错返回
- 两种指定为非阻塞I/O的方法
  - 调用open获得描述符，可指定O_NONBLOCK标志
  - 对一个打开的描述符，调用fcnt1,由该函数打开O_NONBLOCK文件状态标志
- ```
  #include "apue.h"
  #include <errno.h>
  #include <fcntl.h>

  char    buf[500000];

  int
  main(void)
  {
      int     ntowrite, nwrite;
      char    *ptr;

      ntowrite = read(STDIN_FILENO, buf, sizeof(buf));
      fprintf(stderr, "read %d bytes\n", ntowrite);

      set_fl(STDOUT_FILENO, O_NONBLOCK);  /* set nonblocking */

      ptr = buf;
      while (ntowrite > 0) {
          errno = 0;
  ```
  ```

```
        nwrite = write(STDOUT_FILENO, ptr, ntowrite);
        fprintf(stderr, "nwrite = %d, errno = %d\n", nwrite, errno);

        if (nwrite > 0) {
            ptr += nwrite;
            ntowrite -= nwrite;
        }
    }

    clr_fl(STDOUT_FILENO, O_NONBLOCK);  /* clear nonblocking */

    exit(0);
}
```

14.3 记录锁

- 记录锁的功能: 当一个进程正在读或修改文件的某个部分时，使用记录锁可以阻止其他进程修改同一文件区
- fcnt1记录锁
  - fcnt1
  - 对于记录锁，cmd是F_GETLK F_SETLK F_SETLKW
  - 第三个参数是一个指向flock结构的指针
  - 任意多个进程在一个给定的字节上可以有一把共享的读锁，但是在一个给定的字节上只能有一个进程有一把独占写锁
  - 兼容性规则适用于不同进程提出的锁请求，并不适用于单个进程提出的多个锁请求
- 死锁实例

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

static void
lockabyte(const char *name, int fd, off_t offset)
{
    if (writew_lock(fd, offset, SEEK_SET, 1) < 0)
        err_sys("%s: writew_lock error", name);
    printf("%s: got the lock, byte %lld\n", name, (long long)offset);
}

int
main(void)
{
    int     fd;
    pid_t   pid;

    /*
     * Create a file and write two bytes to it.
     */
    if ((fd = creat("templock", FILE_MODE)) < 0)
        err_sys("creat error");
    if (write(fd, "ab", 2) != 2)
        err_sys("write error");
```

```
    TELL_WAIT();
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {          /* child */
        lockabyte("child", fd, 0);
        TELL_PARENT(getppid());
        WAIT_PARENT();
        lockabyte("child", fd, 1);
    } else {                        /* parent */
        lockabyte("parent", fd, 1);
        TELL_CHILD(pid);
        WAIT_CHILD();
        lockabyte("parent", fd, 0);
    }
    exit(0);
}
```

    - 检测到死锁时，内核必须选择一个进程接收出错返回
- 锁的隐含继承和释放
    - 当一个进程终止时，它所建立的锁全部释放，无论一个描述符何时关闭，该进程通过这一描述符引用的文件上的任何一把锁都会释放
    - 由fork产生的子进程不继承父进程所设置的锁
    - 在执行exec后，新程序可以继承原执行程序的锁
- FressBSD实现
    - 每个lockf结构描述了一个给定进程的一个加锁区域
    - 守护进程可用一把文件锁来保证只有该守护进程的唯一副本在运行，在文件整体上加一把写锁
    - ```
      #include <unistd.h>
      #include <fcntl.h>

      int
      lockfile(int fd)
      {
          struct flock fl;

          fl.l_type = F_WRLCK;
          fl.l_start = 0;
          fl.l_whence = SEEK_SET;
          fl.l_len = 0;
          return(fcntl(fd, F_SETLK, &fl));
      }
      ```
- 在文件尾端加锁
    - 当前偏移量和文件尾端可能会不断变化，而这种变化又不应影响现有锁的状态，所以内核必须独立于当前文件偏移量或文件尾端而记住锁
- 建议性锁和强制性锁
    - 对一个特定文件打开其设置组ID位、关闭其组执行位便开启了对该文件的强制性锁机制
    - 合作进程
    - 非合作进程
      ```

- 建议性锁不能阻止对数据库文件有写权限的任何其他进程写这个数据库文件
- 强制性锁会让内核检查每一个open read write 验证调用进程是否违背了正在访问的文件上的某一把锁
- 如果欲打开的文件具有强制性纪录锁，而且open调用中的标志指定为O_TRUNC或O_CREAT，则不论是否指定O_NONBLOCK, open都立即出错，errno被设置为EAGAIN
- 强制性锁对unlink函数没有影响
- Linux中的　strace命令提供一个进程的系统调用跟踪信息
- ```
  #include "apue.h"
  #include <errno.h>
  #include <fcntl.h>
  #include <sys/wait.h>

  int
  main(int argc, char *argv[])
  {
      int             fd;
      pid_t           pid;
      char            buf[5];
      struct stat     statbuf;

      if (argc != 2) {
          fprintf(stderr, "usage: %s filename\n", argv[0]);
          exit(1);
      }
      if ((fd = open(argv[1], O_RDWR | O_CREAT | O_TRUNC, FILE_MODE)) < 0)
          err_sys("open error");
      if (write(fd, "abcdef", 6) != 6)
          err_sys("write error");

      /* turn on set-group-ID and turn off group-execute */
      if (fstat(fd, &statbuf) < 0)
          err_sys("fstat error");
      if (fchmod(fd, (statbuf.st_mode & ~S_IXGRP) | S_ISGID) < 0)
          err_sys("fchmod error");

      TELL_WAIT();

      if ((pid = fork()) < 0) {
          err_sys("fork error");
      } else if (pid > 0) {   /* parent */
          /* write lock entire file */
          if (write_lock(fd, 0, SEEK_SET, 0) < 0)
              err_sys("write_lock error");

          TELL_CHILD(pid);

          if (waitpid(pid, NULL, 0) < 0)
              err_sys("waitpid error");
      } else {                /* child */
          WAIT_PARENT();      /* wait for parent to set lock */
  ```
  ```

```
        set_fl(fd, O_NONBLOCK);

        /* first let's see what error we get if region is locked */
        if (read_lock(fd, 0, SEEK_SET, 0) != -1)    /* no wait */
            err_sys("child: read_lock succeeded");
        printf("read_lock of already-locked region returns %d\n",
          errno);

        /* now try to read the mandatory locked file */
        if (lseek(fd, 0, SEEK_SET) == -1)
            err_sys("lseek error");
        if (read(fd, buf, 2) < 0)
            err_ret("read failed (mandatory locking works)");
        else
            printf("read OK (no mandatory locking), buf = %2.2s\n",
              buf);
    }
    exit(0);
}
```

14.4 I/O多路转换

- 父进程、子进程
- 多线程
- 轮询方式
- 异步I/O
- poll pselect select这三个函数能够执行I/O多路转接
- 函数select和pselect
    - FD_ISSET
    - FD_CLR
    - FD_SET
    - FD_ZERO
    - 异常条件包括: 在网络连接上到达带外的数据，或者处于数据包模式的伪终端上发生了某些条件
    - 如果在一个描述符上碰到了文件尾端，则select会认为该描述符是可读的，然后调用read,它返回0
    - pselect使用timespec结构
    - pselect可使用信号屏蔽字，sigmask指向一信号屏蔽字，在调用pselect时，以原子操作的方式安装该信号屏蔽字，在返回时，恢复以前的信号屏蔽字
    - pselect的超时值被声明为const
- 函数poll
    - poll构造一个pollfd结构的数组，每个数组元素指定一个描述符编号以及我们对该描述符感兴趣的条件
    - ```
      struct pollfd{
          int fd;
          short events;
          short events;
      };
      ```
    - 文件尾端与挂断之间的区别
      ```

- select和poll的可中断性

14.5 异步I/O

- System V异步I/O
  - System V的异步信号是SIGPOLL
  - 调用ioct1,启动异步I/O
  - 建立信号处理程序
- BSD异步I/O
  - 异步I/O是信号SIGIO和SIGURG的组合
  - SIGIO是通用异步I/O
  - SIGURG是通知进程网络连接上的带外数据已经到达
- POSIX异步I/O
  - 这些异步I/O接口使用AIO控制块来描述I/O操作
  - aiocb结构定义了AIO控制块
  - 在进行异步I/O之前需要先初始化AIO控制块
  - aio_read函数进行异步读操作，aio_write函数进行异步写操作
  - aio_fsync
  - aio_error 获知一个异步读、写或者同步操作的完成状态
  - aio_return异步操作成功，获取异步操作的返回值
  - aio_suspend函数阻塞进程，直到操作完成
  - aio_cancel
  - lio_listio
  - POSIX异步I/O接口现在是Single UNIX Specification的基本部分
  - ```
    #include "apue.h"
    #include <ctype.h>
    #include <fcntl.h>
    #include <aio.h>
    #include <errno.h>

    #define BSZ 4096
    #define NBUF 8

    enum rwop {
        UNUSED = 0,
        READ_PENDING = 1,
        WRITE_PENDING = 2
    };

    struct buf {
        enum rwop     op;
        int           last;
        struct aiocb  aiocb;
        unsigned char data[BSZ];
    };
    ```
    ```

```
struct buf bufs[NBUF];

unsigned char
translate(unsigned char c)
{
    /* same as before */
}

int
main(int argc, char* argv[])
{
    int                 ifd, ofd, i, j, n, err, numop;
    struct stat         sbuf;
    const struct aiocb  *aiolist[NBUF];
    off_t               off = 0;

    if (argc != 3)
        err_quit("usage: rot13 infile outfile");
    if ((ifd = open(argv[1], O_RDONLY)) < 0)
        err_sys("can't open %s", argv[1]);
    if ((ofd = open(argv[2], O_RDWR|O_CREAT|O_TRUNC, FILE_MODE)) < 0)
        err_sys("can't create %s", argv[2]);
    if (fstat(ifd, &sbuf) < 0)
        err_sys("fstat failed");

    /* initialize the buffers */
    for (i = 0; i < NBUF; i++) {
        bufs[i].op = UNUSED;
        bufs[i].aiocb.aio_buf = bufs[i].data;
        bufs[i].aiocb.aio_sigevent.sigev_notify = SIGEV_NONE;
        aiolist[i] = NULL;
    }

    numop = 0;
    for (;;) {
        for (i = 0; i < NBUF; i++) {
            switch (bufs[i].op) {
            case UNUSED:
                /*
                 * Read from the input file if more data
                 * remains unread.
                 */
                if (off < sbuf.st_size) {
                    bufs[i].op = READ_PENDING;
                    bufs[i].aiocb.aio_fildes = ifd;
                    bufs[i].aiocb.aio_offset = off;
                    off += BSZ;
                    if (off >= sbuf.st_size)
                        bufs[i].last = 1;
                    bufs[i].aiocb.aio_nbytes = BSZ;
                    if (aio_read(&bufs[i].aiocb) < 0)
                        err_sys("aio_read failed");
                    aiolist[i] = &bufs[i].aiocb;
```

```
                numop++;
            }
            break;

        case READ_PENDING:
            if ((err = aio_error(&bufs[i].aiocb)) == EINPROGRESS)
                continue;
            if (err != 0) {
                if (err == -1)
                    err_sys("aio_error failed");
                else
                    err_exit(err, "read failed");
            }

            /*
             * A read is complete; translate the buffer
             * and write it.
             */
            if ((n = aio_return(&bufs[i].aiocb)) < 0)
                err_sys("aio_return failed");
            if (n != BSZ && !bufs[i].last)
                err_quit("short read (%d/%d)", n, BSZ);
            for (j = 0; j < n; j++)
                bufs[i].data[j] = translate(bufs[i].data[j]);
            bufs[i].op = WRITE_PENDING;
            bufs[i].aiocb.aio_fildes = ofd;
            bufs[i].aiocb.aio_nbytes = n;
            if (aio_write(&bufs[i].aiocb) < 0)
                err_sys("aio_write failed");
            /* retain our spot in aiolist */
            break;

        case WRITE_PENDING:
            if ((err = aio_error(&bufs[i].aiocb)) == EINPROGRESS)
                continue;
            if (err != 0) {
                if (err == -1)
                    err_sys("aio_error failed");
                else
                    err_exit(err, "write failed");
            }

            /*
             * A write is complete; mark the buffer as unused.
             */
            if ((n = aio_return(&bufs[i].aiocb)) < 0)
                err_sys("aio_return failed");
            if (n != bufs[i].aiocb.aio_nbytes)
                err_quit("short write (%d/%d)", n, BSZ);
            aiolist[i] = NULL;
            bufs[i].op = UNUSED;
            numop--;
            break;
```

```
            }
        }
        if (numop == 0) {
            if (off >= sbuf.st_size)
                break;
        } else {
            if (aio_suspend(aiolist, NBUF, NULL) < 0)
                err_sys("aio_suspend failed");
        }
    }

    bufs[0].aiocb.aio_fildes = ofd;
    if (aio_fsync(O_SYNC, &bufs[0].aiocb) < 0)
        err_sys("aio_fsync failed");
    exit(0);
}
```

14.6 函数readv和writev

- 用于在一次函数调用中读写多个非连续缓冲区
- ```
  struct iovec{
      void*iov_base;
      size_t iov_len;
  };
  ```

14.7 函数readn和writen

- 管道、FIFO以及某些设备(如终端和网络)有以下两种性质
  - 一次read操作所返回的数据可能少于所要求的数据
  - 一次write操作的返回值也可能少于指定输出的字节数
- readn writen并不是哪个标准的组成部分
- ```
  #include "apue.h"

  ssize_t             /* Read "n" bytes from a descriptor  */
  readn(int fd, void *ptr, size_t n)
  {
      size_t      nleft;
      ssize_t     nread;

      nleft = n;
      while (nleft > 0) {
          if ((nread = read(fd, ptr, nleft)) < 0) {
              if (nleft == n)
                  return(-1); /* error, return -1 */
              else
                  break;      /* error, return amount read so far */
          } else if (nread == 0) {
              break;          /* EOF */
          }
          nleft -= nread;
  ```

```
        ptr   += nread;
    }
    return(n - nleft);      /* return >= 0 */
}
```

- ```
  #include "apue.h"

  ssize_t             /* Write "n" bytes to a descriptor  */
  writen(int fd, const void *ptr, size_t n)
  {
      size_t      nleft;
      ssize_t     nwritten;

      nleft = n;
      while (nleft > 0) {
          if ((nwritten = write(fd, ptr, nleft)) < 0) {
              if (nleft == n)
                  return(-1); /* error, return -1 */
              else
                  break;      /* error, return amount written so far */
          } else if (nwritten == 0) {
              break;
          }
          nleft -= nwritten;
          ptr   += nwritten;
      }
      return(n - nleft);      /* return >= 0 */
  }
  ```

