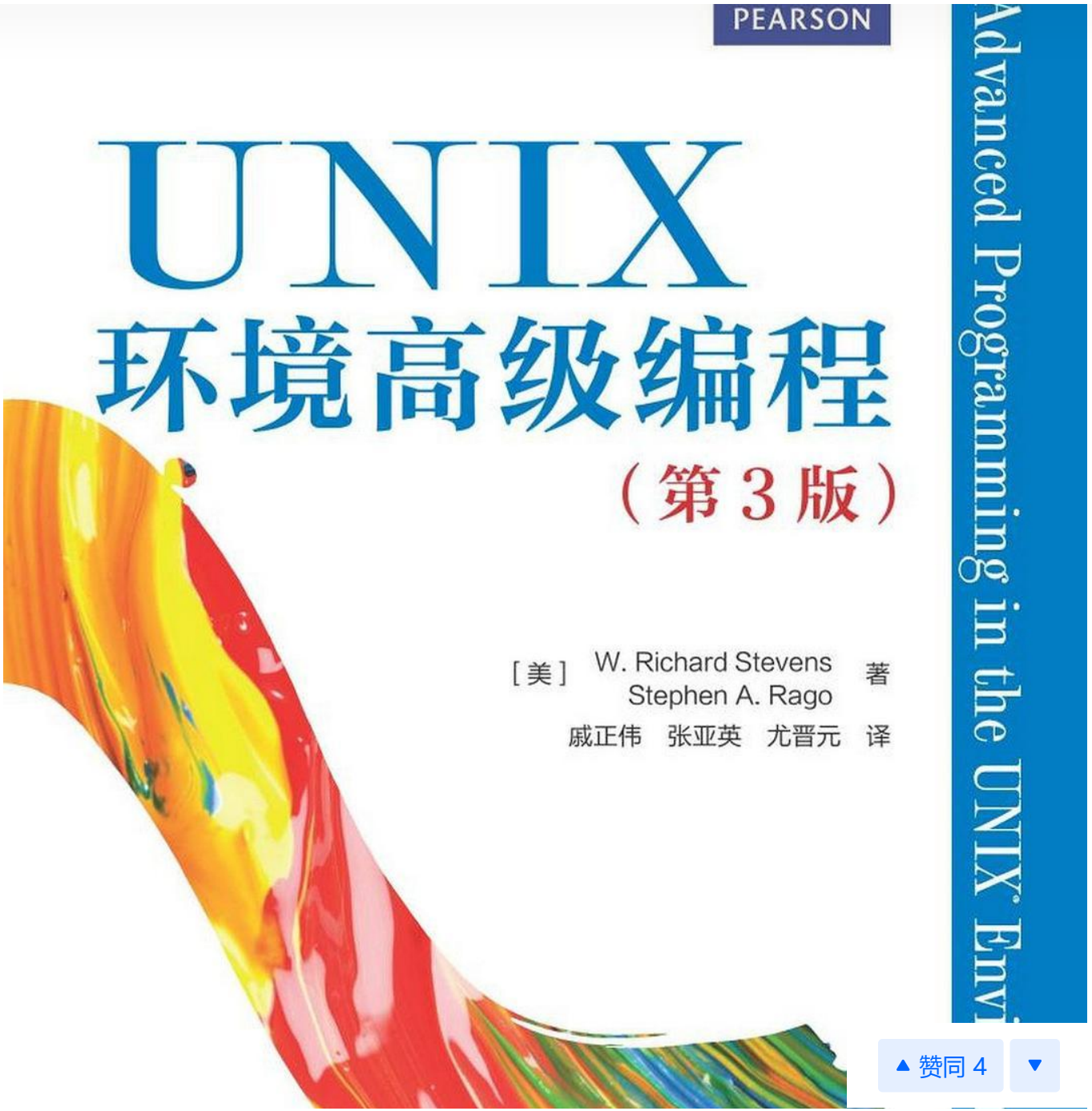


UNIX环境高级编程概念整理

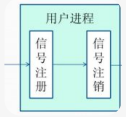
Hanjie WU
学生

参考资料：

UNIX环境高级编程
book.douban.com/subject/1788421/

Linux 下的进程间通信：套接字和信号
linux.cn/article-10930-1.html

进程间通信IPC (InterProcess Communication)
www.jianshu.com/p/c1015f5ffa74



操作系统（内核kernel）是一种软件，控制计算机硬件资源，提供程序运行环境。内核的接口称为系统调用。公用函数库构建在系统调用接口之上，应用程序既可以使用公用函数库也可以使用系统调用。Shell是一种特殊的应用程序，为运行其他应用程序提供了一个接口。

图 1-1　UNIX 操作系统的体系结构

程序：

程序是一个存储在磁盘上某个目录中的可执行文件。内核使用exec函数，将程序读入内存，并执行程序。

进程：

程序的执行实例被称为进程

出错处理：

POSIX和ISO C将error定义为一个符号，并扩展成为一个可修改的整形左值。

对于errno应当注意两条规则：

1.如果没有出错，其值不会被例程清除。因此，仅当函数的返回值出错时，才检验其值

2.任何函数都不会将errno值设置为0，而且在<errno.h>中定义的所有常量都不为0

时间值：

当度量一个进程的执行时间时，Unix系统为一个进程维护了3个进程时间值

时钟时间：进程运行时间的总量

用户CPU时间：执行用户指令所用的时间量

系统CPU时间：为该进程执行内核程序所经历的时间量

用时间，但不包括等待I/O或其他进程运行所占时间，它是一个相对固定的值。进程的墙钟时间是进程从开始执行到结束期间墙钟实际走动的时间。由于分时的原因，进程的墙钟时间可能包含了其他进程的运行时间，因此对于一个进程的每一次运行而言，它可能是一个不固定的值。

系统调用和库函数的差别：

1、系统调用是运行于内核状态；而库函数由用户调用，运行于用户态。

2、系统调用通常提供一种最小接口，而库函数通常提供比较复杂的功能。

3、系统调用是为了方便使用操作系统的接口，而库函数则是为了人们编程的方便。

库函数就是为了减少系统调用的次数

微内核C/S模式：

微内核的特点：

（1）足够小的内核

内核做到足够小。

实现与硬件紧密相关的处理，实现一些较基本的功能，负责客服端和服务器之间的通信

（2）基于 C/S 模式

将操作系统中最基本的部分放入内核中，把操作系统的绝大部分功能放在微内核外面的一组服务器（进程）中实现。这些服务器运行在用户态，客户与服务器之间借助微内核提供的消息传递机制来实现通信。如：

用于对进程（线程）进行管理的进程（线程）服务器，提供虚拟存储器管理功能的存储器服务器，提供I/O设备管理的I/O设备管理服务器

（3）"机制与策略分离"原理

在传统的OS中，机制通常放在OS的内核较低层，策略放在内核的较高层。而在微内核的OS中，通常将机制放在OS的微内核中。这样微内核才能够做的更小。

（4）采用面向对象技术

优点：可扩展性，可靠性，可移植性

内核态和用户态：

内核态：控制计算机的硬件资源，并提供上层应用程序运行的环境。

用户态：上层应用程序的活动空间，应用程序的执行必须依托于内核提供的资源。

系统调用：为了使上层应用能够访问到这些资源，内核为上层应用提供访问的接口。

用户态切换为内核态的三种情况：

1.系统调用：主动，软中断

2.异常事件： 当CPU正在执行运行在用户态的程序时，突然发生某些预先不可知的异常事件，这个时候就会触发从当前用户态执行的进程转向内核态执行相关的异常事件，典型的如缺页异常。被动

3.外围设备的中断：当外围设备完成用户的请求操作后，会向CPU发出中断信号，此时，CPU就会暂停执行下一条即将要执行的指令，转而去执行中断信号

计算机启动过程

通电，BIOS程序刷入内存，

进行硬件自检，BIOS程序首先检查，计算机硬件能否满足运行的基本条件

自检完成后，BIOS按照启动顺序，把控制权转移到排在第一位的储存设备，读取设备第一个扇区，如果这设备不可以用于启动，则读取下一位设备，直到读取到MBR（主引导记录）