14.8 存储映射I/O

- 能将一个磁盘文件映射到存储空间中的一个缓冲区上，当从缓冲区中取数据时，就相当于读文件中的相应字节，将数据写入缓冲区时，相应字节就自动写入文件。即可在不使用read和write的情况下执行I/O
- mmap函数　告诉内核将一个给定的文件映射到一个存储区域中
- 不能用mmap将数据添加到文件中，必须先加长该文件
- SIGSEGV 将数据存入只读映射区时
- SIGBUS　访问不存在的映射区的某个部分时
- 子进程能通过fork继承存储映射区(因为存储映射区是地址空间中的一部分)
- mprotect可以更改一个现有映射的权限
- 如果共享映射中的页已修改，那么可以调用msync将该页冲洗到被映射的文件中
- 当进程终止时，会自动解除存储映射区的映射，或者直接调用munmap函数也可以解除映射区
- 数据被写到文件的确切时间依赖于系统的页管理算法，如果想确保数据安全地写到文件中，则需要在进程终止前以MS_SYNC标志调用msync
- ```
  #include "apue.h"
  #include <fcntl.h>
  #include <sys/mman.h>
  ```

```c
#define COPYINCR (1024*1024*1024)   /* 1 GB */

int
main(int argc, char *argv[])
{
    int         fdin, fdout;
    void        *src, *dst;
    size_t      copysz;
    struct stat sbuf;
    off_t       fsz = 0;

    if (argc != 3)
        err_quit("usage: %s <fromfile> <tofile>", argv[0]);

    if ((fdin = open(argv[1], O_RDONLY)) < 0)
        err_sys("can't open %s for reading", argv[1]);

    if ((fdout = open(argv[2], O_RDWR | O_CREAT | O_TRUNC,
      FILE_MODE)) < 0)
        err_sys("can't creat %s for writing", argv[2]);

    if (fstat(fdin, &sbuf) < 0)         /* need size of input file */
        err_sys("fstat error");

    if (ftruncate(fdout, sbuf.st_size) < 0) /* set output file size */
        err_sys("ftruncate error");

    while (fsz < sbuf.st_size) {
        if ((sbuf.st_size - fsz) > COPYINCR)
            copysz = COPYINCR;
        else
            copysz = sbuf.st_size - fsz;

        if ((src = mmap(0, copysz, PROT_READ, MAP_SHARED,
          fdin, fsz)) == MAP_FAILED)
            err_sys("mmap error for input");
        if ((dst = mmap(0, copysz, PROT_READ | PROT_WRITE,
          MAP_SHARED, fdout, fsz)) == MAP_FAILED)
            err_sys("mmap error for output");

        memcpy(dst, src, copysz);   /* does the file copy */
        munmap(src, copysz);
        munmap(dst, copysz);
        fsz += copysz;
    }
    exit(0);
}
```

第15章　进程间通信

- 进程间通信IPC

15.2 管道

- 局限性
  - 历史上是半双工的，不应预先假设系统支持全双工管道
  - 管道只能在具有公共祖先的两个进程之间使用
- 半双工管道是最常用的IPC形式
- 每当在管道中键入一个命令序列，让shell执行时，shell都会为每一条命令单独创建一个进程，然后使用管道将前一条命令进程的标准输出与后一条命令的标准输入相连接
- pipe函数创建
- 通常，进程会先调用pipe,接着调用fork, 从而创建从父进程到子进程的IPC通道
- 当管道的一端被关闭后，下列两条规则
  - 当读一个写端已被关闭的管道时，在所有数据都被读取后，read返回0,表示文件结束
  - 如果写一个读端已被关闭的管道，则产生信号SIGPIPE
  - 常量PIPE_BUF规定了内核的管道缓冲区大小
- ```c
  #include "apue.h"

  int
  main(void)
  {
      int     n;
      int     fd[2];
      pid_t   pid;
      char    line[MAXLINE];

      if (pipe(fd) < 0)
          err_sys("pipe error");
      if ((pid = fork()) < 0) {
          err_sys("fork error");
      } else if (pid > 0) {       /* parent */
          close(fd[0]);
          write(fd[1], "hello world\n", 12);
      } else {                    /* child */
          close(fd[1]);
          n = read(fd[0], line, MAXLINE);
          write(STDOUT_FILENO, line, n);
      }
      exit(0);
  }
  ```
- 通过管道将输出直接送到分页程序
  - ```c
    #include "apue.h"
    #include <sys/wait.h>

    #define DEF_PAGER   "/bin/more"     /* default pager program */

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

```
    int     n;
    int     fd[2];
    pid_t   pid;
    char    *pager, *argv0;
    char    line[MAXLINE];
    FILE    *fp;

    if (argc != 2)
        err_quit("usage: a.out <pathname>");

    if ((fp = fopen(argv[1], "r")) == NULL)
        err_sys("can't open %s", argv[1]);
    if (pipe(fd) < 0)
        err_sys("pipe error");

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {                                   /* parent */
        close(fd[0]);       /* close read end */

        /* parent copies argv[1] to pipe */
        while (fgets(line, MAXLINE, fp) != NULL) {
            n = strlen(line);
            if (write(fd[1], line, n) != n)
                err_sys("write error to pipe");
        }
        if (ferror(fp))
            err_sys("fgets error");

        close(fd[1]);   /* close write end of pipe for reader */

        if (waitpid(pid, NULL, 0) < 0)
            err_sys("waitpid error");
        exit(0);
    } else {                                            /* child */
        close(fd[1]);   /* close write end */
        if (fd[0] != STDIN_FILENO) {
            if (dup2(fd[0], STDIN_FILENO) != STDIN_FILENO)
                err_sys("dup2 error to stdin");
            close(fd[0]);   /* don't need this after dup2 */
        }

        /* get arguments for execl() */
        if ((pager = getenv("PAGER")) == NULL)
            pager = DEF_PAGER;
        if ((argv0 = strrchr(pager, '/')) != NULL)
            argv0++;        /* step past rightmost slash */
        else
            argv0 = pager;  /* no slash in pager */

        if (execl(pager, argv0, (char *)0) < 0)
            err_sys("execl error for %s", pager);
    }
```

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

- 图10-24提供了TELL_WAIT TELL_PARENT TELL_CHILD WAIT_PARENT和WAIT_CHILD的使用信号的实现
- 图15-7提供了TELL_WAIT TELL_PARENT TELL_CHILD WAIT_PARENT和WAIT_CHILD的使用管道的实现
- 用两个管道实现子进程和父进程同步
- ```
  #include "apue.h"

  static int  pfd1[2], pfd2[2];

  void
  TELL_WAIT(void)
  {
      if (pipe(pfd1) < 0 || pipe(pfd2) < 0)
          err_sys("pipe error");
  }

  void
  TELL_PARENT(pid_t pid)
  {
      if (write(pfd2[1], "c", 1) != 1)
          err_sys("write error");
  }

  void
  WAIT_PARENT(void)
  {
      char    c;

      if (read(pfd1[0], &c, 1) != 1)
          err_sys("read error");

      if (c != 'p')
          err_quit("WAIT_PARENT: incorrect data");
  }

  void
  TELL_CHILD(pid_t pid)
  {
      if (write(pfd1[1], "p", 1) != 1)
          err_sys("write error");
  }

  void
  WAIT_CHILD(void)
  {
      char    c;

      if (read(pfd2[0], &c, 1) != 1)
          err_sys("read error");
  ```
  ```

```
    if (c != 'c')
        err_quit("WAIT_CHILD: incorrect data");
}
```

15.3 函数popen和pclose

- 创建一个管道，fork一个子进程，关闭未使用的管道端，执行一个shell运行命令，然后等待命令终止
- 函数popen先执行fork,然后调用esec执行cmdstring,并且返回一个标准I/O文件指针
- pclose关闭标准I/O流，等待命令终止，然后返回shell的终止状态
- 用popen向分页程序传送文件
- ```
  #include "apue.h"
  #include <sys/wait.h>

  #define PAGER   "${PAGER:-more}" /* environment variable, or default */

  int
  main(int argc, char *argv[])
  {
      char    line[MAXLINE];
      FILE    *fpin, *fpout;

      if (argc != 2)
          err_quit("usage: a.out <pathname>");
      if ((fpin = fopen(argv[1], "r")) == NULL)
          err_sys("can't open %s", argv[1]);

      if ((fpout = popen(PAGER, "w")) == NULL)
          err_sys("popen error");

      /* copy argv[1] to pager */
      while (fgets(line, MAXLINE, fpin) != NULL) {
          if (fputs(line, fpout) == EOF)
              err_sys("fputs error to pipe");
      }
      if (ferror(fpin))
          err_sys("fgets error");
      if (pclose(fpout) == -1)
          err_sys("pclose error");

      exit(0);
  }
  ```
- ${PAGER:-more} pipe2.c
- ${PAGER:-more}的意思是: 如果shell变量PAGER已经定义，且其值非空，则使用其值，否则使用字符串more
- ```
  #include "apue.h"
  #include <errno.h>
  #include <fcntl.h>
  #include <sys/wait.h>
  ```

```c
/*
 * Pointer to array allocated at run-time.
 */
static pid_t    *childpid = NULL;

/*
 * From our open_max(), {Prog openmax}.
 */
static int      maxfd;

FILE *
popen(const char *cmdstring, const char *type)
{
    int     i;
    int     pfd[2];
    pid_t   pid;
    FILE    *fp;

    /* only allow "r" or "w" */
    if ((type[0] != 'r' && type[0] != 'w') || type[1] != 0) {
        errno = EINVAL;
        return(NULL);
    }

    if (childpid == NULL) {     /* first time through */
        /* allocate zeroed out array for child pids */
        maxfd = open_max();
        if ((childpid = calloc(maxfd, sizeof(pid_t))) == NULL)
            return(NULL);
    }

    if (pipe(pfd) < 0)
        return(NULL);   /* errno set by pipe() */
    if (pfd[0] >= maxfd || pfd[1] >= maxfd) {
        close(pfd[0]);
        close(pfd[1]);
        errno = EMFILE;
        return(NULL);
    }

    if ((pid = fork()) < 0) {
        return(NULL);   /* errno set by fork() */
    } else if (pid == 0) {                          /* child */
        if (*type == 'r') {
            close(pfd[0]);
            if (pfd[1] != STDOUT_FILENO) {
                dup2(pfd[1], STDOUT_FILENO);
                close(pfd[1]);
            }
        } else {
            close(pfd[1]);
            if (pfd[0] != STDIN_FILENO) {
                dup2(pfd[0], STDIN_FILENO);
```

```
                close(pfd[0]);
            }
        }

        /* close all descriptors in childpid[] */
        for (i = 0; i < maxfd; i++)
            if (childpid[i] > 0)
                close(i);

        execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
        _exit(127);
    }

    /* parent continues... */
    if (*type == 'r') {
        close(pfd[1]);
        if ((fp = fdopen(pfd[0], type)) == NULL)
            return(NULL);
    } else {
        close(pfd[0]);
        if ((fp = fdopen(pfd[1], type)) == NULL)
            return(NULL);
    }

    childpid[fileno(fp)] = pid; /* remember child pid for this fd */
    return(fp);
}

int
pclose(FILE *fp)
{
    int     fd, stat;
    pid_t   pid;

    if (childpid == NULL) {
        errno = EINVAL;
        return(-1);     /* popen() has never been called */
    }

    fd = fileno(fp);
    if (fd >= maxfd) {
        errno = EINVAL;
        return(-1);     /* invalid file descriptor */
    }
    if ((pid = childpid[fd]) == 0) {
        errno = EINVAL;
        return(-1);     /* fp wasn't opened by popen() */
    }

    childpid[fd] = 0;
    if (fclose(fp) == EOF)
        return(-1);
```

```
    while (waitpid(pid, &stat, 0) < 0)
        if (errno != EINTR)
            return(-1); /* error other than EINTR from waitpid() */

    return(stat);   /* return child's termination status */
}
```

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

  int
  main(void)
  {
      char    line[MAXLINE];
      FILE    *fpin;

      if ((fpin = popen("myuclc", "r")) == NULL)
          err_sys("popen error");
      for ( ; ; ) {
          fputs("prompt> ", stdout);
          fflush(stdout);
          if (fgets(line, MAXLINE, fpin) == NULL) /* read from pipe */
              break;
          if (fputs(line, stdout) == EOF)
              err_sys("fputs error to pipe");
      }
      if (pclose(fpin) == -1)
          err_sys("pclose error");
      putchar('\n');
      exit(0);
  }
  ```

- 因为标准输出是行缓冲的，而提示并不包含换行符，所以在写了提示之后，需要调用fflush

15.4 协同进程

- Unix系统过滤程序从标准输入读取数据，向标准输出写数据
- 当一个过滤程序既产生某个过滤程序的输入，又读取该过滤程序的输出时，它就变成了协同进程
- 协同进程通常在shell的后台运行，其标准输入和标准输出通过管道连接到另一个程序

- ```
  #include "apue.h"

  static void sig_pipe(int);      /* our signal handler */

  int
  main(void)
  {
      int     n, fd1[2], fd2[2];
      pid_t   pid;
      char    line[MAXLINE];

      if (signal(SIGPIPE, sig_pipe) == SIG_ERR)
  ```

```
        err_sys("signal error");

    if (pipe(fd1) < 0 || pipe(fd2) < 0)
        err_sys("pipe error");

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {                           /* parent */
        close(fd1[0]);
        close(fd2[1]);

        while (fgets(line, MAXLINE, stdin) != NULL) {
            n = strlen(line);
            if (write(fd1[1], line, n) != n)
                err_sys("write error to pipe");
            if ((n = read(fd2[0], line, MAXLINE)) < 0)
                err_sys("read error from pipe");
            if (n == 0) {
                err_msg("child closed pipe");
                break;
            }
            line[n] = 0;    /* null terminate */
            if (fputs(line, stdout) == EOF)
                err_sys("fputs error");
        }

        if (ferror(stdin))
            err_sys("fgets error on stdin");
        exit(0);
    } else {                                    /* child */
        close(fd1[1]);
        close(fd2[0]);
        if (fd1[0] != STDIN_FILENO) {
            if (dup2(fd1[0], STDIN_FILENO) != STDIN_FILENO)
                err_sys("dup2 error to stdin");
            close(fd1[0]);
        }

        if (fd2[1] != STDOUT_FILENO) {
            if (dup2(fd2[1], STDOUT_FILENO) != STDOUT_FILENO)
                err_sys("dup2 error to stdout");
            close(fd2[1]);
        }
        if (execl("./add2", "add2", (char *)0) < 0)
            err_sys("execl error");
    }
    exit(0);
}

static void
sig_pipe(int signo)
{
    printf("SIGPIPE caught\n");
```

```
    exit(1);
}
```

- 标准I/O的默认缓冲机制问题
- 对标准输入的第一个fgets引起标准I/O库分配一个缓冲区，并选择缓冲的类型，因为标准输入是一个管道，所以标准I/O库默认是全缓冲的，标准输出也是。当add2从其标准输入读取而发生阻塞时，从管道读时也发生阻塞，于是产生了死锁
- setvbuf 解决fget　　pritnf的问题

15.5 FIFO

- 命名管道，通过FIFO,不相关的进程也能交换数据
- 创建FIFO类似于创建文件，FIFO是一种文件类型，FIFO的路径名存在于文件系统中
- mkfifo
- mkfifoat
- 可以用一条shell命令mkfifo创建一个FIFO,然后用一般的shell I/O重定向对其进行访问
- 用open打开FIFO
- 一个给定的FIFO有多个写进程是常见的
- FIFO用途
  - shell命令使用FIFO将数据从一条管道传送到另一条管道时，无需创建中间临时文件
  - 客户进程-服务器进程应用程序中，FIFO用作汇聚点
- tee命令程序将其标准输入同时复制到其标准输出以及其命令行中命名的文件中
- ```
  mkfifo fifo1
  prog3 < fifo1 &
  prog1 < infile | tee fifo1 | prog2
  #使用FIFO和tee将一个流发送到两个不同的进程
  ```

15.6 XSI IPC

- 三种: 消息队列　信号量　共享存储器
- 标识符和键
  - 标识符是IPC对象的内部名，键为外部名
  - ftok由一个路径名和项目ID产生一个键
- 权限结构
  - ipc_perm
- 结构限制
  - ipcs -l
- 优点和缺点
  - XSI IPC的IPC结构是在系统范围内起作用的，没有引用计数
  - IPC结构在文件系统中没有名字
  - 不使用文件描述符
  - 优点: 可靠的、流控制的、面向记录的，可以以用非先进先出次序处理
  ```

15.7 消息队列

- 消息队列是消息的链接表，存储在内核中，由消息队列标识符标识
- 每个队列都有一个msqid_ds结构与其相关联
- msgget，打开一个现有队列或创建一个新队列
- msgct1函数对队列执行多种操作
- msgsnd将数据放到消息队列中
- 消息队列没有维护引用计数器
- msgrcv从队列中取消息
- 考虑到使用消息队列时遇到的问题，在新的应用程序中不应当再使用它们

15.8 信号量

- 信号量与之前的IPC结构不同，它是一个计数器，用于为多个进程提供对共享数据对象的访问，信号量通常是在内核中实现的
- 内核为每个信号量维护着一个semid_ds结构
- semget获得一个信号量ID
- semctl函数包含了多种信号量操作
- 函数semop自动执行信号量集合上的操作数组
- 信号量、记录锁和互斥量的时间比较
- 在多个进程间共享的内存使用互斥量来恢复一个终止的进程更难

15.9 共享存储

- 共享存储允许多个进程共享一个给定的存储区，这是最快的一种IPC
- 在多个进程之间同步访问一个给定的存储区
- 信号量用于同步共享存储访问(可以用记录锁或互斥量)
- XSI共享存储和内存映射的文件的不同之处在于，它没有相关的文件，XSI共享存储段是内存的匿名段
- 内核为每个共享存储段维护着一个结构
- 函数shmget获得一个共享存储标识符
- shmctl函数对共享存储段执行多种操作
- 一旦创建了一个共享存储段，进程就可调用shmat将其连接到进程的地址空间中
- shmdt使相关的shmid_ds结构中的shm_nattch计数器值减1
- ```
  #include "apue.h"
  #include <sys/shm.h>

  #define ARRAY_SIZE  40000
  #define MALLOC_SIZE 100000
  #define SHM_SIZE    100000
  #define SHM_MODE    0600    /* user read/write */

  char    array[ARRAY_SIZE];  /* uninitialized data = bss */

  int
  main(void)
  {
      int     shmid;
  ```
  ```

```
    char    *ptr, *shmptr;

    printf("array[] from %p to %p\n", (void *)&array[0],
      (void *)&array[ARRAY_SIZE]);
    printf("stack around %p\n", (void *)&shmid);

    if ((ptr = malloc(MALLOC_SIZE)) == NULL)
        err_sys("malloc error");
    printf("malloced from %p to %p\n", (void *)ptr,
      (void *)ptr+MALLOC_SIZE);

    if ((shmid = shmget(IPC_PRIVATE, SHM_SIZE, SHM_MODE)) < 0)
        err_sys("shmget error");
    if ((shmptr = shmat(shmid, 0, 0)) == (void *)-1)
        err_sys("shmat error");
    printf("shared memory attached from %p to %p\n", (void *)shmptr,
      (void *)shmptr+SHM_SIZE);

    if (shmctl(shmid, IPC_RMID, 0) < 0)
        err_sys("shmctl error");

    exit(0);
}
```

- 图7-6典型的进程存储空间安排，图15-32进程存储空间布局
- 用mmap映射的存储段是与文件相关联的，而XSI共享存储段则并无这种关联
- 共享存储可由两个不相关的进程使用
- 匿名存储映射: 类似与/dev/zero的设施，在调用mmap时指定MAP_ANON标志，并将文件描述符指定为-1
- 如果在两个无关进程之间要使用共享存储段，有两种替代方法: 一种是应用程序使用XSI共享存储函数，另一种是使用mmap将同一文件映射至它们的地址空间

15.10 POSIX信号量

- POSIX信号量接口意在解决XSI信号量接口的几个缺陷
- 命名的: 可以通过名字访问
- 未命名的: 只存在于内存中
- sem_open创建一个新的命名信号量或使用一个现有的信号量
- 可以调用sem_close函数来释放任何信号量相关的资源
- sem_unlink函数销毁一个命名信号量
- sem_wait或sem_trywait函数实现信号量减1操作
- sem_timewait
- sem_post函数使信号量值增1
- 调用sem_init函数创建一个未命名的信号量
- 调用sem_destroy丢弃它
- sem_getvalue函数可以用来检索信号量值

- 二进制信号量可以像互斥量一样使用，可以使用信号量来创建自己的锁原语从而提供互斥

- ```
  #include <semaphore.h>
  #include <fcntl.h>
  #include <limits.h>
  #include <sys/stat.h>

  struct slock {
      sem_t *semp;
      char name[_POSIX_NAME_MAX];
  };

  struct slock * s_alloc();
  void s_free(struct slock *);
  int s_lock(struct slock *);
  int s_trylock(struct slock *);
  int s_unlock(struct slock *);
  ```

- ```
  #include "slock.h"
  #include <stdlib.h>
  #include <stdio.h>
  #include <unistd.h>
  #include <errno.h>

  struct slock *
  s_alloc()
  {
      struct slock *sp;
      static int cnt;

      if ((sp = malloc(sizeof(struct slock))) == NULL)
          return(NULL);
      do {
          snprintf(sp->name, sizeof(sp->name), "/%ld.%d", (long)getpid(),
            cnt++);
          sp->semp = sem_open(sp->name, O_CREAT|O_EXCL, S_IRWXU, 1);
      } while ((sp->semp == SEM_FAILED) && (errno == EEXIST));
      if (sp->semp == SEM_FAILED) {
          free(sp);
          return(NULL);
      }
      sem_unlink(sp->name);
      return(sp);
  }

  void
  s_free(struct slock *sp)
  {
      sem_close(sp->semp);
      free(sp);
  }

  int
  ```

```
s_lock(struct slock *sp)
{
    return(sem_wait(sp->semp));
}

int
s_trylock(struct slock *sp)
{
    return(sem_trywait(sp->semp));
}

int
s_unlock(struct slock *sp)
{
    return(sem_post(sp->semp));
}
```

15.11 客户进程-服务器进程属性

15.12

- 进程间通信的多种形式: 管道　命名管道(FIFO)　XSI IPC的3种形式的IPC(消息队列、信号量和共享存储)
- 信号量实际上是同步原语而不是IPC,常用于共享资源(如共享存储段)的同步访问
- 尽量使用管道和FIFO以及全双工管道和记录锁，避免使用消息队列和信号量

第16章　网络IPC: 套接字

16.2 套接字描述符

- socket
- shutdown

16.3 寻址

- 字节序
- 地址格式
- 地址查询
    - gethostent
    - sethostent
    - endhostent
- 将套接字与地址关联

16.4 建立连接

16.5 数据传输

- send,使用send时套接字必须已经连接
- sendto
- 调用带有msghdr结构的sendmsg来指定多重缓冲区传输数据

- recv,可以指定标志来控制如何接收数据
- recvfrom
- recvmsg用msghdr结构指定接收数据的输入缓冲区

16.6 套接字选项

16.7 带外数据

- TCP支持带外数据，UDP不支持
- TCP将带外数据称为紧急数据，仅支持一个字节的紧急数据，但是允许紧急数据在普通数据传递机制数据流之外传输

16.8 非阻塞和异步I/O

- 套接字非阻塞模式
- 基于套接字的异步I/O,当从套接字中读取数据时，或者当套接字写队列中空间变得可用时，可以安排要发送的信号SIGIO
- 采用fcntl和ioctl完成上述步骤

第17章高级进程间通信

- UNIX域套接字机制，这种形式的IPC可以在同一台计算机系统上运行的两个进程之间传送打开的文件描述符

17.2 UNIX域套接字

- UNIX域套接字用于同一台计算机上运行的进程之间的通信，因特网域套接字可用于同一目的，但UNIX域套接字效率更高
- 流
- 数据报，UNIX域数据报服务是可靠的
- UNIX域套接字就像是套接字和管道的混合，可以使用它们面向网络的域套接字接口
- 使用socketpair函数创建一对无名的、相互连接的UNIX域套接字，一对相互连接的UNIX域套接字可以起到全双工管道的作用，称为fd管道
- 借助UNIX域套接字轮训XSI消息队列,对每个消息队列使用一个线程，每个线程都会在msgrcv调用中阻塞，当消息到达时，线程会把它写入一个UNIX域套接字的一端，当poll指示套接字可以读取数据时，应用程序会使用这个套接字的另外一端来接收这个消息
- ```
  #include "apue.h"
  #include <poll.h>
  #include <pthread.h>
  #include <sys/msg.h>
  #include <sys/socket.h>

  #define NQ      3       /* number of queues */
  #define MAXMSZ  512     /* maximum message size */
  #define KEY     0x123   /* key for first message queue */

  struct threadinfo {
      int qid;
  ```
  ```

```
    int fd;
};

struct mymesg {
    long mtype;
    char mtext[MAXMSZ];
};

void *
helper(void *arg)
{
    int                 n;
    struct mymesg       m;
    struct threadinfo   *tip = arg;

    for(;;) {
        memset(&m, 0, sizeof(m));
        if ((n = msgrcv(tip->qid, &m, MAXMSZ, 0, MSG_NOERROR)) < 0)
            err_sys("msgrcv error");
        if (write(tip->fd, m.mtext, n) < 0)
            err_sys("write error");
    }
}

int
main()
{
    int                 i, n, err;
    int                 fd[2];
    int                 qid[NQ];
    struct pollfd       pfd[NQ];
    struct threadinfo   ti[NQ];
    pthread_t           tid[NQ];
    char                buf[MAXMSZ];

    for (i = 0; i < NQ; i++) {
        if ((qid[i] = msgget((KEY+i), IPC_CREAT|0666)) < 0)
            err_sys("msgget error");

        printf("queue ID %d is %d\n", i, qid[i]);

        if (socketpair(AF_UNIX, SOCK_DGRAM, 0, fd) < 0)
            err_sys("socketpair error");
        pfd[i].fd = fd[0];
        pfd[i].events = POLLIN;
        ti[i].qid = qid[i];
        ti[i].fd = fd[1];
        if ((err = pthread_create(&tid[i], NULL, helper, &ti[i])) != 0)
            err_exit(err, "pthread_create error");
    }

    for (;;) {
        if (poll(pfd, NQ, -1) < 0)
```

```
            err_sys("poll error");
        for (i = 0; i < NQ; i++) {
            if (pfd[i].revents & POLLIN) {
                if ((n = read(pfd[i].fd, buf, sizeof(buf))) < 0)
                    err_sys("read error");
                buf[n] = 0;
                printf("queue id %d, message %s\n", qid[i], buf);
            }
        }
    }

    exit(0);
}
```

- 使用数据报套接字可以保证消息边界
- 给XSI消息队列发送消息
- ```
  #include "apue.h"
  #include <sys/msg.h>

  #define MAXMSZ 512

  struct mymesg {
      long mtype;
      char mtext[MAXMSZ];
  };

  int
  main(int argc, char *argv[])
  {
      key_t key;
      long qid;
      size_t nbytes;
      struct mymesg m;

      if (argc != 3) {
          fprintf(stderr, "usage: sendmsg KEY message\n");
          exit(1);
      }
      key = strtol(argv[1], NULL, 0);
      if ((qid = msgget(key, 0)) < 0)
          err_sys("can't open queue key %s", argv[1]);
      memset(&m, 0, sizeof(m));
      strncpy(m.mtext, argv[2], MAXMSZ-1);
      nbytes = strlen(m.mtext);
      m.mtype = 1;
      if (msgsnd(qid, &m, nbytes, 0) < 0)
          err_sys("can't send message");
      exit(0);
  }
  ```
  ```

```
./pollmsg &
./sendmsg 0x123 "hello world"
```

- 命名UNIX域套接字
  - UNIX域套接字使用的地址由sockaddr_un结构表示
  - ```
    struct sockaddr_un{
        sa_family_t sun_family;
        char sun_path[108];
    }
    ```
  - 当将一个地址绑定到一个UNIX域套接字时，系统会用该路径名创建一个S_IFSOCK类型的文件
  - ```
    #include "apue.h"
    #include <sys/socket.h>
    #include <sys/un.h>

    int
    main(void)
    {
        int fd, size;
        struct sockaddr_un un;

        un.sun_family = AF_UNIX;
        strcpy(un.sun_path, "foo.socket");
        if ((fd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
            err_sys("socket failed");
        size = offsetof(struct sockaddr_un, sun_path) + strlen(un.sun_path);
        if (bind(fd, (struct sockaddr *)&un, size) < 0)
            err_sys("bind failed");
        printf("UNIX domain socket bound\n");
        exit(0);
    }
    ```

17.3 唯一连接

- 服务器进程可以使用标准bind listen 和accept函数，为客户进程安排一个唯一UNIX域连接，客户进程使用connect与服务器进程联系，在服务器进程接受了connect请求后，在服务器进程和客户进程之间就存在了唯一连接
- serv_listen函数
- serv_accept函数
- cli_conn函数

17.4 传送文件描述符

- 传送一个打开的文件描述符，想让发送进程和接收进程共享同一文件表项
- 将指向一个打开文件表项的指针从一个进程发送到另外一个进程，该指针被分配存在在接收进程的第一个可用描述符中，两个进程共享同一个打开文件表
- send_fd

- send_err
- recv_fd
- 为了用UNIX域套接字交换文件描述符，调用sendmsg和recvmsg函数，通过一个指向msghdr结构的指针，msg_control字段指向cmsghdr结构
- 通过UNIX域套接字发送证书，将证书作为cmsgcred或ucred结构发送
- ```
  #include "apue.h"
  #include <sys/socket.h>

  #if defined(SCM_CREDS)          /* BSD interface */
  #define CREDSTRUCT      cmsgcred
  #define SCM_CREDTYPE    SCM_CREDS
  #elif defined(SCM_CREDENTIALS)  /* Linux interface */
  #define CREDSTRUCT      ucred
  #define SCM_CREDTYPE    SCM_CREDENTIALS
  #else
  #error passing credentials is unsupported!
  #endif

  /* size of control buffer to send/recv one file descriptor */
  #define RIGHTSLEN   CMSG_LEN(sizeof(int))
  #define CREDSLEN    CMSG_LEN(sizeof(struct CREDSTRUCT))
  #define CONTROLLEN  (RIGHTSLEN + CREDSLEN)

  static struct cmsghdr   *cmptr = NULL;  /* malloc'ed first time */

  /*
   * Pass a file descriptor to another process.
   * If fd<0, then -fd is sent back instead as the error status.
   */
  int
  send_fd(int fd, int fd_to_send)
  {
      struct CREDSTRUCT   *credp;
      struct cmsghdr      *cmp;
      struct iovec        iov[1];
      struct msghdr       msg;
      char                buf[2]; /* send_fd/recv_ufd 2-byte protocol */

      iov[0].iov_base = buf;
      iov[0].iov_len  = 2;
      msg.msg_iov     = iov;
      msg.msg_iovlen  = 1;
      msg.msg_name    = NULL;
      msg.msg_namelen = 0;
      msg.msg_flags = 0;
      if (fd_to_send < 0) {
          msg.msg_control    = NULL;
          msg.msg_controllen = 0;
          buf[1] = -fd_to_send;   /* nonzero status means error */
          if (buf[1] == 0)
              buf[1] = 1; /* -256, etc. would screw up protocol */
  ```
  ```

```
    } else {
        if (cmptr == NULL && (cmptr = malloc(CONTROLLEN)) == NULL)
            return(-1);
        msg.msg_control    = cmptr;
        msg.msg_controllen = CONTROLLEN;
        cmp = cmptr;
        cmp->cmsg_level  = SOL_SOCKET;
        cmp->cmsg_type   = SCM_RIGHTS;
        cmp->cmsg_len    = RIGHTSLEN;
        *(int *)CMSG_DATA(cmp) = fd_to_send;    /* the fd to pass */
        cmp = CMSG_NXTHDR(&msg, cmp);
        cmp->cmsg_level  = SOL_SOCKET;
        cmp->cmsg_type   = SCM_CREDTYPE;
        cmp->cmsg_len    = CREDSLEN;
        credp = (struct CREDSTRUCT *)CMSG_DATA(cmp);
#if defined(SCM_CREDENTIALS)
        credp->uid = geteuid();
        credp->gid = getegid();
        credp->pid = getpid();
#endif
        buf[1] = 0;     /* zero status means OK */
    }
    buf[0] = 0;         /* null byte flag to recv_ufd() */
    if (sendmsg(fd, &msg, 0) != 2)
        return(-1);
    return(0);
}
```

- ```
  #include "apue.h"
  #include <sys/socket.h>     /* struct msghdr */
  #include <sys/un.h>

  #if defined(SCM_CREDS)          /* BSD interface */
  #define CREDSTRUCT      cmsgcred
  #define CR_UID          cmcred_uid
  #define SCM_CREDTYPE    SCM_CREDS
  #elif defined(SCM_CREDENTIALS)  /* Linux interface */
  #define CREDSTRUCT      ucred
  #define CR_UID          uid
  #define CREDOPT         SO_PASSCRED
  #define SCM_CREDTYPE    SCM_CREDENTIALS
  #else
  #error passing credentials is unsupported!
  #endif

  /* size of control buffer to send/recv one file descriptor */
  #define RIGHTSLEN   CMSG_LEN(sizeof(int))
  #define CREDSLEN    CMSG_LEN(sizeof(struct CREDSTRUCT))
  #define CONTROLLEN  (RIGHTSLEN + CREDSLEN)

  static struct cmsghdr   *cmptr = NULL;      /* malloc'ed first time */

  /*
  ```
  ```