根据MBR，将控制权转移给操作系统所在分区，读取磁盘分区。

控制权转交给操作系统后，操作系统内核首先被载入内存，内核载入成功后，首先产生init进程，然后init进程加载系统各个模块，比如窗口程序和网络程序，直至执行login程序，等待用户输入用户名和密码。

POSIX标准：

可移植操作系统接口（缩写为POSIX）是IEEE为要在各种UNIX操作系统上运行软件，而定义API的一系列互相关联的标准的总称。在一个PC

文件I/O

文件描述符：

对于内核而言，所有打开的文件都通过文件描述符引用。文件描述符是一个非负整数。当打开一个现有文件或新建一个新文件时，内核向进程返回一个文件描述符。当读，写一个文件时，使用open或create返回的文件描述符标识该文件，将其作为参数传递给read或write。

UNIX中文件描述符0（STDIN_FILENO)表示标准输入，1(STDOUT_FILENO)表示标准输出，2(STDERR_FILENO)表示标准错误。

为什么要close文件：

如果以共享方式打开文件，系统会在文件表中的引用计数上记录引用该文件的进程数。关闭文件后，引用计数减1。当引用计数变为0时，系统才会从内存中删除这个文件表。如果不关闭，这个文件表会一直留在内存中，占用内存。

文件共享：

内核使用3种数据结构表示打开的文件，他们之间的关系决定了在文件共享方面一个进程对另一个进程的影响。

（1） 每个进程在进程表中都有一个纪录项，纪录项中包含一张打开文件描述符表，每个文件描述符各占一项，与每个文件描述符相关的是

a. 文件描述符标志

b. 指向一个文件表项（内核维护）的指针

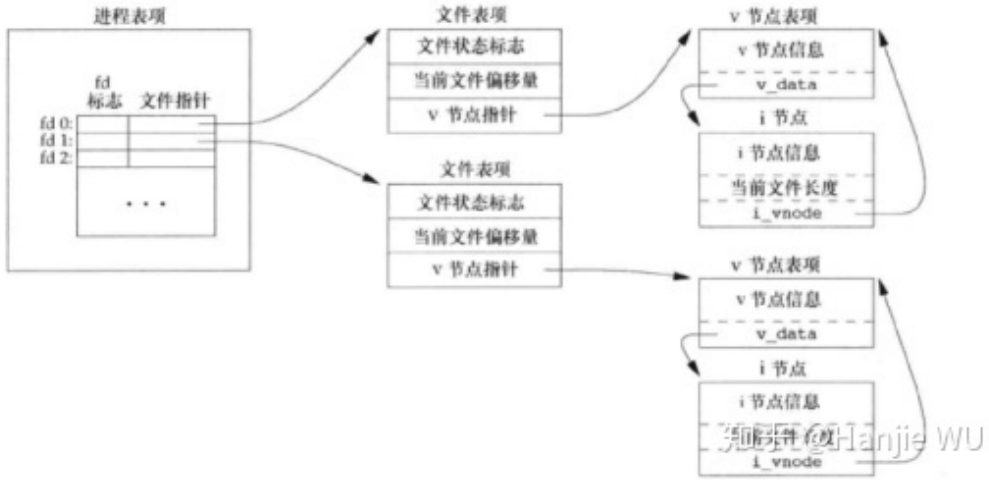
（2） 内核为所有打开文件维护一张文件表项，每个文件表项包含：

a. 文件状态(读 写 同步 非阻塞等)

b. 当前文件偏移量

c. 指向该文件V节点(i节点)的指针

索引节点，这些信息是在打开文件时从磁盘读入内存的，所以，文件的所有文件信息都是随时可用的。i节点包含了文件的所有者，文件长度，指向文件实际数据块在磁盘上位置的指针等。



原子操作

多步组成的一个操作，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。

文件类型：

的数据。基于这种数据是文本还是二进制数据对于内核而言并无区别。对普通文件内容的解释由处理该文件的应用程序进行。

(2) 目录文件(directory file)。这种文件包含了其他文件的名字以及指向与这些文件有关信息的指针。对一个目录文件具有读许可权的任一进程都可以读该目录的内容，但只有内核可以写目录文件。

(3) 字符特殊文件(character special file)。这种类型的文件提供对设备不带缓冲的访问，每次的访问长度可变。

(4) 块特殊文件(block special file)。这种文件典型地用于磁盘设备。提供对设备带缓冲的访问，每次访问以固定长度单位进行。系统中的所有设备或者是字符特殊文件，或者是块特殊文件。

(5) FIFO。这种文件用于进程间的通信，有时也将其称为命名管道。14.5节将对其进行说明。

(6) 套接口(socket)。这种文件用于进程间的网络通信。套接口也可用于在一台宿主机上的进程之间的非网络通信。第15章将用套接口进行进程间的通信。

(7) 符号链接文件，这种类型的文件指向另一个文件

所有文件类型（目录、字符特别文件等）都有访问权限。st_mode值包含了对文件的访问权限位。每个文件有9个访问权限位，可将它们分为3类，如图

常量	说明	对普通文件的影响	对目录的影响
S_ISUID	设置用户 ID	执行时设置有效用户 ID	（未使用）
S_ISGID	设置组 ID	若组执行位设置，则执行时设置有效组 ID；否则使强制性锁起作用（若支持）	将在目录中创建的新文件的组 ID 设置为目录的组 ID
S_ISVTX	粘着位	在交换区缓存程序正文（若支持）	限止在目录中删除和重命名文件
S_IRUSR	用户读	许可用户读文件	许可用户读目录项
S_IWUSR	用户写	许可用户写文件	许可用户在目录中删除和创建文件
S_IXUSR	用户执行	许可用户执行文件	许可用户在目录中搜索给定路径名
S_IRGRP	组读	许可组读文件	许可组读目录项
S_IWGRP	组写	许可组写文件	许可组在目录中删除和创建文件
S_IXGRP	组执行	许可组执行文件	许可组在目录中搜索给定路径名
S_IROTH	其他读	许可其他读文件	许可其他读目录项
S_IWOTH	其他写	许可其他写文件	许可其他在目录中删除和创建文件
S_IXOTH	其他执行	许可其他执行文件	许可其他在目录中搜索给定路径名

图 4-26　文件访问权限位小结

在一个目录内创建和删除文件，都必须对包含该文件的目录具有写权限和执行权限。进程每次打开，创建，删除一个文件时，内核都会进行文件访问权限检查。

粘着位（sticky bit，saved-text bit)

如果一个可执行程序文件的这一位被设置，那么当该程序第一次被执行，在其终止时，程序正文部分（机器指令）的一个副本仍被保存在交

INODE:

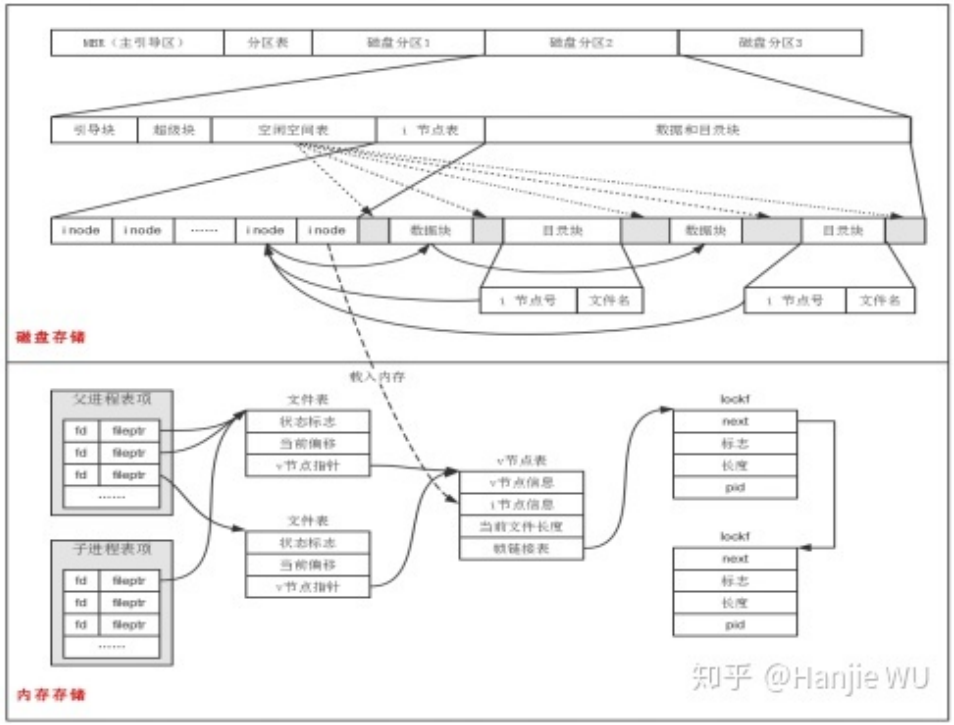
inode是指在许多“类Unix文件系统”中的一种数据结构。每个inode保存了文件系统中所存储文件的元信息，包含着这些内容：