```
 * Receive a file descriptor from a server process.  Also, any data
 * received is passed to (*userfunc)(STDERR_FILENO, buf, nbytes).
 * We have a 2-byte protocol for receiving the fd from send_fd().
 */
int
recv_ufd(int fd, uid_t *uidptr,
         ssize_t (*userfunc)(int, const void *, size_t))
{
    struct cmsghdr      *cmp;
    struct CREDSTRUCT   *credp;
    char                *ptr;
    char                buf[MAXLINE];
    struct iovec        iov[1];
    struct msghdr       msg;
    int                 nr;
    int                 newfd = -1;
    int                 status = -1;
#if defined(CREDOPT)
    const int           on = 1;

    if (setsockopt(fd, SOL_SOCKET, CREDOPT, &on, sizeof(int)) < 0) {
        err_ret("setsockopt error");
        return(-1);
    }
#endif
    for ( ; ; ) {
        iov[0].iov_base = buf;
        iov[0].iov_len  = sizeof(buf);
        msg.msg_iov     = iov;
        msg.msg_iovlen  = 1;
        msg.msg_name    = NULL;
        msg.msg_namelen = 0;
        if (cmptr == NULL && (cmptr = malloc(CONTROLLEN)) == NULL)
            return(-1);
        msg.msg_control    = cmptr;
        msg.msg_controllen = CONTROLLEN;
        if ((nr = recvmsg(fd, &msg, 0)) < 0) {
            err_ret("recvmsg error");
            return(-1);
        } else if (nr == 0) {
            err_ret("connection closed by server");
            return(-1);
        }

        /*
         * See if this is the final data with null & status.  Null
         * is next to last byte of buffer; status byte is last byte.
         * Zero status means there is a file descriptor to receive.
         */
        for (ptr = buf; ptr < &buf[nr]; ) {
            if (*ptr++ == 0) {
                if (ptr != &buf[nr-1])
                    err_dump("message format error");
```

```
                status = *ptr & 0xFF;   /* prevent sign extension */
                if (status == 0) {
                    if (msg.msg_controllen != CONTROLLEN)
                        err_dump("status = 0 but no fd");

                    /* process the control data */
                    for (cmp = CMSG_FIRSTHDR(&msg);
                      cmp != NULL; cmp = CMSG_NXTHDR(&msg, cmp)) {
                        if (cmp->cmsg_level != SOL_SOCKET)
                            continue;
                        switch (cmp->cmsg_type) {
                        case SCM_RIGHTS:
                            newfd = *(int *)CMSG_DATA(cmp);
                            break;
                        case SCM_CREDTYPE:
                            credp = (struct CREDSTRUCT *)CMSG_DATA(cmp);
                            *uidptr = credp->CR_UID;
                        }
                    }
                } else {
                    newfd = -status;
                }
                nr -= 2;
            }
        }
        if (nr > 0 && (*userfunc)(STDERR_FILENO, buf, nr) != nr)
            return(-1);
        if (status >= 0)    /* final data has arrived */
            return(newfd);  /* descriptor, or -status */
    }
}
```

17.5 open服务器进程第一版

- 从客户进程到服务器进程传送文件名和打开模式，而从服务进程到客户进程返回描述符，文件内容不需要通过IPC交换

17.6 open服务器进程第2版

- 一个守护进程方式的open服务器进程，一个服务器进程处理所有客户进程的请求
- getopt

第18章　终端I/O

18.2 综述

- 终端I/O的两种工作模式
  - 规范模式输入处理，以行为单位
  - 非规范模式输入处理

- 终端设备是由通常位于内核中的终端驱动程序控制的，每个终端设备都有一个输入队列和输出队列
- 终端行规程中进行规范处理，位于内核读写函数和实际设备驱动程序之间
- 所有可以检测和更改的终端设备特性都包含在termios结构中,终端标志
- ```
  struct termios{
      tcflag_t c_iflag;
      tcflag_t c_oflag;
      tcflag_t c_cflag;
      tcflag_t c_lflag;
      cc_t c_cc[NCCS];
  };
  ```
- 13个终端I/O函数
  - tcgetattr
  - tcsetattr
  - cfgetispeed
  - cfgetospeed
  - cfsetispeed
  - cfsetospeed
  - tcdrain
  - tcflow
  - tcflush
  - tcsendbreak
  - tcgetpgrp
  - tcsetprgp
  - tcgetsid

18.3 特殊输入字符

- 11个在输入时要特殊处理的字符
- 将c_cc数组中的某项设置为_POSIX_VDISABLE的值，则禁止使用相应特殊字符
- ```
  #include "apue.h"
  #include <termios.h>

  int
  main(void)
  {
      struct termios  term;
      long            vdisable;

      if (isatty(STDIN_FILENO) == 0)
          err_quit("standard input is not a terminal device");

      if ((vdisable = fpathconf(STDIN_FILENO, _PC_VDISABLE)) < 0)
          err_quit("fpathconf error or _POSIX_VDISABLE not in effect");

      if (tcgetattr(STDIN_FILENO, &term) < 0) /* fetch tty state */
          err_sys("tcgetattr error");
  ```

```
    term.c_cc[VINTR] = vdisable;    /* disable INTR character */
    term.c_cc[VEOF]  = 2;           /* EOF is Control-B */

    if (tcsetattr(STDIN_FILENO, TCSAFLUSH, &term) < 0)
        err_sys("tcsetattr error");

    exit(0);
}
```

- CR
- DISCARD
- DSUSP
- EOF
- EOL
- EOL2
- ERASE
- ERASE2
- INTR
- KILL
- LNEXT
- NL
- QUIT
- REPRINT
- START
- STATUS
- STOP
- SUSP
- WERSAE
- BREAK 异步串行数据传送时发生的一个条件

18.4 获得和设置终端属性

- tcgetattr
- tcsetattr
- 这两个函数只对终端设备进行操作

18.5 终端选项标志

- ```
  #include "apue.h"
  #include <termios.h>

  int
  ```
  ```

```c
main(void)
{
    struct termios  term;

    if (tcgetattr(STDIN_FILENO, &term) < 0)
        err_sys("tcgetattr error");

    switch (term.c_cflag & CSIZE) {
    case CS5:
        printf("5 bits/byte\n");
        break;
    case CS6:
        printf("6 bits/byte\n");
        break;
    case CS7:
        printf("7 bits/byte\n");
        break;
    case CS8:
        printf("8 bits/byte\n");
        break;
    default:
        printf("unknown bits/byte\n");
    }

    term.c_cflag &= ~CSIZE;     /* zero out the bits */
    term.c_cflag |= CS8;        /* set 8 bits/byte */
    if (tcsetattr(STDIN_FILENO, TCSANOW, &term) < 0)
        err_sys("tcsetattr error");

    exit(0);
}
```

- 各选项标志

18.6 stty命令

- 在程序中使用tcgetattr和tcsetattr函数进行选项的检查和更改
- 在命令行中使用stty命令进行检查和更改
- stty -a 显示终端的所有选项
- stty命令使用它的标准输入获得和设置终端的选项标志
- 如果希望了解名为ttyla的终端的设置，可以键入　stty -a</dev/ttyla

18.7 波特率函数

- 位/秒
- cfgetispeed
- cfgetospedd
- cfsetispeed
- cfsetospeed
- 输入、输出波特率是存储在设备的termio结构中的

18.8 行控制函数

- tcdrain
- tcflow
- tcflush
- tcsendbreak

18.9 终端标识

- 大多数UNIX系统的控制终端的名字一直是/dev/tty
- ctermid 函数确定控制终端名
- ```
  #include    <stdio.h>
  #include    <string.h>

  static char ctermid_name[L_ctermid];

  char *
  ctermid(char *str)
  {
      if (str == NULL)
          str = ctermid_name;
      return(strcpy(str, "/dev/tty"));    /* strcpy() returns str */
  }
  ```
- 如果文件描述符引用一个终端设备，则isatty返回真
- ```
  #include    <termios.h>

  int
  isatty(int fd)
  {
      struct termios  ts;

      return(tcgetattr(fd, &ts) != -1); /* true if no error (is a tty) */
  }
  ```
- ttyname 返回的是在该描述符上打开的终端设备的路径名
- ```
  #include    <sys/stat.h>
  #include    <dirent.h>
  #include    <limits.h>
  #include    <string.h>
  #include    <termios.h>
  #include    <unistd.h>
  #include    <stdlib.h>

  struct devdir {
      struct devdir   *d_next;
      char            *d_name;
  };

  static struct devdir    *head;
  static struct devdir    *tail;
  ```
  ```

```
static char             pathname[_POSIX_PATH_MAX + 1];

static void
add(char *dirname)
{
    struct devdir   *ddp;
    int             len;

    len = strlen(dirname);

    /*
     * Skip ., .., and /dev/fd.
     */
    if ((dirname[len-1] == '.') && (dirname[len-2] == '/' ||
      (dirname[len-2] == '.' && dirname[len-3] == '/')))
        return;
    if (strcmp(dirname, "/dev/fd") == 0)
        return;
    if ((ddp = malloc(sizeof(struct devdir))) == NULL)
        return;

    if ((ddp->d_name = strdup(dirname)) == NULL) {
        free(ddp);
        return;
    }

    ddp->d_next = NULL;
    if (tail == NULL) {
        head = ddp;
        tail = ddp;
    } else {
        tail->d_next = ddp;
        tail = ddp;
    }
}

static void
cleanup(void)
{
    struct devdir   *ddp, *nddp;

    ddp = head;
    while (ddp != NULL) {
        nddp = ddp->d_next;
        free(ddp->d_name);
        free(ddp);
        ddp = nddp;
    }
    head = NULL;
    tail = NULL;
}

static char *
```

```
searchdir(char *dirname, struct stat *fdstatp)
{
    struct stat     devstat;
    DIR             *dp;
    int             devlen;
    struct dirent   *dirp;

    strcpy(pathname, dirname);
    if ((dp = opendir(dirname)) == NULL)
        return(NULL);
    strcat(pathname, "/");
    devlen = strlen(pathname);
    while ((dirp = readdir(dp)) != NULL) {
        strncpy(pathname + devlen, dirp->d_name,
          _POSIX_PATH_MAX - devlen);

        /*
         * Skip aliases.
         */
        if (strcmp(pathname, "/dev/stdin") == 0 ||
          strcmp(pathname, "/dev/stdout") == 0 ||
          strcmp(pathname, "/dev/stderr") == 0)
            continue;
        if (stat(pathname, &devstat) < 0)
            continue;
        if (S_ISDIR(devstat.st_mode)) {
            add(pathname);
            continue;
        }
        if (devstat.st_ino == fdstatp->st_ino &&
          devstat.st_dev == fdstatp->st_dev) {  /* found a match */
            closedir(dp);
            return(pathname);
        }
    }

    closedir(dp);
    return(NULL);
}

char *
ttyname(int fd)
{
    struct stat     fdstat;
    struct devdir   *ddp;
    char            *rval;

    if (isatty(fd) == 0)
        return(NULL);
    if (fstat(fd, &fdstat) < 0)
        return(NULL);
    if (S_ISCHR(fdstat.st_mode) == 0)
        return(NULL);
```

```
    rval = searchdir("/dev", &fdstat);
    if (rval == NULL) {
        for (ddp = head; ddp != NULL; ddp = ddp->d_next)
            if ((rval = searchdir(ddp->d_name, &fdstat)) != NULL)
                break;
    }

    cleanup();
    return(rval);
}
```

- 每个文件设备都有一个唯一的设备号，每个目录项都有一个唯一的i节点编号
- ```
  #include "apue.h"

  int
  main(void)
  {
      char *name;
      if (isatty(0)) {
          name = ttyname(0);
          if (name == NULL)
              name = "undefined";
      } else {
          name = "not a tty";
      }
      printf("fd 0: %s\n", name);

      if (isatty(1)) {
          name = ttyname(1);
          if (name == NULL)
              name = "undefined";
      } else {
          name = "not a tty";
      }
      printf("fd 1: %s\n", name);

      if (isatty(2)) {
          name = ttyname(2);
          if (name == NULL)
              name = "undefined";
      } else {
          name = "not a tty";
      }
      printf("fd 2: %s\n", name);

      exit(0);
  }
  ```

18.10 规范模式

- 发一个读请求，当一行已经输入后，终端驱动程序即返回
  ```

- getpass函数，为了读取口令，该函数必须关闭回显，但仍可使终端以规范模式进行工作
- ```
#include   <signal.h>
#include   <stdio.h>
#include   <termios.h>

#define MAX_PASS_LEN   8   /* max #chars for user to enter */

char *
getpass(const char *prompt)
{
    static char   buf[MAX_PASS_LEN + 1]; /* null byte at end */
    char   *ptr;
    sigset_t   sig, osig;
    struct termios ts, ots;
    FILE   *fp;
    int   c;

    if ((fp = fopen(ctermid(NULL), "r+")) == NULL)
        return(NULL);
    setbuf(fp, NULL);

    sigemptyset(&sig);
    sigaddset(&sig, SIGINT);   /* block SIGINT */
    sigaddset(&sig, SIGTSTP);   /* block SIGTSTP */
    sigprocmask(SIG_BLOCK, &sig, &osig);   /* and save mask */

    tcgetattr(fileno(fp), &ts);   /* save tty state */
    ots = ts;   /* structure copy */
    ts.c_lflag &= ~(ECHO | ECHOE | ECHOK | ECHONL);
    tcsetattr(fileno(fp), TCSAFLUSH, &ts);
    fputs(prompt, fp);

    ptr = buf;
    while ((c = getc(fp)) != EOF && c != '\n')
        if (ptr < &buf[MAX_PASS_LEN])
            *ptr++ = c;
    ptr = 0;   / null terminate */
    putc('\n', fp);   /* we echo a newline */

    tcsetattr(fileno(fp), TCSAFLUSH, &ots); /* restore TTY state */
    sigprocmask(SIG_SETMASK, &osig, NULL); /* restore mask */
    fclose(fp);   /* done with /dev/tty */
    return(buf);
}
```

## 18.11 非规范模式

- 可以通过关闭termios结构中的c_lflag字段的ICANON标志来指定非规范模式，在非规范模式下，输入数据不装配成行，不处理下列特殊字符:
- 当已经读了指定量的数据后，或者已经超过了给定量的时间后，即通知系统返回
- MIN, TIME
```

- 在要转入非规范模式时，将整个termios结构保存起来，以后再转回规范模式时恢复它
- cbreak模式
- 原始模式

18.12 终端窗口大小

- winsize结构
- 用ioctl的TIOCGWINSZ命令可以取此结构的当前值
- 用ioctl的TIOCSWINSZ命令可以将此结构的新值存储到内核
- ```
  #include "apue.h"
  #include <termios.h>
  #ifndef TIOCGWINSZ
  #include <sys/ioctl.h>
  #endif

  static void
  pr_winsize(int fd)
  {
      struct winsize  size;

      if (ioctl(fd, TIOCGWINSZ, (char *) &size) < 0)
          err_sys("TIOCGWINSZ error");
      printf("%d rows, %d columns\n", size.ws_row, size.ws_col);
  }

  static void
  sig_winch(int signo)
  {
      printf("SIGWINCH received\n");
      pr_winsize(STDIN_FILENO);
  }

  int
  main(void)
  {
      if (isatty(STDIN_FILENO) == 0)
          exit(1);
      if (signal(SIGWINCH, sig_winch) == SIG_ERR)
          err_sys("signal error");
      pr_winsize(STDIN_FILENO);   /* print initial size */
      for ( ; ; )                 /* and sleep forever */
          pause();
  }
  ```

18.13 termcap terminfo curses

第19章　伪终端
  ```

19.2 概述

- 伪终端: 对一个应用程序而言，它看上去像一个终端，但事实上它不是一个真正的终端
- 使用伪终端的相关进程的典型结构
- 网络登录服务器，伪终端可用于构造提供网络登录的服务器
- 窗口系统终端模拟，终端模拟器作为shell和窗口管理器之间的媒介，每个shell在自己的窗口中执行
- script程序
- expect程序
- 运行协同程序，将一个伪终端放到两个进程之间，诱使协同进程认为它是由终端驱动的，而非另一个进程
- 观看长时间运行程序的输出，在pty下运行该程序，让标准I/O库认为标准输出是终端

19.3 打开伪终端设备

- posix_openpt
- grantpt
- unlockpt
- ptsname函数找到伪终端从设备的路径名
- ptym_open
- ptys_open

19.4 函数pty_fork

- 用fork调用打开主设备和从设备，创建作为会话首进程的子进程并使其具有控制终端

19.5 pty程序

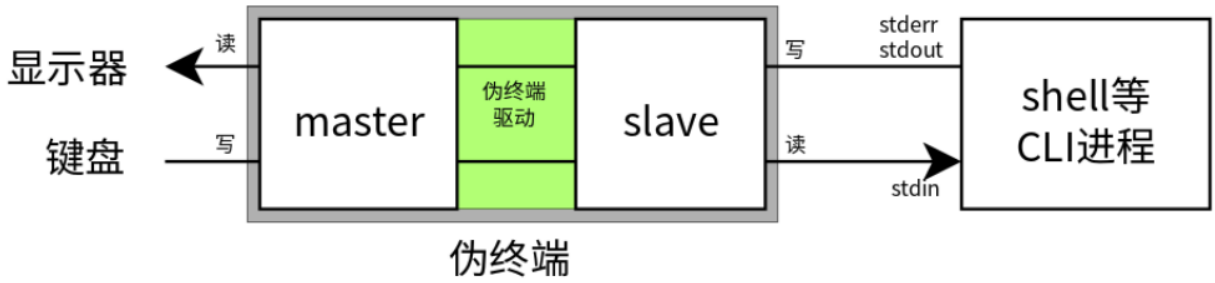
- 当用pty来执行另一个程序时，那个程序在一个它自己的会话中执行，并和一个伪终端连接
- loop函数仅仅是将从标准输入接收到的所有内容复制到PTY主设备，并将PTY主设备接收到的所有内容复制到标准输出

19.6 使用pty程序

- 一般使用伪终端的程序将不能对utmp文件进行写操作
- 作业控制交互

19.7 高级特性

- https://www.cnblogs.com/zzdyyy/p/7538077.html
- `/dev/pts/` 文件夹里的以数字命名的文件就是伪终端的设备文件。
- linux中的每个进程有一个“控制终端（control terminal）”的属性（取值为设备文件），用于实现作业控制。在终端上输入Ctrl+C、Ctrl+Z，则以该终端为控制终端的前台进程组会收到终止、暂停的信号。
- 终端模拟器是应用程序，用于模拟一个终端。它一般是GUI程序，带有窗口。从窗口输入的字符作为模拟键盘的输入，在窗口上打印的字符作为模拟显示器的输出。终端模拟器还需要创建模拟的终端设备（如`/dev/pts/1`），用于当做命令行进程（CLI进程）的输入输出、控制终端。当键盘键入一个字符，它要让CLI进程从终端设备中读到这个字符，当CLI进程写入终端设备时，终端模拟器要读到并显示出来
- 伪终端是伪终端master和伪终端slave（终端设备文件）这一对字符设备。`/dev/ptmx`是用于创建一对master、slave的文件。当一个进程打开它时，获得了一个master的文件描述符（file descriptor），同时在`/dev/pts`下创建了一个slave设备文件。

- 

- 当终端模拟器运行时，它通过`/dev/ptmx`打开master端，创建了一个伪终端对，并让shell运行在slave端。当用户在终端模拟器中按下键盘按键时，它产生字节流并写入master中，shell便可从slave中读取输入；shell和它的子程序，将输出内容写入slave中，由终端模拟器负责将字符打印到窗口中。

第20章　数据库函数库

20.2 历史

- gdbm库
- 并发控制(记录锁机制)
- B+树　动态散列技术

20.3 函数库

- db_open
- db_close
- db_store
- db_fetch
- db_delete
- db_rewind回滚到数据库的第一条记录
- db_nextrec顺序地读每条记录

20.4 实现概述

- 索引文件包括实际的索引值和一个指向数据文件中对应记录的指针，可采用散列表和B+树实现
- 数据文件
- 按照字符串形式存储所有的记录，包含键和数据
- ```
  #include "apue.h"
  #include "apue_db.h"
  #include <fcntl.h>

  int
  main(void)
  {
      DBHANDLE    db;

      if ((db = db_open("db4", O_RDWR | O_CREAT | O_TRUNC,
        FILE_MODE)) == NULL)
          err_sys("db_open error");
  ```
  ```

```
    if (db_store(db, "Alpha", "data1", DB_INSERT) != 0)
        err_quit("db_store error for alpha");
    if (db_store(db, "beta", "Data for beta", DB_INSERT) != 0)
        err_quit("db_store error for beta");
    if (db_store(db, "gamma", "record3", DB_INSERT) != 0)
        err_quit("db_store error for gamma");

    db_close(db);
    exit(0);
}
```

20.5 集中式或非集中式

- 集中式，由一个进程作为数据库管理者，所有的数据库访问工作由此进程完成，其他进程通过IPC机制与此中心进程进行联系
- 非集中式，每个库函数使用要求的并发控制(加锁)，然后发起自己的I/O函数调用

20.6 并发

- 粗粒度锁
- 细粒度锁

20.7 构造函数库

- 构造一个静态函数库
  - gcc -I../include -Wall -c db.c
  - ar rsv libapue_db.a db.o
- 构造一个动态共享库版本
  - gcc -I../include -Wall -fPIC -c db.c
  - gcc -shared -Wl,-soname,libapue_db.so.1 -o libapue_db.so.1 -L../lib -lapue -lc db.o
  - 构建成的共享库libapue_db.so.1需要放置在动态连接程序/载入程序能够找到的一个公共目录中，还可以将共享库放在一个私有目录中，修改LD_LIBRARY_PATH环境变量，使动态连接程序/载入程序的搜索路径包含该私有目录

20.8 源代码

```
#include "apue.h"
#include "apue_db.h"
#include <fcntl.h>      /* open & db_open flags */
#include <stdarg.h>
#include <errno.h>
#include <sys/uio.h>    /* struct iovec */

/*
 * Internal index file constants.
 * These are used to construct records in the
 * index file and data file.
 */
#define IDXLEN_SZ      4    /* index record length (ASCII chars) */
```

```
#define SEP         ':' /* separator char in index record */
#define SPACE       ' ' /* space character */
#define NEWLINE     '\n'    /* newline character */

/*
 * The following definitions are for hash chains and free
 * list chain in the index file.
 */
#define PTR_SZ        7 /* size of ptr field in hash chain */
#define PTR_MAX 9999999 /* max file offset = 10**PTR_SZ - 1 */
#define NHASH_DEF    137    /* default hash table size */
#define FREE_OFF      0 /* free list offset in index file */
#define HASH_OFF PTR_SZ /* hash table offset in index file */

typedef unsigned long   DBHASH; /* hash values */
typedef unsigned long   COUNT;  /* unsigned counter */

/*
 * Library's private representation of the database.
 */
typedef struct {
  int    idxfd;  /* fd for index file */
  int    datfd;  /* fd for data file */
  char  *idxbuf; /* malloc'ed buffer for index record */
  char  *datbuf; /* malloc'ed buffer for data record*/
  char  *name;   /* name db was opened under */
  off_t  idxoff; /* offset in index file of index record */
                 /* key is at (idxoff + PTR_SZ + IDXLEN_SZ) */
  size_t idxlen; /* length of index record */
                 /* excludes IDXLEN_SZ bytes at front of record */
                 /* includes newline at end of index record */
  off_t  datoff; /* offset in data file of data record */
  size_t datlen; /* length of data record */
                 /* includes newline at end */
  off_t  ptrval; /* contents of chain ptr in index record */
  off_t  ptroff; /* chain ptr offset pointing to this idx record */
  off_t  chainoff; /* offset of hash chain for this index record */
  off_t  hashoff;  /* offset in index file of hash table */
  DBHASH nhash;    /* current hash table size */
  COUNT  cnt_delok;    /* delete OK */
  COUNT  cnt_delerr;   /* delete error */
  COUNT  cnt_fetchok;  /* fetch OK */
  COUNT  cnt_fetcherr; /* fetch error */
  COUNT  cnt_nextrec;  /* nextrec */
  COUNT  cnt_stor1;    /* store: DB_INSERT, no empty, appended */
  COUNT  cnt_stor2;    /* store: DB_INSERT, found empty, reused */
  COUNT  cnt_stor3;    /* store: DB_REPLACE, diff len, appended */
  COUNT  cnt_stor4;    /* store: DB_REPLACE, same len, overwrote */
  COUNT  cnt_storerr;  /* store error */
} DB;

/*
 * Internal functions.
```

```
 */
static DB     *_db_alloc(int);
static void    _db_dodelete(DB *);
static int     _db_find_and_lock(DB *, const char *, int);
static int     _db_findfree(DB *, int, int);
static void    _db_free(DB *);
static DBHASH  _db_hash(DB *, const char *);
static char   *_db_readdat(DB *);
static off_t   _db_readidx(DB *, off_t);
static off_t   _db_readptr(DB *, off_t);
static void    _db_writedat(DB *, const char *, off_t, int);
static void    _db_writeidx(DB *, const char *, off_t, int, off_t);
static void    _db_writeptr(DB *, off_t, off_t);

/*
 * Open or create a database.  Same arguments as open(2).
 */
DBHANDLE
db_open(const char *pathname, int oflag, ...)
{
    DB          *db;
    int         len, mode;
    size_t      i;
    char        asciiptr[PTR_SZ + 1],
                hash[(NHASH_DEF + 1) * PTR_SZ + 2];
                    /* +2 for newline and null */
    struct stat statbuff;

    /*
     * Allocate a DB structure, and the buffers it needs.
     */
    len = strlen(pathname);
    if ((db = _db_alloc(len)) == NULL)
        err_dump("db_open: _db_alloc error for DB");

    db->nhash   = NHASH_DEF;/* hash table size */
    db->hashoff = HASH_OFF; /* offset in index file of hash table */
    strcpy(db->name, pathname);
    strcat(db->name, ".idx");

    if (oflag & O_CREAT) {
        va_list ap;

        va_start(ap, oflag);
        mode = va_arg(ap, int);
        va_end(ap);

        /*
         * Open index file and data file.
         */
        db->idxfd = open(db->name, oflag, mode);
        strcpy(db->name + len, ".dat");
        db->datfd = open(db->name, oflag, mode);
```

```
    } else {
        /*
         * Open index file and data file.
         */
        db->idxfd = open(db->name, oflag);
        strcpy(db->name + len, ".dat");
        db->datfd = open(db->name, oflag);
    }

    if (db->idxfd < 0 || db->datfd < 0) {
        _db_free(db);
        return(NULL);
    }

    if ((oflag & (O_CREAT | O_TRUNC)) == (O_CREAT | O_TRUNC)) {
        /*
         * If the database was created, we have to initialize
         * it.  Write lock the entire file so that we can stat
         * it, check its size, and initialize it, atomically.
         */
        if (writew_lock(db->idxfd, 0, SEEK_SET, 0) < 0)
            err_dump("db_open: writew_lock error");

        if (fstat(db->idxfd, &statbuff) < 0)
            err_sys("db_open: fstat error");

        if (statbuff.st_size == 0) {
            /*
             * We have to build a list of (NHASH_DEF + 1) chain
             * ptrs with a value of 0.  The +1 is for the free
             * list pointer that precedes the hash table.
             */
            sprintf(asciiptr, "%*d", PTR_SZ, 0);
            hash[0] = 0;
            for (i = 0; i < NHASH_DEF + 1; i++)
                strcat(hash, asciiptr);
            strcat(hash, "\n");
            i = strlen(hash);
            if (write(db->idxfd, hash, i) != i)
                err_dump("db_open: index file init write error");
        }
        if (un_lock(db->idxfd, 0, SEEK_SET, 0) < 0)
            err_dump("db_open: un_lock error");
    }
    db_rewind(db);
    return(db);
}

/*
 * Allocate & initialize a DB structure and its buffers.
 */
static DB *
_db_alloc(int namelen)
```

```
{
    DB      *db;

    /*
     * Use calloc, to initialize the structure to zero.
     */
    if ((db = calloc(1, sizeof(DB))) == NULL)
        err_dump("_db_alloc: calloc error for DB");
    db->idxfd = db->datfd = -1;             /* descriptors */

    /*
     * Allocate room for the name.
     * +5 for ".idx" or ".dat" plus null at end.
     */
    if ((db->name = malloc(namelen + 5)) == NULL)
        err_dump("_db_alloc: malloc error for name");

    /*
     * Allocate an index buffer and a data buffer.
     * +2 for newline and null at end.
     */
    if ((db->idxbuf = malloc(IDXLEN_MAX + 2)) == NULL)
        err_dump("_db_alloc: malloc error for index buffer");
    if ((db->datbuf = malloc(DATLEN_MAX + 2)) == NULL)
        err_dump("_db_alloc: malloc error for data buffer");
    return(db);
}

/*
 * Relinquish access to the database.
 */
void
db_close(DBHANDLE h)
{
    _db_free((DB *)h);  /* closes fds, free buffers & struct */
}

/*
 * Free up a DB structure, and all the malloc'ed buffers it
 * may point to.  Also close the file descriptors if still open.
 */
static void
_db_free(DB *db)
{
    if (db->idxfd >= 0)
        close(db->idxfd);
    if (db->datfd >= 0)
        close(db->datfd);
    if (db->idxbuf != NULL)
        free(db->idxbuf);
    if (db->datbuf != NULL)
        free(db->datbuf);
    if (db->name != NULL)
```

```
        free(db->name);
    free(db);
}

/*
 * Fetch a record.  Return a pointer to the null-terminated data.
 */
char *
db_fetch(DBHANDLE h, const char *key)
{
    DB      *db = h;
    char    *ptr;

    if (_db_find_and_lock(db, key, 0) < 0) {
        ptr = NULL;             /* error, record not found */
        db->cnt_fetcherr++;
    } else {
        ptr = _db_readdat(db);  /* return pointer to data */
        db->cnt_fetchok++;
    }

    /*
     * Unlock the hash chain that _db_find_and_lock locked.
     */
    if (un_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
        err_dump("db_fetch: un_lock error");
    return(ptr);
}

/*
 * Find the specified record.  Called by db_delete, db_fetch,
 * and db_store.  Returns with the hash chain locked.
 */
static int
_db_find_and_lock(DB *db, const char *key, int writelock)
{
    off_t   offset, nextoffset;

    /*
     * Calculate the hash value for this key, then calculate the
     * byte offset of corresponding chain ptr in hash table.
     * This is where our search starts.  First we calculate the
     * offset in the hash table for this key.
     */
    db->chainoff = (_db_hash(db, key) * PTR_SZ) + db->hashoff;
    db->ptroff = db->chainoff;

    /*
     * We lock the hash chain here.  The caller must unlock it
     * when done.  Note we lock and unlock only the first byte.
     */
    if (writelock) {
        if (writew_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
```

```
            err_dump("_db_find_and_lock: writew_lock error");
    } else {
        if (readw_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
            err_dump("_db_find_and_lock: readw_lock error");
    }

    /*
     * Get the offset in the index file of first record
     * on the hash chain (can be 0).
     */
    offset = _db_readptr(db, db->ptroff);
    while (offset != 0) {
        nextoffset = _db_readidx(db, offset);
        if (strcmp(db->idxbuf, key) == 0)
            break;       /* found a match */
        db->ptroff = offset; /* offset of this (unequal) record */
        offset = nextoffset; /* next one to compare */
    }
    /*
     * offset == 0 on error (record not found).
     */
    return(offset == 0 ? -1 : 0);
}