- 文件的字节数。
- 文件创建者的ID。
- 文件的Group ID。
- 文件的读写等权限。
- 文件的相关时间戳。具体的有三个：ctime-->inode上一次变动的时间；mtime-->文件内容上一次变动的时间；atime-->文件上一次打开的时间。
- 链接数
- 文件数据块的位置
- inode号码

但不包括数据内容或者文件名。**操作系统是通过inode号码来识别不同文件的。**

在Unix / Linux系统中，用户层名是通过文件名来打开文件的，系统层面主要是通过了三个步骤来打开文件：

- 通过inode号码获取inode信息。
- 根据inode信息，找到文件数据所存的块，并读取数据。



inode的特殊作用

Unix / Linux系统中inode号码和文件名分离，这导致了系统中一些特别的现象：

- 删除inode节点，即是删除文件。有些文件可能无法正确删除，这时我们直接删掉对应的inode节点，就可以起到删除文件的作用。

通常来说，系统是无法通过inode号码得到文件名的，当打开一个文件，系统往后就通过inode来识别该文件，不再考虑文件名。

- 因为inode号码的存在，系统可以在软件不关闭的情况下进行更新。系统通过inode号码，识别运行中的文件，更新过程中，文件以相同的文件名，新的inode存在，而不会影响到目前运行中的文件。而原先旧版的inode会在软件下一次打开时被回收，文件名会自动指向新的inode号码。

目录文件的结构就是一个列表。**目录项 = 所包含文件文件名 + 对应inode号码。**

在inode中，有一个存储项叫做“链接数”，记录目录项指向该inode的总数。如果通过硬链接方式创建一个文件名指向某文件，那该文件对应的inode数据域中链接数部分就会 + 1，反之 - 1 。当这个值为0时，系统就会默认没有文件名指向该inode，此时，就会回收该inode号码，并且回收对应的块区域。

硬链接：

创建一个硬链接效果就是，使用目标文件名和目标文件名的inode编号存放在目录下面。一旦创建硬链接之后，那么被链接的文件的属性里面就会将链接数目+1。链接数目对应于struct stat 结构里面的st_nlink字段。

①硬链接通常要求链接和文件位于同一文件系统中

(因为硬链接是使用inode节点来操作的，所以硬链接不可以跨越文件系统)

②只有超级用户才能创建指向目录的硬链接

符号链接

符号链接是对一个文件的间接指针，它与硬链接有所不同，硬链接直接指向文件的i节点。符号链接包含的是文件的路径名。引入符号链接的目的是为了避开硬链接上述的一些限制。

对符号链接以及它指向何种对象，并无任何文件系统限制，任何用户都可以创建指向目录的符号链接。符号链接一般用于将一个文件或整个目录结构移到系统中另一个位置。

工作目录：

从逻辑上讲，用户在登录到Linux系统中之后，每时每刻都处在某个目录之中，此目录被称做工作目录。工作目录是可以随时改变的。用户初始登录到系

所有相对路径的起点

不带缓冲的IO和带缓冲的IO：

1. 不带缓存IO，例如read()和write()函数，它们都属于系统调用，在用户层没有缓存，但在内核进行了缓存。

2. 带缓存IO也叫标准IO，不依赖系统内核，所以移植性强，我们使用标准IO操作很多时候是为了减少对read()和write()的系统调用次数，带缓存IO其实就是在用户层再建立一个缓存区，这个缓存区的分配和优化长度等细节都是标准IO库了来处理。

标准I/O库：

标准I/O库打开或创建文件时，使一个流与一个文件相关联。

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：在把缓冲区全部填满之后，才进行IO操作，换言之在缓冲区没满之前get之类的操作是停止的。flush操作就是将缓冲区中的数据写到磁盘上。磁盘文件是全缓冲。
- 行缓冲：在输入或者输出过程中遇到换行符，标准I/O库进行

UNIX加密口令：

/etc/passwd 文件是存取用户密码口令的文件夹，该文件允许所有用户读取，易导致用户密码泄露，因此 Linux 系统将用户的密码信息从 /etc/passwd 文件中分离出来，并单独放到了/etc/shadow文件夹中。/etc/shadow 文件只有 root 用户拥有读权限，其他用户没有任何权限，这样就保证了用户密码的安全性。

加盐算法：

用户密码在存储在/etc/shadow时，不是直接存储密码明文通过特定加密算法后的密文，而是先用程序随机生成一段字符串，称为盐。将用户明文密码和盐组合再通过特定加密算法得到对应的密码密文，存储在/etc/shadow中。

用户修改密码：

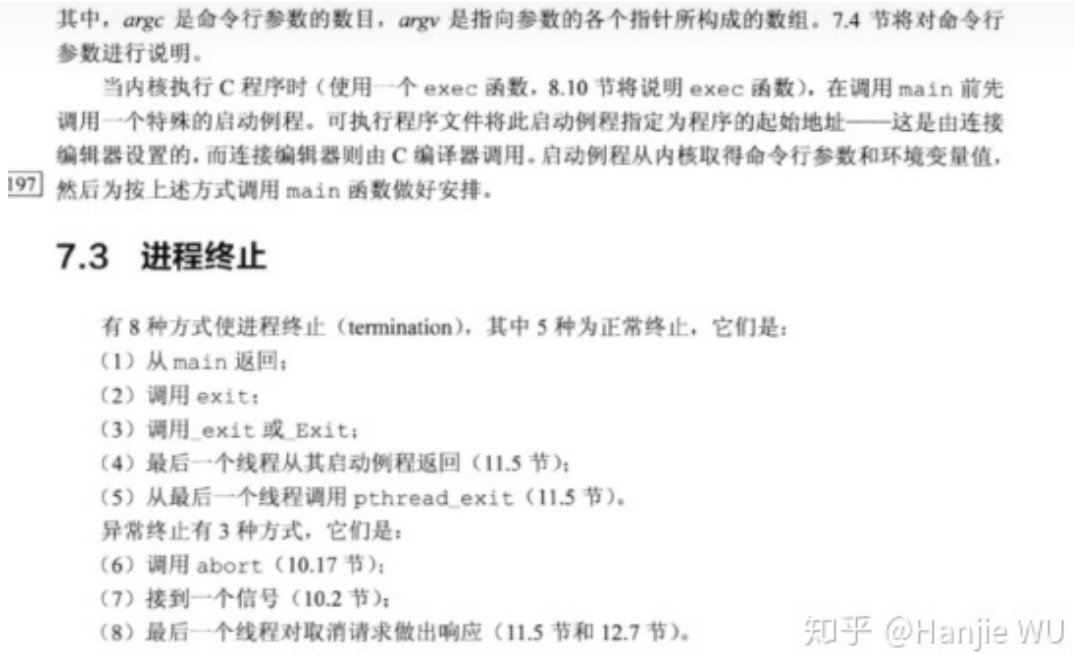
当你使用 passwd 命令改变了你的密码后，您的新密码存储在文件 /etc/shadow 中。作为一个普通用户，出于安全原因你没有读或写访问这个文件的权限，但是当你改变你的密码时，你需要拥有对这个文件