/*
 * Calculate the hash value for a key.
 */
static DBHASH
_db_hash(DB *db, const char *key)
{
    DBHASH      hval = 0;
    char        c;
    int         i;

    for (i = 1; (c = *key++) != 0; i++)
        hval += c * i;      /* ascii char times its 1-based index */
    return(hval % db->nhash);
}

/*
 * Read a chain ptr field from anywhere in the index file:
 * the free list pointer, a hash table chain ptr, or an
 * index record chain ptr.
 */
static off_t
_db_readptr(DB *db, off_t offset)
{
    char    asciiptr[PTR_SZ + 1];

    if (lseek(db->idxfd, offset, SEEK_SET) == -1)
        err_dump("_db_readptr: lseek error to ptr field");
    if (read(db->idxfd, asciiptr, PTR_SZ) != PTR_SZ)
        err_dump("_db_readptr: read error of ptr field");
```

```
    asciiptr[PTR_SZ] = 0;       /* null terminate */
    return(atol(asciiptr));
}

/*
 * Read the next index record.  We start at the specified offset
 * in the index file.  We read the index record into db->idxbuf
 * and replace the separators with null bytes.  If all is OK we
 * set db->datoff and db->datlen to the offset and length of the
 * corresponding data record in the data file.
 */
static off_t
_db_readidx(DB *db, off_t offset)
{
    ssize_t             i;
    char            *ptr1, *ptr2;
    char            asciiptr[PTR_SZ + 1], asciilen[IDXLEN_SZ + 1];
    struct iovec    iov[2];

    /*
     * Position index file and record the offset.  db_nextrec
     * calls us with offset==0, meaning read from current offset.
     * We still need to call lseek to record the current offset.
     */
    if ((db->idxoff = lseek(db->idxfd, offset,
      offset == 0 ? SEEK_CUR : SEEK_SET)) == -1)
        err_dump("_db_readidx: lseek error");

    /*
     * Read the ascii chain ptr and the ascii length at
     * the front of the index record.  This tells us the
     * remaining size of the index record.
     */
    iov[0].iov_base = asciiptr;
    iov[0].iov_len  = PTR_SZ;
    iov[1].iov_base = asciilen;
    iov[1].iov_len  = IDXLEN_SZ;
    if ((i = readv(db->idxfd, &iov[0], 2)) != PTR_SZ + IDXLEN_SZ) {
        if (i == 0 && offset == 0)
            return(-1);     /* EOF for db_nextrec */
        err_dump("_db_readidx: readv error of index record");
    }

    /*
     * This is our return value; always >= 0.
     */
    asciiptr[PTR_SZ] = 0;        /* null terminate */
    db->ptrval = atol(asciiptr); /* offset of next key in chain */

    asciilen[IDXLEN_SZ] = 0;     /* null terminate */
    if ((db->idxlen = atoi(asciilen)) < IDXLEN_MIN ||
      db->idxlen > IDXLEN_MAX)
        err_dump("_db_readidx: invalid length");
```

```
    /*
     * Now read the actual index record.  We read it into the key
     * buffer that we malloced when we opened the database.
     */
    if ((i = read(db->idxfd, db->idxbuf, db->idxlen)) != db->idxlen)
        err_dump("_db_readidx: read error of index record");
    if (db->idxbuf[db->idxlen-1] != NEWLINE)    /* sanity check */
        err_dump("_db_readidx: missing newline");
    db->idxbuf[db->idxlen-1] = 0;    /* replace newline with null */

    /*
     * Find the separators in the index record.
     */
    if ((ptr1 = strchr(db->idxbuf, SEP)) == NULL)
        err_dump("_db_readidx: missing first separator");
    *ptr1++ = 0;                /* replace SEP with null */

    if ((ptr2 = strchr(ptr1, SEP)) == NULL)
        err_dump("_db_readidx: missing second separator");
    *ptr2++ = 0;                /* replace SEP with null */

    if (strchr(ptr2, SEP) != NULL)
        err_dump("_db_readidx: too many separators");

    /*
     * Get the starting offset and length of the data record.
     */
    if ((db->datoff = atol(ptr1)) < 0)
        err_dump("_db_readidx: starting offset < 0");
    if ((db->datlen = atol(ptr2)) <= 0 || db->datlen > DATLEN_MAX)
        err_dump("_db_readidx: invalid length");
    return(db->ptrval);     /* return offset of next key in chain */
}

/*
 * Read the current data record into the data buffer.
 * Return a pointer to the null-terminated data buffer.
 */
static char *
_db_readdat(DB *db)
{
    if (lseek(db->datfd, db->datoff, SEEK_SET) == -1)
        err_dump("_db_readdat: lseek error");
    if (read(db->datfd, db->datbuf, db->datlen) != db->datlen)
        err_dump("_db_readdat: read error");
    if (db->datbuf[db->datlen-1] != NEWLINE)    /* sanity check */
        err_dump("_db_readdat: missing newline");
    db->datbuf[db->datlen-1] = 0; /* replace newline with null */
    return(db->datbuf);     /* return pointer to data record */
}

/*
```

```
 * Delete the specified record.
 */
int
db_delete(DBHANDLE h, const char *key)
{
    DB      *db = h;
    int     rc = 0;         /* assume record will be found */

    if (_db_find_and_lock(db, key, 1) == 0) {
        _db_dodelete(db);
        db->cnt_delok++;
    } else {
        rc = -1;            /* not found */
        db->cnt_delerr++;
    }
    if (un_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
        err_dump("db_delete: un_lock error");
    return(rc);
}

/*
 * Delete the current record specified by the DB structure.
 * This function is called by db_delete and db_store, after
 * the record has been located by _db_find_and_lock.
 */
static void
_db_dodelete(DB *db)
{
    int     i;
    char    *ptr;
    off_t   freeptr, saveptr;

    /*
     * Set data buffer and key to all blanks.
     */
    for (ptr = db->datbuf, i = 0; i < db->datlen - 1; i++)
        *ptr++ = SPACE;
    *ptr = 0;   /* null terminate for _db_writedat */
    ptr = db->idxbuf;
    while (*ptr)
        *ptr++ = SPACE;

    /*
     * We have to lock the free list.
     */
    if (writew_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
        err_dump("_db_dodelete: writew_lock error");

    /*
     * Write the data record with all blanks.
     */
    _db_writedat(db, db->datbuf, db->datoff, SEEK_SET);
```

```
    /*
     * Read the free list pointer.  Its value becomes the
     * chain ptr field of the deleted index record.  This means
     * the deleted record becomes the head of the free list.
     */
    freeptr = _db_readptr(db, FREE_OFF);

    /*
     * Save the contents of index record chain ptr,
     * before it's rewritten by _db_writeidx.
     */
    saveptr = db->ptrval;

    /*
     * Rewrite the index record.  This also rewrites the length
     * of the index record, the data offset, and the data length,
     * none of which has changed, but that's OK.
     */
    _db_writeidx(db, db->idxbuf, db->idxoff, SEEK_SET, freeptr);

    /*
     * Write the new free list pointer.
     */
    _db_writeptr(db, FREE_OFF, db->idxoff);

    /*
     * Rewrite the chain ptr that pointed to this record being
     * deleted.  Recall that _db_find_and_lock sets db->ptroff to
     * point to this chain ptr.  We set this chain ptr to the
     * contents of the deleted record's chain ptr, saveptr.
     */
    _db_writeptr(db, db->ptroff, saveptr);
    if (un_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
        err_dump("_db_dodelete: un_lock error");
}

/*
 * Write a data record.  Called by _db_dodelete (to write
 * the record with blanks) and db_store.
 */
static void
_db_writedat(DB *db, const char *data, off_t offset, int whence)
{
    struct iovec    iov[2];
    static char     newline = NEWLINE;

    /*
     * If we're appending, we have to lock before doing the lseek
     * and write to make the two an atomic operation.  If we're
     * overwriting an existing record, we don't have to lock.
     */
    if (whence == SEEK_END) /* we're appending, lock entire file */
        if (writew_lock(db->datfd, 0, SEEK_SET, 0) < 0)
```

```
            err_dump("_db_writedat: writew_lock error");

    if ((db->datoff = lseek(db->datfd, offset, whence)) == -1)
        err_dump("_db_writedat: lseek error");
    db->datlen = strlen(data) + 1;  /* datlen includes newline */

    iov[0].iov_base = (char *) data;
    iov[0].iov_len  = db->datlen - 1;
    iov[1].iov_base = &newline;
    iov[1].iov_len  = 1;
    if (writev(db->datfd, &iov[0], 2) != db->datlen)
        err_dump("_db_writedat: writev error of data record");

    if (whence == SEEK_END)
        if (un_lock(db->datfd, 0, SEEK_SET, 0) < 0)
            err_dump("_db_writedat: un_lock error");
}

/*
 * Write an index record.  _db_writedat is called before
 * this function to set the datoff and datlen fields in the
 * DB structure, which we need to write the index record.
 */
static void
_db_writeidx(DB *db, const char *key,
             off_t offset, int whence, off_t ptrval)
{
    struct iovec    iov[2];
    char            asciiptrlen[PTR_SZ + IDXLEN_SZ + 1];
    int             len;

    if ((db->ptrval = ptrval) < 0 || ptrval > PTR_MAX)
        err_quit("_db_writeidx: invalid ptr: %d", ptrval);
    sprintf(db->idxbuf, "%s%c%lld%c%ld\n", key, SEP,
      (long long)db->datoff, SEP, (long)db->datlen);
    len = strlen(db->idxbuf);
    if (len < IDXLEN_MIN || len > IDXLEN_MAX)
        err_dump("_db_writeidx: invalid length");
    sprintf(asciiptrlen, "%*lld%*d", PTR_SZ, (long long)ptrval,
      IDXLEN_SZ, len);

    /*
     * If we're appending, we have to lock before doing the lseek
     * and write to make the two an atomic operation.  If we're
     * overwriting an existing record, we don't have to lock.
     */
    if (whence == SEEK_END)     /* we're appending */
        if (writew_lock(db->idxfd, ((db->nhash+1)*PTR_SZ)+1,
          SEEK_SET, 0) < 0)
            err_dump("_db_writeidx: writew_lock error");

    /*
     * Position the index file and record the offset.
```

```
     */
    if ((db->idxoff = lseek(db->idxfd, offset, whence)) == -1)
        err_dump("_db_writeidx: lseek error");

    iov[0].iov_base = asciiptrlen;
    iov[0].iov_len  = PTR_SZ + IDXLEN_SZ;
    iov[1].iov_base = db->idxbuf;
    iov[1].iov_len  = len;
    if (writev(db->idxfd, &iov[0], 2) != PTR_SZ + IDXLEN_SZ + len)
        err_dump("_db_writeidx: writev error of index record");

    if (whence == SEEK_END)
        if (un_lock(db->idxfd, ((db->nhash+1)*PTR_SZ)+1,
          SEEK_SET, 0) < 0)
            err_dump("_db_writeidx: un_lock error");
}

/*
 * Write a chain ptr field somewhere in the index file:
 * the free list, the hash table, or in an index record.
 */
static void
_db_writeptr(DB *db, off_t offset, off_t ptrval)
{
    char    asciiptr[PTR_SZ + 1];

    if (ptrval < 0 || ptrval > PTR_MAX)
        err_quit("_db_writeptr: invalid ptr: %d", ptrval);
    sprintf(asciiptr, "%*lld", PTR_SZ, (long long)ptrval);

    if (lseek(db->idxfd, offset, SEEK_SET) == -1)
        err_dump("_db_writeptr: lseek error to ptr field");
    if (write(db->idxfd, asciiptr, PTR_SZ) != PTR_SZ)
        err_dump("_db_writeptr: write error of ptr field");
}

/*
 * Store a record in the database.  Return 0 if OK, 1 if record
 * exists and DB_INSERT specified, -1 on error.
 */
int
db_store(DBHANDLE h, const char *key, const char *data, int flag)
{
    DB      *db = h;
    int     rc, keylen, datlen;
    off_t   ptrval;

    if (flag != DB_INSERT && flag != DB_REPLACE &&
      flag != DB_STORE) {
        errno = EINVAL;
        return(-1);
    }
    keylen = strlen(key);
```

```
    datlen = strlen(data) + 1;      /* +1 for newline at end */
    if (datlen < DATLEN_MIN || datlen > DATLEN_MAX)
        err_dump("db_store: invalid data length");

    /*
     * _db_find_and_lock calculates which hash table this new record
     * goes into (db->chainoff), regardless of whether it already
     * exists or not. The following calls to _db_writeptr change the
     * hash table entry for this chain to point to the new record.
     * The new record is added to the front of the hash chain.
     */
    if (_db_find_and_lock(db, key, 1) < 0) { /* record not found */
        if (flag == DB_REPLACE) {
            rc = -1;
            db->cnt_storerr++;
            errno = ENOENT;     /* error, record does not exist */
            goto doreturn;
        }

        /*
         * _db_find_and_lock locked the hash chain for us; read
         * the chain ptr to the first index record on hash chain.
         */
        ptrval = _db_readptr(db, db->chainoff);

        if (_db_findfree(db, keylen, datlen) < 0) {
            /*
             * Can't find an empty record big enough. Append the
             * new record to the ends of the index and data files.
             */
            _db_writedat(db, data, 0, SEEK_END);
            _db_writeidx(db, key, 0, SEEK_END, ptrval);

            /*
             * db->idxoff was set by _db_writeidx.  The new
             * record goes to the front of the hash chain.
             */
            _db_writeptr(db, db->chainoff, db->idxoff);
            db->cnt_stor1++;
        } else {
            /*
             * Reuse an empty record. _db_findfree removed it from
             * the free list and set both db->datoff and db->idxoff.
             * Reused record goes to the front of the hash chain.
             */
            _db_writedat(db, data, db->datoff, SEEK_SET);
            _db_writeidx(db, key, db->idxoff, SEEK_SET, ptrval);
            _db_writeptr(db, db->chainoff, db->idxoff);
            db->cnt_stor2++;
        }
    } else {                        /* record found */
        if (flag == DB_INSERT) {
            rc = 1;     /* error, record already in db */
```

```
            db->cnt_storerr++;
            goto doreturn;
        }

        /*
         * We are replacing an existing record.  We know the new
         * key equals the existing key, but we need to check if
         * the data records are the same size.
         */
        if (datlen != db->datlen) {
            _db_dodelete(db);   /* delete the existing record */

            /*
             * Reread the chain ptr in the hash table
             * (it may change with the deletion).
             */
            ptrval = _db_readptr(db, db->chainoff);

            /*
             * Append new index and data records to end of files.
             */
            _db_writedat(db, data, 0, SEEK_END);
            _db_writeidx(db, key, 0, SEEK_END, ptrval);

            /*
             * New record goes to the front of the hash chain.
             */
            _db_writeptr(db, db->chainoff, db->idxoff);
            db->cnt_stor3++;
        } else {
            /*
             * Same size data, just replace data record.
             */
            _db_writedat(db, data, db->datoff, SEEK_SET);
            db->cnt_stor4++;
        }
    }
    rc = 0;     /* OK */

doreturn:   /* unlock hash chain locked by _db_find_and_lock */
    if (un_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
        err_dump("db_store: un_lock error");
    return(rc);
}

/*
 * Try to find a free index record and accompanying data record
 * of the correct sizes.  We're only called by db_store.
 */
static int
_db_findfree(DB *db, int keylen, int datlen)
{
    int     rc;
```

```
    off_t   offset, nextoffset, saveoffset;

    /*
     * Lock the free list.
     */
    if (writew_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
        err_dump("_db_findfree: writew_lock error");

    /*
     * Read the free list pointer.
     */
    saveoffset = FREE_OFF;
    offset = _db_readptr(db, saveoffset);

    while (offset != 0) {
        nextoffset = _db_readidx(db, offset);
        if (strlen(db->idxbuf) == keylen && db->datlen == datlen)
            break;      /* found a match */
        saveoffset = offset;
        offset = nextoffset;
    }

    if (offset == 0) {
        rc = -1;    /* no match found */
    } else {
        /*
         * Found a free record with matching sizes.
         * The index record was read in by _db_readidx above,
         * which sets db->ptrval.  Also, saveoffset points to
         * the chain ptr that pointed to this empty record on
         * the free list.  We set this chain ptr to db->ptrval,
         * which removes the empty record from the free list.
         */
        _db_writeptr(db, saveoffset, db->ptrval);
        rc = 0;

        /*
         * Notice also that _db_readidx set both db->idxoff
         * and db->datoff.  This is used by the caller, db_store,
         * to write the new index record and data record.
         */
    }

    /*
     * Unlock the free list.
     */
    if (un_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
        err_dump("_db_findfree: un_lock error");
    return(rc);
}

/*
 * Rewind the index file for db_nextrec.
```

```
 * Automatically called by db_open.
 * Must be called before first db_nextrec.
 */
void
db_rewind(DBHANDLE h)
{
    DB      *db = h;
    off_t   offset;

    offset = (db->nhash + 1) * PTR_SZ;  /* +1 for free list ptr */

    /*
     * We're just setting the file offset for this process
     * to the start of the index records; no need to lock.
     * +1 below for newline at end of hash table.
     */
    if ((db->idxoff = lseek(db->idxfd, offset+1, SEEK_SET)) == -1)
        err_dump("db_rewind: lseek error");
}

/*
 * Return the next sequential record.
 * We just step our way through the index file, ignoring deleted
 * records.  db_rewind must be called before this function is
 * called the first time.
 */
char *
db_nextrec(DBHANDLE h, char *key)
{
    DB      *db = h;
    char    c;
    char    *ptr;

    /*
     * We read lock the free list so that we don't read
     * a record in the middle of its being deleted.
     */
    if (readw_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
        err_dump("db_nextrec: readw_lock error");

    do {
        /*
         * Read next sequential index record.
         */
        if (_db_readidx(db, 0) < 0) {
            ptr = NULL;     /* end of index file, EOF */
            goto doreturn;
        }

        /*
         * Check if key is all blank (empty record).
         */
        ptr = db->idxbuf;
```

```
        while ((c = *ptr++) != 0  &&  c == SPACE)
            ;   /* skip until null byte or nonblank */
    } while (c == 0);   /* loop until a nonblank key is found */

    if (key != NULL)
        strcpy(key, db->idxbuf);    /* return key */
    ptr = _db_readdat(db);  /* return pointer to data buffer */
    db->cnt_nextrec++;

doreturn:
    if (un_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
        err_dump("db_nextrec: un_lock error");
    return(ptr);
}
```

- 多进程并发访问需要的对记录加锁的功能

第21章　与网络打印机通信

- 一般使用网络打印协议(IPP)与打印机通信
- 打印假脱机守护进程将作业发送到打印机
- 命令行程序将打印作业提交到假脱机守护进程

21.2 网络打印机协议

- IPP建立在超文本传输协议HTTP之上，HTTP又建立在TCP/IP之上

21.3 HTTP

21.4 打印假脱机技术

- IPP首部的结构
    - ```
      struct ipp_hdr {
          int8_t  major_version;  /* always 1 */
          int8_t  minor_version;  /* always 1 */
          union {
              int16_t op; /* operation ID */
              int16_t st; /* status */
          } u;
          int32_t request_id;     /* request ID */
          char    attr_group[1];  /* start of optional attributes group */
          /* optional data follows */
      };
      ```

- printreq结构和printresp结构定义了print程序和打印脱机守护进程之间的协议
- 返回的字符串放在静态缓冲区中，该缓冲区会被紧接着的调用覆盖，不能用于多线程程序
- ```
  /*
   * Print server daemon.
   */
  ```

```c
#include "apue.h"
#include <fcntl.h>
#include <dirent.h>
#include <ctype.h>
#include <pwd.h>
#include <pthread.h>
#include <strings.h>
#include <sys/select.h>
#include <sys/uio.h>

#include "print.h"
#include "ipp.h"

/*
 * These are for the HTTP response from the printer.
 */
#define HTTP_INFO(x)    ((x) >= 100 && (x) <= 199)
#define HTTP_SUCCESS(x) ((x) >= 200 && (x) <= 299)

/*
 * Describes a print job.
 */
struct job {
    struct job      *next;      /* next in list */
    struct job      *prev;      /* previous in list */
    int32_t          jobid;     /* job ID */
    struct printreq  req;       /* copy of print request */
};

/*
 * Describes a thread processing a client request.
 */
struct worker_thread {
    struct worker_thread  *next;    /* next in list */
    struct worker_thread  *prev;    /* previous in list */
    pthread_t              tid;     /* thread ID */
    int                    sockfd;  /* socket */
};

/*
 * Needed for logging.
 */
int                 log_to_stderr = 0;

/*
 * Printer-related stuff.
 */
struct addrinfo     *printer;
char                    *printer_name;
pthread_mutex_t     configlock = PTHREAD_MUTEX_INITIALIZER;
int                 reread;

/*
```

```
 * Thread-related stuff.
 */
struct worker_thread    *workers;
pthread_mutex_t     workerlock = PTHREAD_MUTEX_INITIALIZER;
sigset_t                mask;

/*
 * Job-related stuff.
 */
struct job              *jobhead, *jobtail;
int                 jobfd;
int32_t             nextjob;
pthread_mutex_t     joblock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t          jobwait = PTHREAD_COND_INITIALIZER;

/*
 * Function prototypes.
 */
void        init_request(void);
void        init_printer(void);
void        update_jobno(void);
int32_t get_newjobno(void);
void        add_job(struct printreq *, int32_t);
void        replace_job(struct job *);
void        remove_job(struct job *);
void        build_qonstart(void);
void        *client_thread(void *);
void        *printer_thread(void *);
void        *signal_thread(void *);
ssize_t readmore(int, char **, int, int *);
int     printer_status(int, struct job *);
void        add_worker(pthread_t, int);
void        kill_workers(void);
void        client_cleanup(void *);

/*
 * Main print server thread.  Accepts connect requests from
 * clients and spawns additional threads to service requests.
 *
 * LOCKING: none.
 */
int
main(int argc, char *argv[])
{
    pthread_t           tid;
    struct addrinfo     *ailist, *aip;
    int                 sockfd, err, i, n, maxfd;
    char                *host;
    fd_set              rendezvous, rset;
    struct sigaction    sa;
    struct passwd       *pwdp;

    if (argc != 1)
```

```
        err_quit("usage: printd");
    daemonize("printd");

    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sa.sa_handler = SIG_IGN;
    if (sigaction(SIGPIPE, &sa, NULL) < 0)
        log_sys("sigaction failed");
    sigemptyset(&mask);
    sigaddset(&mask, SIGHUP);
    sigaddset(&mask, SIGTERM);
    if ((err = pthread_sigmask(SIG_BLOCK, &mask, NULL)) != 0)
        log_sys("pthread_sigmask failed");

    n = sysconf(_SC_HOST_NAME_MAX);
    if (n < 0)  /* best guess */
        n = HOST_NAME_MAX;
    if ((host = malloc(n)) == NULL)
        log_sys("malloc error");
    if (gethostname(host, n) < 0)
        log_sys("gethostname error");

    if ((err = getaddrlist(host, "print", &ailist)) != 0) {
        log_quit("getaddrinfo error: %s", gai_strerror(err));
        exit(1);
    }
    FD_ZERO(&rendezvous);
    maxfd = -1;
    for (aip = ailist; aip != NULL; aip = aip->ai_next) {
        if ((sockfd = initserver(SOCK_STREAM, aip->ai_addr,
          aip->ai_addrlen, QLEN)) >= 0) {
            FD_SET(sockfd, &rendezvous);
            if (sockfd > maxfd)
                maxfd = sockfd;
        }
    }
    if (maxfd == -1)
        log_quit("service not enabled");

    pwdp = getpwnam(LPNAME);
    if (pwdp == NULL)
        log_sys("can't find user %s", LPNAME);
    if (pwdp->pw_uid == 0)
        log_quit("user %s is privileged", LPNAME);
    if (setgid(pwdp->pw_gid) < 0 || setuid(pwdp->pw_uid) < 0)
        log_sys("can't change IDs to user %s", LPNAME);

    init_request();
    init_printer();

    err = pthread_create(&tid, NULL, printer_thread, NULL);
    if (err == 0)
        err = pthread_create(&tid, NULL, signal_thread, NULL);
```

```
    if (err != 0)
        log_exit(err, "can't create thread");
    build_qonstart();

    log_msg("daemon initialized");

    for (;;) {
        rset = rendezvous;
        if (select(maxfd+1, &rset, NULL, NULL, NULL) < 0)
            log_sys("select failed");
        for (i = 0; i <= maxfd; i++) {
            if (FD_ISSET(i, &rset)) {
                /*
                 * Accept the connection and handle the request.
                 */
                if ((sockfd = accept(i, NULL, NULL)) < 0)
                    log_ret("accept failed");
                pthread_create(&tid, NULL, client_thread,
                  (void *)((long)sockfd));
            }
        }
    }
    exit(1);
}

/*
 * Initialize the job ID file.  Use a record lock to prevent
 * more than one printer daemon from running at a time.
 *
 * LOCKING: none, except for record-lock on job ID file.
 */
void
init_request(void)
{
    int     n;
    char    name[FILENMSZ];

    sprintf(name, "%s/%s", SPOOLDIR, JOBFILE);
    jobfd = open(name, O_CREAT|O_RDWR, S_IRUSR|S_IWUSR);
    if (write_lock(jobfd, 0, SEEK_SET, 0) < 0)
        log_quit("daemon already running");

    /*
     * Reuse the name buffer for the job counter.
     */
    if ((n = read(jobfd, name, FILENMSZ)) < 0)
        log_sys("can't read job file");
    if (n == 0)
        nextjob = 1;
    else
        nextjob = atol(name);
}
```

```
/*
 * Initialize printer information from configuration file.
 *
 * LOCKING: none.
 */
void
init_printer(void)
{
    printer = get_printaddr();
    if (printer == NULL)
        exit(1);    /* message already logged */
    printer_name = printer->ai_canonname;
    if (printer_name == NULL)
        printer_name = "printer";
    log_msg("printer is %s", printer_name);
}

/*
 * Update the job ID file with the next job number.
 * Doesn't handle wrap-around of job number.
 *
 * LOCKING: none.
 */
void
update_jobno(void)
{
    char    buf[32];

    if (lseek(jobfd, 0, SEEK_SET) == -1)
        log_sys("can't seek in job file");
    sprintf(buf, "%d", nextjob);
    if (write(jobfd, buf, strlen(buf)) < 0)
        log_sys("can't update job file");
}

/*
 * Get the next job number.
 *
 * LOCKING: acquires and releases joblock.
 */
int32_t
get_newjobno(void)
{
    int32_t jobid;

    pthread_mutex_lock(&joblock);
    jobid = nextjob++;
    if (nextjob <= 0)
        nextjob = 1;
    pthread_mutex_unlock(&joblock);
    return(jobid);
}
```

```
/*
 * Add a new job to the list of pending jobs.  Then signal
 * the printer thread that a job is pending.
 *
 * LOCKING: acquires and releases joblock.
 */
void
add_job(struct printreq *reqp, int32_t jobid)
{
    struct job  *jp;

    if ((jp = malloc(sizeof(struct job))) == NULL)
        log_sys("malloc failed");
    memcpy(&jp->req, reqp, sizeof(struct printreq));
    jp->jobid = jobid;
    jp->next = NULL;
    pthread_mutex_lock(&joblock);
    jp->prev = jobtail;
    if (jobtail == NULL)
        jobhead = jp;
    else
        jobtail->next = jp;
    jobtail = jp;
    pthread_mutex_unlock(&joblock);
    pthread_cond_signal(&jobwait);
}

/*
 * Replace a job back on the head of the list.
 *
 * LOCKING: acquires and releases joblock.
 */
void
replace_job(struct job *jp)
{
    pthread_mutex_lock(&joblock);
    jp->prev = NULL;
    jp->next = jobhead;
    if (jobhead == NULL)
        jobtail = jp;
    else
        jobhead->prev = jp;
    jobhead = jp;
    pthread_mutex_unlock(&joblock);
}

/*
 * Remove a job from the list of pending jobs.
 *
 * LOCKING: caller must hold joblock.
 */
void
remove_job(struct job *target)
```

```
{
    if (target->next != NULL)
        target->next->prev = target->prev;
    else
        jobtail = target->prev;
    if (target->prev != NULL)
        target->prev->next = target->next;
    else
        jobhead = target->next;
}

/*
 * Check the spool directory for pending jobs on start-up.
 *
 * LOCKING: none.
 */
void
build_qonstart(void)
{
    int             fd, err, nr;
    int32_t         jobid;
    DIR             *dirp;
    struct dirent   *entp;
    struct printreq req;
    char            dname[FILENMSZ], fname[FILENMSZ];

    sprintf(dname, "%s/%s", SPOOLDIR, REQDIR);
    if ((dirp = opendir(dname)) == NULL)
        return;
    while ((entp = readdir(dirp)) != NULL) {
        /*
         * Skip "." and ".."
         */
        if (strcmp(entp->d_name, ".") == 0 ||
          strcmp(entp->d_name, "..") == 0)
            continue;

        /*
         * Read the request structure.
         */
        sprintf(fname, "%s/%s/%s", SPOOLDIR, REQDIR, entp->d_name);
        if ((fd = open(fname, O_RDONLY)) < 0)
            continue;
        nr = read(fd, &req, sizeof(struct printreq));
        if (nr != sizeof(struct printreq)) {
            if (nr < 0)
                err = errno;
            else
                err = EIO;
            close(fd);
            log_msg("build_qonstart: can't read %s: %s",
              fname, strerror(err));
            unlink(fname);
```

```
            sprintf(fname, "%s/%s/%s", SPOOLDIR, DATADIR,
              entp->d_name);
            unlink(fname);
            continue;
        }
        jobid = atol(entp->d_name);
        log_msg("adding job %d to queue", jobid);
        add_job(&req, jobid);
    }
    closedir(dirp);
}

/*
 * Accept a print job from a client.
 *
 * LOCKING: none.
 */
void *
client_thread(void *arg)
{
    int                 n, fd, sockfd, nr, nw, first;
    int32_t             jobid;
    pthread_t           tid;
    struct printreq     req;
    struct printresp    res;
    char                name[FILENMSZ];
    char                buf[IOBUFSZ];

    tid = pthread_self();
    pthread_cleanup_push(client_cleanup, (void *)((long)tid));
    sockfd = (long)arg;
    add_worker(tid, sockfd);

    /*
     * Read the request header.
     */
    if ((n = treadn(sockfd, &req, sizeof(struct printreq), 10)) !=
      sizeof(struct printreq)) {
        res.jobid = 0;
        if (n < 0)
            res.retcode = htonl(errno);
        else
            res.retcode = htonl(EIO);
        strncpy(res.msg, strerror(res.retcode), MSGLEN_MAX);
        writen(sockfd, &res, sizeof(struct printresp));
        pthread_exit((void *)1);
    }
    req.size = ntohl(req.size);
    req.flags = ntohl(req.flags);

    /*
     * Create the data file.
     */
```

```
    jobid = get_newjobno();
    sprintf(name, "%s/%s/%d", SPOOLDIR, DATADIR, jobid);
    fd = creat(name, FILEPERM);
    if (fd < 0) {
        res.jobid = 0;
        res.retcode = htonl(errno);
        log_msg("client_thread: can't create %s: %s", name,
          strerror(res.retcode));
        strncpy(res.msg, strerror(res.retcode), MSGLEN_MAX);
        writen(sockfd, &res, sizeof(struct printresp));
        pthread_exit((void *)1);
    }