的权限。当执行一个启用了 SUID 的程序，继承了程序所有者的权限。当steve用户执行passwd命令的时候。Shell会fork出一个子进程，此时进程的EUID还是steve，然后exec程序/usr/bin/passwd。exec会根据/usr/bin/passwd的SUID位会把进程的EUID设成root, 此时这个进程都获得了root权限, 得到了读写/etc/shadow文件的权限, 从而steve用户可完成密码的修改。 exec退出后会恢复steve用户的EUID为steve.这样就不会使steve用户一直拥有root权限。

登录过程：

Init进程调用一次fork，生成的子进程则exec getty程序，getty输出“login：”之类的信息。用户键入用户名后，getty将会触发login程序，login程序在有了用户名之后, 它调用getpwnam来获取相应用户的口令文件登录项. 然后, login调用getpass来显示Password:，用户输入密码，程序同时读取密码. 然后login调用crypt来加密密码并把加密的结果与shadow密码文件的pw_passwd域比较; 若发生错误, login调用exit, 重启程序。正确登录后，login程序将当前的工作目录设置为用户的起始目录，调用chown更改终端的所有权，获得环境变量值等。

程序启动和终止：



其中，*argc* 是命令行参数的数目，*argv* 是指向参数的各个指针所构成的数组。7.4 节将对命令行参数进行说明。

当内核执行 C 程序时（使用一个 exec 函数，8.10 节将说明 exec 函数），在调用 main 前先调用一个特殊的启动例程。可执行程序文件将此启动例程指定为程序的起始地址——这是由连接编辑器设置的，而连接编辑器则由 C 编译器调用。启动例程从内核取得命令行参数和环境变量值，然后为按上述方式调用 main 函数做好安排。

197

7.3 进程终止

有 8 种方式使进程终止（termination），其中 5 种为正常终止，它们是：

（1）从 main 返回；

（2）调用 exit；

（3）调用 _exit 或 _Exit；

（4）最后一个线程从其启动例程返回（11.5 节）；

（5）从最后一个线程调用 pthread_exit（11.5 节）。

异常终止有 3 种方式，它们是：

（6）调用 abort（10.17 节）；

（7）接到一个信号（10.2 节）；

（8）最后一个线程对取消请求做出响应（11.5 节和 12.7 节）。

C程序的存储空间布局

（1）**正文段**。这是由CPU执行的机器指令部分。通常正文段是可共享的，所以即使是频繁执行的程序（入文本编辑器、C编译器和shell等）在存储器中也只需有一个副本，另外正文段常常是只读的，以防止程序由于意外而修改其指令。

（2）**初始化数据段**。通常将此段称为数据段，它包含了程序中需明确的赋值的变量。例如C程序中任何函数之外的声明：int maxcount = [illegible]

其初值存放在初始化数据段中。

序开始执行之前，内核将此段中的数据初始化为0或空指针。例如long sum[1000]存放在非初始化数据段中。

（4）**栈**。自动变量以及每次函数调用时所需保存的信息都存放在此段中。

（5）**堆**。通常在堆中进行动态存储分配。 一般由程序员分配释放， 若程序员不释放，程序结束时可能由OS回收 。

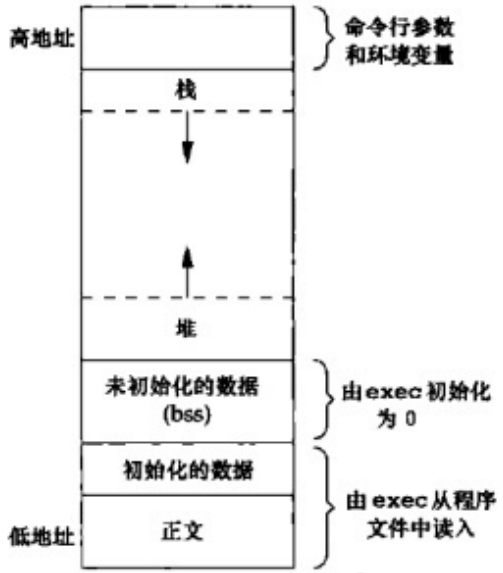


图 7-6 典型的存储空间安排

静态库

由于每个使用静态库的应用程序都需要拷贝所用函数的代码，所以静态链接的文件会比较大。

共享库

使得可执行文件中不再需要包含公用的库函数，只需要在所有进程可以存取的存储区中保留一个函数的副本，程序第一次执行或调用函数时，用动态链接方法链接此函数。这减少了可执行文件的长度，但增加了一些运行时间开销（第一次调用或被执行时）。共享库的另一个优点是可以用库函数新版本代替老版本而无需再编译程序。

存储器分配

三个用于存储空间动态分配的函数：

void *malloc(size_t size);

void *calloc(size_t nmemb, size_t size);

void *realloc(void *ptr, size_t size);

针，此存储区中的初始值不确定。

2. calloc：为指定数量和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。

3. realloc：更改以前分配区的长度（增加或减少）。当增加长度时，可能需将以前分配区的内容移到另一个足够大的区域，以便在尾端提供增加的存储区，而新增区域内的初始值则不确定。

三个函数的返回值：若成功则返回非空指针，若出错则返回NULL.

由于malloc()的返回类型为void*,因而可以将其赋给任意类型的C指针。
malloc()返回内存块所采用的字节对齐方式，总是适宜于高效访问任何类型的C语言数据结构。在大多数硬件架构上，这实际意味着malloc是基于8字节或16字节边界来分配内存的。

若无法分配内存，则malloc()返回NULL，并设置errno以返回错误信息。虽然分配内存失败的可能性很小，但所有对malloc()以及后续提及的相关函数的调用都应对返回值进行错误检查。

free()函数释放ptr参数所指向的内存块，该参数一般是之前由malloc()或其他堆内存分配函数返回的地址。

void free(void *ptr);

free()函数会将释放的这块内存添加到空闲内存列表中，供后续的malloc()函数循环使用。

如果传给free()的是一个空指针，那么函数将什么都不做。（换句话说，给free()传入一个空指针并不是错误代码。）

在调用free()后对参数ptr的任何使用，例如将其再次传递给free()，将产生错误，并可能导致不可预知的后果。所以应对参数做检查。

命令行参数

通过exec函数传递给新程序。

内部命令和外部命令

内部命令实际上是shell程序的一部分，其中包含的是一些比较简单的unix系统命令，这些命令由shell程序识别并在shell程序内部完成运行，通常在unix系统加载运行时shell就被加载并驻留在系统内存中。

存中，而是在需要时才将其调用内存

环境变量：

环境变量相当于给系统，用户或应用程序设置一些参数。Linux中环境变量包括系统级和用户级，系统级的环境变量是每个登录到系统的用户都要读取的系统变量，而用户级的环境变量则是该用户使用系统时加载的环境变量。HOME：指定用户的主工作目录（即用户登陆到Linux系统中时，默认的目录）SHELL：指当前用户用的是哪种Shell。每一个进程中都有一份所有环境变量构成的一个表格，即当前进程可以直接使用这些环境变量。环境表也是一个字符指针数组。其中每个指针都包含一个以NULL结尾的字符串的地址，指向各环境参数。环境表的地址存储在environ全局变量中。

进程标识：

每个进程都有一个非负整型表示的唯一进程ID，虽然唯一但可复用。当进程结束后，该进程ID可以被使用。0号进程是调度进程，并不执行任何磁盘的程序，只负责调度。1号进程是init进程，读取系统有关的初始化文件，并将系统引导到一个状态。2号进程是页守护进程，负责支持虚拟存储器系统的分页操作

头文件：#include<unistd.h>

函数定义：int fork(void);

返回值：子进程中返回0，父进程中返回子进程ID，出错返回-1

函数说明：

一个现有进程可以调用fork函数创建一个新进程。由fork创建的新进程被称为子进程(childprocess)。fork函数被调用一次但返回两次。两次返回的唯一区别是子进程中返回0值而父进程中返回子进程ID。子进程是父进程的副本，它将获得父进程数据空间、堆、栈等资源的副本。注意，子进程持有的是上述存储空间的“副本”，这意味着父子进程间不共享这些存储空间，它们之间共享的存储空间只有正文段。

Vfork()函数，（建立一个新的进程）

相关函数wait，execve

头文件#include<unistd.h>

定义函数pid_tfork(void)

vfork（）会产生一个新的子进程，其子进程会复制父进程的数据与堆栈空间，并继承父进程的用户代码，组代码，环境变量、已打开的文件代码、工作目录和资源限制等。返回值：如果vfork（）成功则在父进程会返回新建立的子进程代码（PID），而在新建立的子进程中则返回0。如果vfork失败则直接返回-1，失败原因存于errno中。

vfork与fork主要有三点区别:

1.fork():子进程拷贝父进程的数据段，堆栈段

2.vfork():子进程与父进程共享数据段

3.fork()父子进程的执行次序不确定,vfork保证子进程先运行,在调用exec或exit之前与父进程数据是共享的,在它调用exec或exit之后父进程才可能被调度运行。如果在调用这两个函数之前子进程依赖于父进程的进一步动作,则会导致死锁。

现在的所有unix变量都使用一种写拷贝的技术(copyonwrite)，它使得一个普通的fork调用非常类似于vfork.因此vfork变得没有必要.

而且内核将它们的访问权限改为只读，如果父进程和子进程中的任一个试图修改这些区域，则内核只为修改区域的那块内存制作一个副本。

wait和waitpid函数：

主要用于挂起正在运行的进程进入等待状态，直到有一个子进程终止。

1.若所有子进程都在运行，则阻塞，直至有子进程终止。

2.若有一个子进程已终止，则返回该子进程的 PID 并通过 status 参数 （若非 NULL）输出其终止状态。

3.若没有需要等待的子进程，则返回 -1，置 error 为 ECHILD。

exec函数：

exec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，栈段。

其中execlp，以文件名作为参数，如果文件名参数包含/视为路径名，否则按PATH环境变量，在它所指定的各目录中搜寻可执行文件。

RUID, 用于在系统中标识一个用户是谁，当用户使用用户名和密码成功登录后一个UNIX系统后就唯一确定了他的RUID.

EUID, 用于系统决定用户对系统资源的访问权限，通常情况下等于RUID。

SUID，用于对外权限的开放。和RUID及EUID是用一个用户绑定不同，它是跟文件绑定。

更改用户ID和组ID：

当执行设置了设置用户ID位的文件时，内核将进程的有效用户ID位临时更改为文件的所有者ID，进程执行完文件后需要将有效用户ID位恢复回原来的有效用户ID。这时使用保存设置用户ID来恢复。

图 8-18 总结了更改这 3 个用户 ID 的不同方法。

ID	exec		setuid(*uid*)	
	设置用户 ID 位关闭	设置用户 ID 位打开	超级用户	非特权用户
实际用户 ID	不变	不变	设为 *uid*	不变
有效用户 ID	不变	设置为程序文件的用户 ID	设为 *uid*	设为 *uid*
保存的设置用户 ID	从有效用户 ID 复制	从有效用户 ID 复制	设为 *uid*	不变

图 8-18　更改 3 个用户 ID 的不同方法

进程组

进程只能为它自己或它的子进程设置进程组ID,在它的子进程调用了exec函数之后就不能再改变该子进程的进程组id。只要某个进程组有一个进程，该进程组就存在。

Pid_t getpgrp(void);

Pid_t getpgid(pid_t pid);若pid为0，则返回调用的进程组ID

Int setpgid(pid_t pid,pid_t pgid);将pid进程的进程组ID设置为pgid.如果这两个参数相等，pid指定的进程变成进程组组长，为0，则使用调用者的进程ID.pgid为0，pid指定的进程id将用作进程组id。

会话

会话是一个或多个进程组的集合。

pid_t setsid(void); //若成功则返回进程组ID，出错则返回-1.

1.如果调用此函数的进程不是一个进程组组长，则此函数就会创建一个新会话，结果将发生下面三件事：

的进程）此时，该进程是新会话中的唯一进程。

b.该进程称为一个新进程组的组长进程，新进程组ID是该调用进程的进程ID。

c.该进程没有控制终端，如果在调用setsid之前该进程有一个控制终端，那么这种联系也会被中断。

2.如果该调用进程已经是一个进程组的组长，则此函数返回出错。为了保证不会发生这种情况，通常先调用fork，然后使其父进程终止，则子进程则继续。因为子进程继承了父进程的进程组ID，而其进程ID是新分配的，两者不可能相等，保证了子进程不会是一个进程组的组长。

控制终端

1.一个会话可以有一个控制终端（controlling terminal），通常会话的第一个进程打开一个终端（终端设备或伪终端设备）后，该终端就成为该会话的控制终端。

2.建立与控制终端连接的会话首进程被称为控制进程。（controlling process）

3.一个会话中的几个进程组可被分成一个前台进程组以及一个[illegible]程组。

process group），会话中的其他进程组则为后台进程组。

5.无论何时进入终端的中断键（ctrl+c）或推出键（ctrl+\），就会将中断信号发送给前台进程组的所有进程。

6.如果终端接口检测到调制解调器（或网络）已经断开，则将挂断信号发送给控制进程。

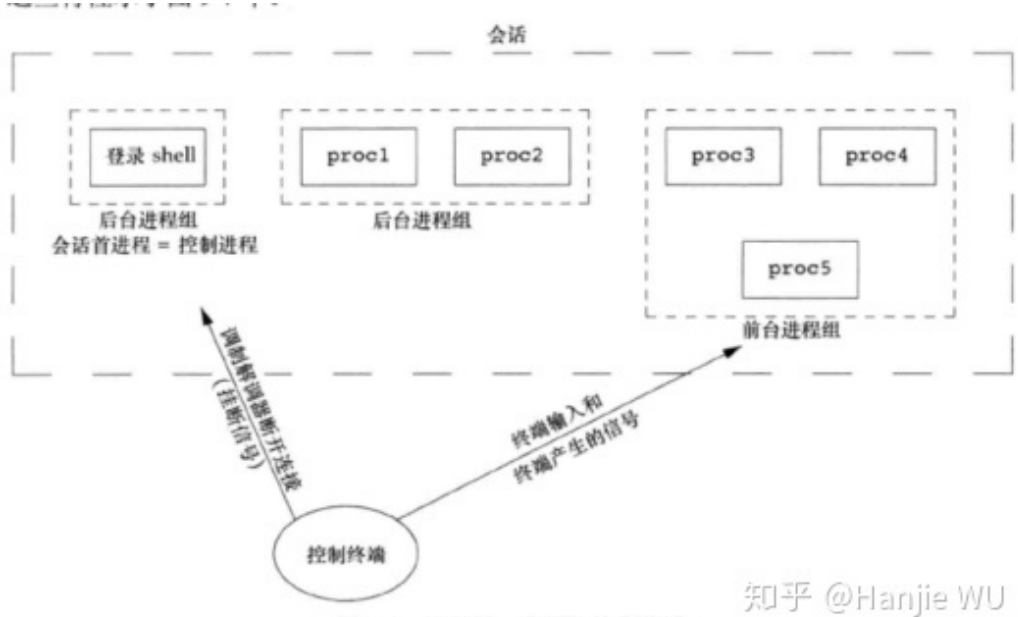


图 9-7　进程组、会话和控制终端

作业控制

作业控制是伯克利在1 9 8 0年左右加到U N I X的一个新特性。它允许在一个终端上起动多个作业(进程组)，控制哪一个作业可以存取该终端，以及哪些作业在后台运行。作业控制要求三种形式的支持：

(1) 支持作业控制的s h e l l。

(2) 内核中的终端驱动程序必须支持作业控制。

(3) 必须提供对某些作业控制信号的支持。

僵尸进程

由于父进程没有调用wait()或者没有设置SIGCHLD信号处理函数，会导致僵尸进程出现。子进程终止时，由于需要保存它的终止状态以备父进程调用wait（），因此子进程在进程表中的登记项不会立刻释放，仍然保持与父进程的连接直到父进程正常终止或者调用wait（）为止。即尽管子进程不在活跃，但仍在系统中，此时它就是所谓的僵尸进程。