    /*
     * Read the file and store it in the spool directory.
     * Try to figure out if the file is a PostScript file
     * or a plain text file.
     */
    first = 1;
    while ((nr = tread(sockfd, buf, IOBUFSZ, 20)) > 0) {
        if (first) {
            first = 0;
            if (strncmp(buf, "%!PS", 4) != 0)
                req.flags |= PR_TEXT;
        }
        nw = write(fd, buf, nr);
        if (nw != nr) {
            res.jobid = 0;
            if (nw < 0)
                res.retcode = htonl(errno);
            else
                res.retcode = htonl(EIO);
            log_msg("client_thread: can't write %s: %s", name,
              strerror(res.retcode));
            close(fd);
            strncpy(res.msg, strerror(res.retcode), MSGLEN_MAX);
            writen(sockfd, &res, sizeof(struct printresp));
            unlink(name);
            pthread_exit((void *)1);
        }
    }
    close(fd);

    /*
     * Create the control file.  Then write the
     * print request information to the control
     * file.
     */
    sprintf(name, "%s/%s/%d", SPOOLDIR, REQDIR, jobid);
    fd = creat(name, FILEPERM);
    if (fd < 0) {
        res.jobid = 0;
        res.retcode = htonl(errno);
```

```
        log_msg("client_thread: can't create %s: %s", name,
          strerror(res.retcode));
        strncpy(res.msg, strerror(res.retcode), MSGLEN_MAX);
        writen(sockfd, &res, sizeof(struct printresp));
        sprintf(name, "%s/%s/%d", SPOOLDIR, DATADIR, jobid);
        unlink(name);
        pthread_exit((void *)1);
    }
    nw = write(fd, &req, sizeof(struct printreq));
    if (nw != sizeof(struct printreq)) {
        res.jobid = 0;
        if (nw < 0)
            res.retcode = htonl(errno);
        else
            res.retcode = htonl(EIO);
        log_msg("client_thread: can't write %s: %s", name,
          strerror(res.retcode));
        close(fd);
        strncpy(res.msg, strerror(res.retcode), MSGLEN_MAX);
        writen(sockfd, &res, sizeof(struct printresp));
        unlink(name);
        sprintf(name, "%s/%s/%d", SPOOLDIR, DATADIR, jobid);
        unlink(name);
        pthread_exit((void *)1);
    }
    close(fd);

    /*
     * Send response to client.
     */
    res.retcode = 0;
    res.jobid = htonl(jobid);
    sprintf(res.msg, "request ID %d", jobid);
    writen(sockfd, &res, sizeof(struct printresp));

    /*
     * Notify the printer thread, clean up, and exit.
     */
    log_msg("adding job %d to queue", jobid);
    add_job(&req, jobid);
    pthread_cleanup_pop(1);
    return((void *)0);
}

/*
 * Add a worker to the list of worker threads.
 *
 * LOCKING: acquires and releases workerlock.
 */
void
add_worker(pthread_t tid, int sockfd)
{
    struct worker_thread    *wtp;
```

```
    if ((wtp = malloc(sizeof(struct worker_thread))) == NULL) {
        log_ret("add_worker: can't malloc");
        pthread_exit((void *)1);
    }
    wtp->tid = tid;
    wtp->sockfd = sockfd;
    pthread_mutex_lock(&workerlock);
    wtp->prev = NULL;
    wtp->next = workers;
    if (workers == NULL)
        workers = wtp;
    else
        workers->prev = wtp;
    pthread_mutex_unlock(&workerlock);
}

/*
 * Cancel (kill) all outstanding workers.
 *
 * LOCKING: acquires and releases workerlock.
 */
void
kill_workers(void)
{
    struct worker_thread    *wtp;

    pthread_mutex_lock(&workerlock);
    for (wtp = workers; wtp != NULL; wtp = wtp->next)
        pthread_cancel(wtp->tid);
    pthread_mutex_unlock(&workerlock);
}

/*
 * Cancellation routine for the worker thread.
 *
 * LOCKING: acquires and releases workerlock.
 */
void
client_cleanup(void *arg)
{
    struct worker_thread    *wtp;
    pthread_t               tid;

    tid = (pthread_t)((long)arg);
    pthread_mutex_lock(&workerlock);
    for (wtp = workers; wtp != NULL; wtp = wtp->next) {
        if (wtp->tid == tid) {
            if (wtp->next != NULL)
                wtp->next->prev = wtp->prev;
            if (wtp->prev != NULL)
                wtp->prev->next = wtp->next;
            else
```

```
                workers = wtp->next;
            break;
        }
    }
    pthread_mutex_unlock(&workerlock);
    if (wtp != NULL) {
        close(wtp->sockfd);
        free(wtp);
    }
}

/*
 * Deal with signals.
 *
 * LOCKING: acquires and releases configlock.
 */
void *
signal_thread(void *arg)
{
    int     err, signo;

    for (;;) {
        err = sigwait(&mask, &signo);
        if (err != 0)
            log_quit("sigwait failed: %s", strerror(err));
        switch (signo) {
        case SIGHUP:
            /*
             * Schedule to re-read the configuration file.
             */
            pthread_mutex_lock(&configlock);
            reread = 1;
            pthread_mutex_unlock(&configlock);
            break;

        case SIGTERM:
            kill_workers();
            log_msg("terminate with signal %s", strsignal(signo));
            exit(0);

        default:
            kill_workers();
            log_quit("unexpected signal %d", signo);
        }
    }
}

/*
 * Add an option to the IPP header.
 *
 * LOCKING: none.
 */
char *
```

```
add_option(char *cp, int tag, char *optname, char *optval)
{
    int     n;
    union {
        int16_t s;
        char c[2];
    }       u;

    *cp++ = tag;
    n = strlen(optname);
    u.s = htons(n);
    *cp++ = u.c[0];
    *cp++ = u.c[1];
    strcpy(cp, optname);
    cp += n;
    n = strlen(optval);
    u.s = htons(n);
    *cp++ = u.c[0];
    *cp++ = u.c[1];
    strcpy(cp, optval);
    return(cp + n);
}

/*
 * Single thread to communicate with the printer.
 *
 * LOCKING: acquires and releases joblock and configlock.
 */
void *
printer_thread(void *arg)
{
    struct job      *jp;
    int             hlen, ilen, sockfd, fd, nr, nw, extra;
    char            *icp, *hcp, *p;
    struct ipp_hdr  *hp;
    struct stat     sbuf;
    struct iovec    iov[2];
    char            name[FILENMSZ];
    char            hbuf[HBUFSZ];
    char            ibuf[IBUFSZ];
    char            buf[IOBUFSZ];
    char            str[64];
    struct timespec ts = { 60, 0 };     /* 1 minute */

    for (;;) {
        /*
         * Get a job to print.
         */
        pthread_mutex_lock(&joblock);
        while (jobhead == NULL) {
            log_msg("printer_thread: waiting...");
            pthread_cond_wait(&jobwait, &joblock);
        }
```

```
        remove_job(jp = jobhead);
        log_msg("printer_thread: picked up job %d", jp->jobid);
        pthread_mutex_unlock(&joblock);
        update_jobno();

        /*
         * Check for a change in the config file.
         */
        pthread_mutex_lock(&configlock);
        if (reread) {
            freeaddrinfo(printer);
            printer = NULL;
            printer_name = NULL;
            reread = 0;
            pthread_mutex_unlock(&configlock);
            init_printer();
        } else {
            pthread_mutex_unlock(&configlock);
        }

        /*
         * Send job to printer.
         */
        sprintf(name, "%s/%s/%d", SPOOLDIR, DATADIR, jp->jobid);
        if ((fd = open(name, O_RDONLY)) < 0) {
            log_msg("job %d canceled - can't open %s: %s",
              jp->jobid, name, strerror(errno));
            free(jp);
            continue;
        }
        if (fstat(fd, &sbuf) < 0) {
            log_msg("job %d canceled - can't fstat %s: %s",
              jp->jobid, name, strerror(errno));
            free(jp);
            close(fd);
            continue;
        }
        if ((sockfd = connect_retry(AF_INET, SOCK_STREAM, 0,
          printer->ai_addr, printer->ai_addrlen)) < 0) {
            log_msg("job %d deferred - can't contact printer: %s",
              jp->jobid, strerror(errno));
            goto defer;
        }

        /*
         * Set up the IPP header.
         */
        icp = ibuf;
        hp = (struct ipp_hdr *)icp;
        hp->major_version = 1;
        hp->minor_version = 1;
        hp->operation = htons(OP_PRINT_JOB);
        hp->request_id = htonl(jp->jobid);
```

```
        icp += offsetof(struct ipp_hdr, attr_group);
        *icp++ = TAG_OPERATION_ATTR;
        icp = add_option(icp, TAG_CHARSET, "attributes-charset",
          "utf-8");
        icp = add_option(icp, TAG_NATULANG,
          "attributes-natural-language", "en-us");
        sprintf(str, "http://%s/ipp", printer_name);
        icp = add_option(icp, TAG_URI, "printer-uri", str);
        icp = add_option(icp, TAG_NAMEWOLANG,
          "requesting-user-name", jp->req.usernm);
        icp = add_option(icp, TAG_NAMEWOLANG, "job-name",
          jp->req.jobnm);
        if (jp->req.flags & PR_TEXT) {
            p = "text/plain";
            extra = 1;
        } else {
            p = "application/postscript";
            extra = 0;
        }
        icp = add_option(icp, TAG_MIMETYPE, "document-format", p);
        *icp++ = TAG_END_OF_ATTR;
        ilen = icp - ibuf;

        /*
         * Set up the HTTP header.
         */
        hcp = hbuf;
        sprintf(hcp, "POST /ipp HTTP/1.1\r\n");
        hcp += strlen(hcp);
        sprintf(hcp, "Content-Length: %ld\r\n",
          (long)sbuf.st_size + ilen + extra);
        hcp += strlen(hcp);
        strcpy(hcp, "Content-Type: application/ipp\r\n");
        hcp += strlen(hcp);
        sprintf(hcp, "Host: %s:%d\r\n", printer_name, IPP_PORT);
        hcp += strlen(hcp);
        *hcp++ = '\r';
        *hcp++ = '\n';
        hlen = hcp - hbuf;

        /*
         * Write the headers first.  Then send the file.
         */
        iov[0].iov_base = hbuf;
        iov[0].iov_len = hlen;
        iov[1].iov_base = ibuf;
        iov[1].iov_len = ilen;
        if (writev(sockfd, iov, 2) != hlen + ilen) {
            log_ret("can't write to printer");
            goto defer;
        }

        if (jp->req.flags & PR_TEXT) {
```

```
             /*
              * Hack: allow PostScript to be printed as plain text.
              */
             if (write(sockfd, "\b", 1) != 1) {
                 log_ret("can't write to printer");
                 goto defer;
             }
         }

         while ((nr = read(fd, buf, IOBUFSZ)) > 0) {
             if ((nw = writen(sockfd, buf, nr)) != nr) {
                 if (nw < 0)
                   log_ret("can't write to printer");
                 else
                   log_msg("short write (%d/%d) to printer", nw, nr);
                 goto defer;
             }
         }
         if (nr < 0) {
             log_ret("can't read %s", name);
             goto defer;
         }

         /*
          * Read the response from the printer.
          */
         if (printer_status(sockfd, jp)) {
             unlink(name);
             sprintf(name, "%s/%s/%d", SPOOLDIR, REQDIR, jp->jobid);
             unlink(name);
             free(jp);
             jp = NULL;
         }
defer:
         close(fd);
         if (sockfd >= 0)
             close(sockfd);
         if (jp != NULL) {
             replace_job(jp);
             nanosleep(&ts, NULL);
         }
     }
}

/*
 * Read data from the printer, possibly increasing the buffer.
 * Returns offset of end of data in buffer or -1 on failure.
 *
 * LOCKING: none.
 */
ssize_t
readmore(int sockfd, char **bpp, int off, int *bszp)
{
```

```
    ssize_t nr;
    char    *bp = *bpp;
    int     bsz = *bszp;

    if (off >= bsz) {
        bsz += IOBUFSZ;
        if ((bp = realloc(*bpp, bsz)) == NULL)
            log_sys("readmore: can't allocate bigger read buffer");
        *bszp = bsz;
        *bpp = bp;
    }
    if ((nr = tread(sockfd, &bp[off], bsz-off, 1)) > 0)
        return(off+nr);
    else
        return(-1);
}

/*
 * Read and parse the response from the printer.  Return 1
 * if the request was successful, and 0 otherwise.
 *
 * LOCKING: none.
 */
int
printer_status(int sfd, struct job *jp)
{
    int             i, success, code, len, found, bufsz, datsz;
    int32_t         jobid;
    ssize_t         nr;
    char            *bp, *cp, *statcode, *reason, *contentlen;
    struct ipp_hdr  *hp;

    /*
     * Read the HTTP header followed by the IPP response header.
     * They can be returned in multiple read attempts.  Use the
     * Content-Length specifier to determine how much to read.
     */
    success = 0;
    bufsz = IOBUFSZ;
    if ((bp = malloc(IOBUFSZ)) == NULL)
        log_sys("printer_status: can't allocate read buffer");

    while ((nr = tread(sfd, bp, bufsz, 5)) > 0) {
        /*
         * Find the status.  Response starts with "HTTP/x.y"
         * so we can skip the first 8 characters.
         */
        cp = bp + 8;
        datsz = nr;
        while (isspace((int)*cp))
            cp++;
        statcode = cp;
        while (isdigit((int)*cp))
```

```
            cp++;
        if (cp == statcode) { /* Bad format; log it and move on */
            log_msg(bp);
        } else {
            *cp++ = '\0';
            reason = cp;
            while (*cp != '\r' && *cp != '\n')
                cp++;
            *cp = '\0';
            code = atoi(statcode);
            if (HTTP_INFO(code))
                continue;
            if (!HTTP_SUCCESS(code)) { /* probable error: log it */
                bp[datsz] = '\0';
                log_msg("error: %s", reason);
                break;
            }

            /*
             * HTTP request was okay, but still need to check
             * IPP status.  Search for the Content-Length.
             */
            i = cp - bp;
            for (;;) {
                while (*cp != 'C' && *cp != 'c' && i < datsz) {
                    cp++;
                    i++;
                }
                if (i >= datsz) {   /* get more header */
                    if ((nr = readmore(sfd, &bp, i, &bufsz)) < 0) {
                        goto out;
                    } else {
                        cp = &bp[i];
                        datsz += nr;
                    }
                }

                if (strncasecmp(cp, "Content-Length:", 15) == 0) {
                    cp += 15;
                    while (isspace((int)*cp))
                        cp++;
                    contentlen = cp;
                    while (isdigit((int)*cp))
                        cp++;
                    *cp++ = '\0';
                    i = cp - bp;
                    len = atoi(contentlen);
                    break;
                } else {
                    cp++;
                    i++;
                }
            }
```

```
            if (i >= datsz) {   /* get more header */
                if ((nr = readmore(sfd, &bp, i, &bufsz)) < 0) {
                    goto out;
                } else {
                    cp = &bp[i];
                    datsz += nr;
                }
            }

            found = 0;
            while (!found) {  /* look for end of HTTP header */
                while (i < datsz - 2) {
                    if (*cp == '\n' && *(cp + 1) == '\r' &&
                      *(cp + 2) == '\n') {
                        found = 1;
                        cp += 3;
                        i += 3;
                        break;
                    }
                    cp++;
                    i++;
                }
                if (i >= datsz) {   /* get more header */
                    if ((nr = readmore(sfd, &bp, i, &bufsz)) < 0) {
                        goto out;
                    } else {
                        cp = &bp[i];
                        datsz += nr;
                    }
                }
            }

            if (datsz - i < len) {  /* get more header */
                if ((nr = readmore(sfd, &bp, i, &bufsz)) < 0) {
                    goto out;
                } else {
                    cp = &bp[i];
                    datsz += nr;
                }
            }

            hp = (struct ipp_hdr *)cp;
            i = ntohs(hp->status);
            jobid = ntohl(hp->request_id);

            if (jobid != jp->jobid) {
                /*
                 * Different jobs.  Ignore it.
                 */
                log_msg("jobid %d status code %d", jobid, i);
                break;
            }
```

```
            if (STATCLASS_OK(i))
                success = 1;
            break;
        }
    }

out:
    free(bp);
    if (nr < 0) {
        log_msg("jobid %d: error reading printer response: %s",
          jobid, strerror(errno));
    }
    return(success);
}
```

-