解决方法：

SIGCHLD信号后，调用wait为子进程清理状态。

2. 调用两次fork，让第一次fork的子进程退出，使第二次fork的进程成为孤儿进程，被init进程接管，由init进程来负责清理孤儿进程。

3. 把父进程杀死，让僵尸进程成为孤儿进程

4. 在父进程创建子进程之前，就申明不会对子进程的exit有关注，这样子进程退出后就会直接被系统回收资源，不用等待父进程的操作。具体调用signal（SIGCHLD，SIG_IGN)

孤儿进程：

一个父进程已经终止的进程。孤儿进程将被1号init进程接管，init检查进程成为孤儿进程的父进程。

信号

信号用于很多复杂的应用程序中。理解进行信号处理的原因和方式对于高级 UNIX 程序设计极其重要。

内核“在此信号发生时,请执行下列操作”。

[core文件：在程序崩溃时，在指定目录下生成一个core文件，这个文件用来调试]

SIGALRM	定时器超时（alarm）		•	•	•	•	终止
SIGBUS	硬件故障		•	•	•	•	终止+core
SIGCANCEL	线程库内部使用					•	忽略
SIGCHLD	子进程状态改变		•	•	•	•	忽略
SIGCONT	使暂停进程继续		•	•	•	•	继续/忽略
SIGEMT	硬件故障			•	•	•	终止+core
SIGFPE	算术异常	•	•	•	•	•	终止+core
SIGFREEZE	检查点冻结					•	忽略
SIGHUP	连接断开		•	•	•	•	终止
SIGILL	非法硬件指令	•	•	•	•	•	终止+core
SIGINFO	键盘状态请求			•		•	忽略
SIGINT	终端中断符	•	•	•	•	•	终止
SIGIO	异步 I/O			•	•	•	终止/忽略
SIGIOT	硬件故障			•	•	•	终止+core
SIGJVM1	Java 虚拟机内部使用					•	忽略
SIGJVM2	Java 虚拟机内部使用					•	忽略
SIGKILL	终止		•	•	•	•	终止
SIGLOST	资源丢失					•	终止
SIGLWP	线程库内部使用		•			•	终止/忽略
SIGPIPE	写至无读进程的管道		•	•	•	•	终止
SIGPOLL	可轮询事件（poll）				•	•	终止
SIGPROF	梗概时间超时（setitimer）			•	•	•	终止
SIGPWR	电源失效/重启动				•	•	终止/忽略
SIGQUIT	终端退出符		•	•	•	•	终止+core
SIGSEGV	无效内存引用	•	•	•	•	•	终止+core
SIGSTKFLT	协处理器栈故障				•		终止
SIGSTOP	停止		•	•	•	•	停止进程
SIGSYS	无效系统调用		XSI	•	•	•	终止+core
SIGTERM	终止	•	•	•	•	•	终止
SIGTHAW	检查点解冻					•	忽略
SIGTHR	线程库内部使用			•			忽略
SIGTRAP	硬件故障		XSI	•	•	•	终止+core
SIGTSTP	终端停止符		•	•	•	•	停止进程
SIGTTIN	后台读控制 tty		•	•	•	•	停止进程
SIGTTOU	后台写向控制 tty		•	•	•	•	停止进程
SIGURG	紧急情况（套接字）		•	•	•	•	忽略
SIGUSR1	用户定义信号		•	•	•	•	终止
SIGUSR2	用户定义信号		•	•	•	•	终止
SIGVTALRM	虚拟时间闹钟（setitimer）		XSI	•	•	•	终止
SIGWAITING	线程库内部使用					•	忽略
SIGWINCH	终端窗口大小改变			•	•	•	忽略
SIGXCPU	超过 CPU 限制（setrlimit）		XSI	•	•	•	终止或终止+core
SIGXFSZ	超过文件长度限制（setrlimit）		XSI	•	•	•	终止或终止+core
SIGXRES	超过资源控制					•	忽略

(1) 忽略此信号（SIG_IGN）。大多数信号都可使用这种方式进行处理,但有两种信号却决不能被忽略。它们是:SIGKILL和SIGSTOP。这两种信号不能被忽略的原因是:它们向超级用户提供一种使进程终止或停止的可靠方法。另外,如果忽略某些由硬件异常产生的信号(例如非法存储访问或除以0),则进程的行为是未定义的。

(2) 捕捉信号。为了做到这一点要通知内核在某种信号发生时,调用一个用户函数。在用户函数中,可执行用户希望对这种事件进行的处理。例如,,如果进程创建了临时文件,那么可能要为SIGTERM信号编写一个信号捕捉函数以清除临时文件(kill命令传送的系统默认信号是终止信号)。

(3) 执行系统默认动作(SIG_DFL)

1、信号的生成既可以是同步的，也可以是异步的。同步信号与程序中的某个具体操作相关并且在那个操作进行的同时生成。多数程序错误生成的信号是同步的。由进程显式请求而生成的给自己的信号也是同步的。

异步信号是进程之外的时间生成的信号。一般外部事件总是异步的生成信号。异步信号可在进程运行中的任意时刻产生，进程无法预期信号到达的时刻，他所能做的只是告诉内核假如有信号生成时应该采取什么行动。

2、SIGINT中断信号，SIGQUIT退出信号，SIGSEGV非法段存

signal(SIGINT,SIG_IGN)。

4、中断信号通常是编码2。

不可靠信号的主要问题是：

进程每次处理信号后，就将对信号的响应设置为默认动作。在某些情况下，将导致对信号的错误处理；因此，用户如果不希望这样的操作，那么就要在信号处理函数结尾再一次调用signal()，重新安装该信号。

信号可能丢失，如果在进程对某个信号进行处理时，这个信号发生多次，对后到来的这类信号不排队，那么仅传送该信号一次，即发生了信号丢失。

因此，早期unix下的不可靠信号主要指的是进程可能对信号做出错误的反应以及信号可能丢失。

慢系统调用：

在调用过程中可能会永久阻塞的调用，包括以下情况：

1.调用read（）时数据不出现；调用write（）时数据不能被立刻接收；

3.读写一个具有强制锁的文件；

4.某些ioctl（）操作；

5.某些进程间通信函数

6.pause函数和wait函数

可重入函数

一个可重入的函数简单来说就是可以被中断的函数，也就是说，可以在这个函数执行的任何时刻中断它，转入OS调度下去执行另外一段代码，而返回控制时不会出现什么错误；而不可重入的函数由于使用了一些系统资源，比如全局变量区，中断向量表等，所以它如果被中断的话，可能会出现问题，这类函数是不能运行在多任务环境下的。

线程

线程概念及标识

线程是程序执行时的最小单位，是CPU调度和分派的基本单位

概念：线程包含了表示进程内执行环境必需的信息，其中包括进程中标识线程的线程ID、一组寄存器值、栈、调度优先级和策略、信号屏蔽字、errno变量以及线程私有数据。进程的所有信息对该进程的所有线程都是共享的，包括可执行的程序文本、程序的全局内存和堆内存、栈以及文件描述符。

线程标识：就像每个进程有一个进程ID一样，每个线程也有一个线程ID。进程ID在整个系统中是唯一的，但线程ID不同，线程ID只在它所属的进程环境中有效。线程ID用pthread_t数据类型来表示，实现的时候可以用一个结构来代表pthread_t数据类型，可移植的操作系统实现不能把它作为整数处理。

线程终止

单个线程可以通过三种方式退出，在不终止整个进程的情况下停止它的控制流。

（1）线程只是从启动例程中返回，返回值是线程的退出码。

（2）线程可以被同一进程中的其他线程取消。

（3）线程调用pthread_exit。

关于线程同步，pthread提供了五种最基本的机制，分别是：

1.互斥量：

互斥量从本质上说是一把锁，在访问共享资源前对互斥量进行加锁，在访问完成后释放互斥量上的锁。对互斥量进行加锁后，任何其他试图再次对互斥量加锁的线程将会被阻塞直到当前线程释放该互斥锁。互斥锁可以确保同一时间只有一个线程访问数据：

//可以设置属性

```
int pthread_mutex_init(pthread_mutex_t* restrict mutex,const pthread_
int pthread_mutex_destroy(pthread_mutex_t* mutex);
```

互斥锁操作上有下面几种，包括加锁，解锁和尝试加锁(非阻塞行为):

```
int pthread_mutex_lock(pthread_mutex_t* mutex);
int pthread_mutex_unlock(pthread_mutex_t* mutex);
int pthread_mutex_trylock(pthread_mutex_t* mutex);
```

对同一个互斥量两次加锁。两个线程互相请求另一个线程拥有的资源。使用pthread_mutex_trylock来加锁，使用这个函数尝试对一个互斥量加锁时，成功则前进，否则会先释放自己拥有的资源，过一段时间再尝试。

2.读写锁：读写锁与互斥锁类似，不过读写锁允许更高的并行性。读写锁可以有三种状态：

l 读模式下加锁状态

l 写模式下加锁状态

l 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。当这个线程处于写加锁状态时，任何尝试对这个锁加锁的线程会被阻塞。当读写锁在读加锁状态时，所有试图以读模式加锁的线程可以得到访问权，所有以写模式加锁的线程会被阻塞。

//在使用之前必须初始化，在释放他们底层的内存前必须销毁。

```
int pthread_rwlock_init(pthread_rwlock_t* restrict rwlock const pthr
int pthread_rwlock_destroy(pthread_rwlock_t* rwlock);
```

//读写锁加锁，解锁。

```
int pthread_rwlock_rdlock(pthread_rwlock_t* rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t* rwlock);
int pthread_rwlock_unlock(pthread_rwlock_t* rwlock);
```

//读写锁尝试加锁

```
int pthread_rwlock_tryrdlock(pthread_rwlock_t* rwlock);
int pthread_rwlock_trywrlock(pthread_rwlock_t* rwlock);
```

3.条件变量：

传递给pthread_cond_wait的互斥量对条件进行保护，调用者把锁住的互斥量传给函数。函数把调用线程放到等待条件的线程列表上，然后对互斥量解锁。条件变量会首先判断条件是否满足，如果不满足的话那么会释放当前这个配对的锁，如果一旦触发的话那么会尝试加锁。

首先，举个例子：在应用程序中有连个线程thread1，thread2，thread3和thread4，有一个int类型的全局变量iCount。iCount初始化为0，thread1和thread2的功能是对iCount的加1，thread3的功能是对iCoun

iCount=0。

如果使用互斥量，线程代码大概应是下面的样子：

```
//thread1/2:
   while (1)
       { pthread_mutex_lock(&mutex);
         iCount++;
         pthread_mutex_unlock(&mutex);}
//thread4:
   while(1)
      { pthead_mutex_lock(&mutex);
        if (100 <= iCount)
        { printf("iCount >= 100\r\n");
        iCount = 0;
        pthread_mutex_unlock(&mutex);}
        else
        {pthread_mutex_unlock(&mutex);}}
```

在上面代码中由于thread4并不知道什么时候iCount会大于等于100，所以就会一直在循环判断，但是每次判断都要加锁、解锁(即使本次并没有修改iCount)。这就带来了问题一，CPU浪费严重。所以在代码中添加了sleep(),这样让每次判断都休眠一定时间。但这由带来的第二个问题，如

上面的两个问题，可以使用条件变量来解决。

首先看一下使用条件变量后，线程代码大概的样子：

```
//thread1/2:
  while(1)
       { pthread_mutex_lock(&mutex);
         iCount++;
         pthread_mutex_unlock(&mutex);
         if (iCount >= 100)
           {pthread_cond_signal(&cond);}}
//thread4:
   while (1)
       {pthread_mutex_lock(&mutex);
        while(iCount < 100){
        pthread_cond_wait(&cond, &mutex);}
        printf("iCount >= 100\r\n");
        iCount = 0;
        pthread_mutex_unlock(&mutex);}
```

从上面的代码可以看出thread4中，当iCount < 100时，会调用pthread_cond_wait。而pthread_cond_wait在上面已经讲到它会释放mutex，然后等待条件变为真返回。当返回时会再次锁住mut

（即thread4），这样当iCount一到大于等于100就会去唤醒thread4。从而不致出现iCount很大了，thread4才去处理。

5. 自旋锁：

类似与互斥量，但是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阻塞状态。适用于锁被持有的时间短，线程不希望在重新调度上花费太多时间。

6. 屏障：

允许每个线程等待，直到所有合作线程到达某一点，然后从该点继续执行。

线程安全：

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。可重入的函数是线程安全的。

线程私有数据：

的常见的变量errno，它返回标准的出错信息。线程私有数据采用了一种被称为一键多值的技术，即一个键对应多个数值。访问数据时都是通过键值来访问，好像是对一个变量进行访问，其实是在访问不同的数据。使用线程私有数据时，首先要为每个线程私有数据创建一个相关联的键。在各个线程内部，都使用这个公用的键来指代线程数据，但是在不同的线程中，这个键代表的数据是不同的。操作线程私有数据的函数主要有4个：pthread_key_create（创建一个键），pthread_setspecific（为一个键设置线程私有数据），pthread_getspecific（从一个键读取线程私有数据），pthread_key_delete（删除一个键）。

守护进程：

守护进程也是精灵进程(daemon),是一种生存期较长的进程，它独立于控制终端，并且周期性地执行某种任务或者等待处理某些发生的事情，守护进程通常在系统引导装入时启动，在系统关闭时终止。

如何创建守护进程：

1. fork然后使得父进程退出。一方面shell认为父进程执行完毕，另外一方面子进程获得新的pid肯定不为进程组组长，这是setsid前提。

3.最好在这里再次fork。这样子进程不是会话首进程，那么永远没有机会获得控制终端。如果这里不fork的话那么会话首进程依然可能打开控制终端。

4.chdir将当前工作目录更改为根目录。父进程继承过来的当前目录可能mount在一个文件系统上。如果不切换到根目录，那么这个文件系统不允许unmount.

5.umask(0).因为我们从shell创建的话，那么继承了shell的umask.这样导致守护进程创建文件会屏蔽某些权限。

6.关闭不需要的文件描述符。可以通过_SC_OPEN_MAX来判断最高文件描述符(不是很必须).

高级I/O:

非阻塞I/O:

非阻塞I/O使我们可以发出open、read和write这样的I/O操作，并使这些操作永远不会阻塞。如果这些操作不能立即完成，则调用立即出错返回，表示该操作如继续执行将阻塞。对于一个给定的描述符，有两种为其指
方法：

（2）对于一个已经打开的描述符，则可以调用fcntl改变已经打开的文件属性，由该函数打开O_NONBLOCK标志。

异步I/O:

在数据拷贝时，进程不需要阻塞。

存储映射I/O:

存储映射I/O（Memory-mapped I/O）使一个磁盘文件与存储空间中的一个缓冲区相映射。于是当从缓冲区中取数据，就相当于读文件中的相应字节。与此类似，将数据存入缓冲区，则相应字节就自动地写入文件。这样就可以在不使用read和write的情况下执行I/O。为了使用这种功能，应首先告诉内核将一个给定的文件映射到一个存储区域中。这是由mmap函数实现的。

进程IPC：7种方式

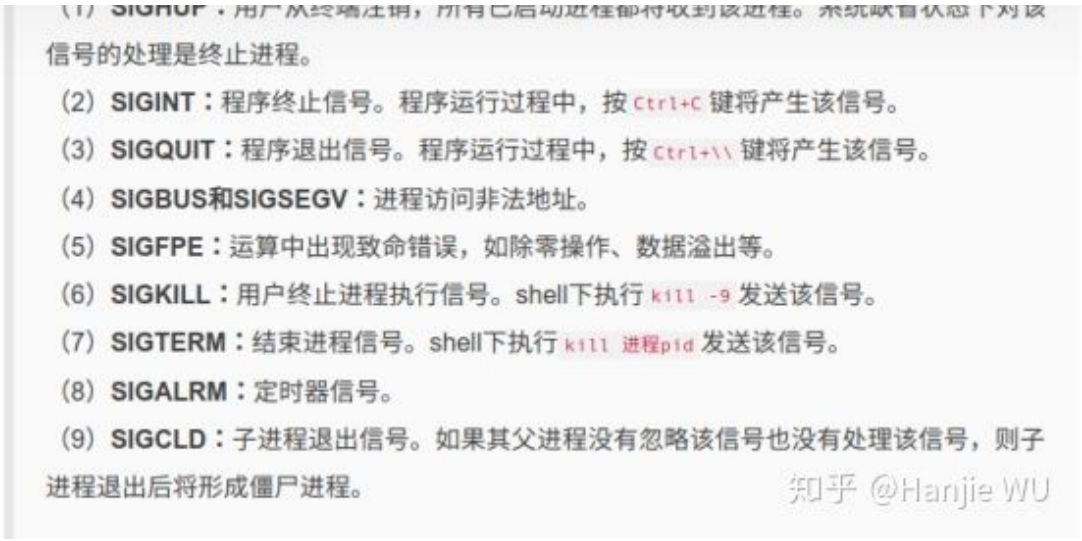
管道是半双工的，数据只能向一个方向流动；需要双方通信时，需要建立起两个管道。只能用于父子进程或者兄弟进程之间(具有亲缘关系的进程);

2. 有名管道(FIFO)

以有名管道的文件形式存在于文件系统中，这样，即使与有名管道的创建进程不存在亲缘关系的进程，只要可以访问该路径，就能够彼此通过有名管道相互通信，因此，通过有名管道不相关的进程也能交换数据。有名管道严格遵循先进先出(first in first out),对匿名管道及有名管道的读总是从开始处返回数据，对它们的写则把数据添加到末尾。

3. 信号(Signal)

信号是Linux系统中用于进程间互相通信或者操作的一种机制，信号可以在任何时候发给某一进程，而无需知道该进程的状态



4. 消息(Message)队列

消息队列是存放在内核中的消息链表，每个消息队列由消息队列标识符表示。另外与管道不同的是，消息队列在某个进程往一个队列写入消息之前，并不需要另外某个进程在该队列上等待消息的到达。

5. 共享内存(share memory)

使得多个进程可以可以直接读写同一块内存空间，是最快的可用IPC形式。是针对其他通信机制运行效率较低而设计的。

6. 信号量(semaphore)

同同步。

为了获得共享资源，进程需要执行下列操作：

（1）测试信号量

（2）若此信号量的值为正，则进程可以使用该资源。进程将信号量值减1，表示它使用了一个资源单位。

（3）若此信号量的值为0，则进程进入休眠状态，直至信号量值大于0。进程被唤醒后，它返回执行第1步。

POSIX信号量

POSIX信号量的两种形式：命名的和未命名的。它们的差异在于创建和销毁的形式上。未命名信号量只存在内存中，并要求能够使用信号的进程必须可以访问该内存。这意味着它们只能应用在同一进程中的线程或者是不同进程的线程，但是这些进程映射了相同的内存地址到自己的地址空间。相反，命名信号量可以通过名字访问，因此可以被任何知道它们名字的进程中的线程使用。

7. 套接字(socket)

就已经可以在不同一台计算机间通过网络连接计算机上的进程进行通信。

套接字：

IPC 套接字（即 Unix 套接字）给予进程在相同设备（主机）上基于通道的通信能力；而网络套接字给予进程运行在不同主机的能力，因此也带来了网络通信的能力。网络套接字需要底层协议的支持，例如 TCP（传输控制协议）或 UDP（用户数据报协议）。IPC 套接字依赖于本地系统内核的支持来进行通信；特别的，IPC 通信使用一个本地的文件作为套接字地址。套接字以流的形式被配置为双向的，并且其控制遵循 C/S（客户端/服务器端）模式：客户端通过尝试连接一个服务器来初始化对话，而服务器端将尝试接受该连接。假如万事顺利，来自客户端的请求和来自服务器端的响应将通过管道进行传输，直到其中任意一方关闭该通道，从而断开这个连接。

服务端：

```c
}

int main() {
  int fd = socket(AF_INET,     /* network versus AF_LOCAL */
          SOCK_STREAM, /* reliable, bidirectional: TCP */
          0);          /* system picks underlying protocol */
  if (fd < 0) report("socket", 1); /* terminate */

  /* bind the server's local address in memory */
  struct sockaddr_in saddr;
  memset(&saddr, 0, sizeof(saddr));          /* clear the bytes */
  saddr.sin_family = AF_INET;                /* versus AF_LOCAL */
  saddr.sin_addr.s_addr = htonl(INADDR_ANY); /* host-to-network endian */
  saddr.sin_port = htons(PortNumber);        /* for listening */

  if (bind(fd, (struct sockaddr *) &saddr, sizeof(saddr)) < 0)
    report("bind", 1); /* terminate */

  /* listen to the socket */
  if (listen(fd, MaxConnects) < 0) /* listen for clients, up to MaxConnects */
    report("listen", 1); /* terminate */

  fprintf(stderr, "Listening on port %i for clients...\n", PortNumber);
  /* a server traditionally listens indefinitely */
  while (1) {
    struct sockaddr_in caddr; /* client address */
    int len = sizeof(caddr);  /* address length could change */

    int client_fd = accept(fd, (struct sockaddr*) &caddr, &len);  /* accept blocks */
    if (client_fd < 0) {
      report("accept", 0); /* don't terminated, though there's a problem */
      continue;
    }

    /* read from client */
    int i;
    for (i = 0; i < ConversationLen; i++) {
      char buffer[BuffSize + 1];
      memset(buffer, '\0', sizeof(buffer));
      int count = read(client_fd, buffer, sizeof(buffer));
      if (count > 0) {
    puts(buffer);
    write(client_fd, buffer, sizeof(buffer)); /* echo as confirmation */
      }
    }
    close(client_fd); /* break connection */
  }  /* while(1) */
  return 0;
}
```

知乎

```c

void report(const char* msg, int terminate) {
  perror(msg);
  if (terminate) exit(-1); /* failure */
}

int main() {
  /* fd for the socket */
  int sockfd = socket(AF_INET,      /* versus AF_LOCAL */
              SOCK_STREAM,  /* reliable, bidirectional */
              0);           /* system picks protocol (TCP) */
  if (sockfd < 0) report("socket", 1); /* terminate */

  /* get the address of the host */
  struct hostent* hptr = gethostbyname(Host); /* localhost: 127.0.0.1 */
  if (!hptr) report("gethostbyname", 1); /* is hptr NULL? */
  if (hptr->h_addrtype != AF_INET)       /* versus AF_LOCAL */
    report("bad address family", 1);

  /* connect to the server: configure server's address 1st */
  struct sockaddr_in saddr;
  memset(&saddr, 0, sizeof(saddr));
  saddr.sin_family = AF_INET;
  saddr.sin_addr.s_addr =
     ((struct in_addr*) hptr->h_addr_list[0])->s_addr;
  saddr.sin_port = htons(PortNumber); /* port number in big-endian */

  if (connect(sockfd, (struct sockaddr*) &saddr, sizeof(saddr)) < 0)
    report("connect", 1);

  /* Write some stuff and read the echoes. */
  puts("Connect to server, about to write some stuff...");
  int i;
  for (i = 0; i < ConversationLen; i++) {
    if (write(sockfd, books[i], strlen(books[i])) > 0) {
      /* get confirmation echoed from server and print */
      char buffer[BuffSize + 1];
      memset(buffer, '\0', sizeof(buffer));
      if (read(sockfd, buffer, sizeof(buffer)) > 0)
    puts(buffer);
    }
  }
  puts("Client done, about to exit...");
  close(sockfd); /* close the connection */
  return 0;
```

编辑于 2019-12-25 19:16

UNIX环境高级编程（书籍） Unix 入门 进程

推荐阅读

UNIX环境高级编程（APUE）第1章（CentOS8环境）-...

CoachHe

Unix网络编程的5种I/O模型

这一篇介绍一下Unix网络编程种I/O模型，在介绍这5中I/O之前先介绍一下Unix网络编程同步/异步和阻塞/非阻塞以及户空间和内核空间的概念 用间和内核空间在Linux/Unix中

可乐船长2... 发表于网络

还没有评论

写下你的评论...

▲ 赞同 4 ▼ 添加评论 分享