# 第一章 UNIX基础知识

本章从宏观角度介绍了UNIX相关的基础知识如：内核介绍、体系结构、登录方式、输入输出、错误输出、用户标识、信号量等方面。

~一个操作系统基本由内核、系统调用、共用函数库、交互接口程式、应用程序等构成。linux是GNU的内核。

没启动一个视窗就会运行一个或多个shell，shell经历了各种版本的迭代，从开始的shell到bash、csh、ksh、tcsh等，最常用的就是以上几个。

~unix系统的文件系统是文件和目录组成的树形结构，遵循万物都是文件的观点，所有系统信息都以文件存储。

1.4列出了一些文件和目录的命名规则和默认格式、工作目录、起始目录等，并且给出了一个使用apue.h标准库实现ls的简要实现。

~文件描述符是一个非负整数，kernel标识一个特定进程正在访问的文件。每运行新的程序，shell就会打开三个文件描述符：标准输入、标准输出、标准错误，无特殊处理的文件描述符都指向终端显示，也可通过管道、重定向等输出到文件或下一个shell程序。

~程序在UNIX中指可执行文件，内核使用exec函数将其加载入内存，并执行。

~进程是程序执行的实例，使用processID来标识，为一个长整型。，控制进程主要用fork、exec、waitpid这三个函数。1.6提供了一个简单的shell程序的基本实现来从标准输入实现执行命令。

一个进程可以有一个或多个控制线程，多个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。多进程也用ID来标识，但是其生效范围只是当前进程。多进程访问共享数据时注意采取同步措施来避免不一致性。

~UNIX函数出错时，往往会返回一个负值，大多数指向指针的函数会在出错时指向null，1.7介绍了一些出错处理函数：errno是个符号，他可以是个包含出错编号的整数也可以是个返回出错编号的函数。多线程每个线程都有自己的errno。仅当函数的返回值指明出错时才检验其值，任何函数都不会将errno置为0。介绍了strerror、perror等函数的用法。一般编程时，不直接使用strerror、perror，而是调用出错函数（附录B）。出错一般定义为：致命与非致命两种，顾名思义。非致命的典型操作是延迟一段时间后重试。

~用户标识是一个整形数，（UserID）ID为0的是root用户。可以使用getuid（）和getgid（）返回用户的UID和GID

用户可以拥有大于16个附属组已获得相应的组权限。

~信号用于进程间通信，告诉进程发生了某种情况。进程有3中信号处理模式：忽略信号、系统默认方法处理、提供一个函数。

~linux时间值分为日历时间和进程时间。日历时间顾名思义使用UTC，进程时间分为3种：时钟时间、用户CPU时间和系统cpu时间。一般把后两种的和称为cpu时间。

~系统调用是直接进入内核的一些入口，从应用角度，可以理解为系统调用就是联系内核的C函数。库函数是比系统调用更上层的，可能使用多个系统调用也可能不使用的一些函数，比如printf（）、strcpy（）等。

# 第二章 UNIX标准及实现

本章介绍了一些需要了解的标准化组织所制定的UNIX规范标准，以及实现方法。

~UNIX使用了一些标准化组织制定的标准：ISO C、IEEE POSIX、Single UNIX Specification、FIPS等，具体标准定义的头文件和接口请阅P25、P26.

~UNIX有超级多的具体实现，如：SVR4、BSD、FreeBSD、Linux、macOS High Sierra、Solaris等。具体介绍请阅P28

~UNIX中不同的标准有不同的限制，（规范化）一般限制有三种：编译时限制（头文件）、与文件或目录无关的运行时限制（sysconf函数）、与文件和目录有关的运行时限制（pathconf、fpathconf）。

关于具体的各大标准的对于限制的具体内容，请阅P30-P33。

关于运行时限制中的三个函数：sysconf、pathconf、fpathconf，具体内容请阅P34-P35.

P35构建了一个打印个pathconf和sysconf的程序。

前面说的是确定的运行限制。在不确定的运行限制中举出两个例子：P40为路径名动态的分配空间、P42最大打开文件数，但是我看的不是很明白，需要读第二遍。

~如果只想使用某一个标准的定义，则可以通过制定功能测试宏实现（\_POSIX\_C\_SOURCE \_XOPEN\_SOURCE）使用方法是在cc时加上 -D 参数。

~P47介绍了一些基本系统数据类型，（如：clock\_t uid\_t）等。大多数基本系统数据类型都是以 \_t 结尾的。

~一般来说各标准之间很少有冲突，如果有，以ISO C标准为首。

# 第三章 文件I/O

从本章开始，将讲述UNIX系统中的重要的组成模块。主要详细说了文件IO函数

~文件描述符是一个非负整数，所有打开的文件都使用文件描述符引用。UNIX将0定义为标准输入（STDIN\_FILENO），1为标准输出（STDOUT\_FILENO），2为标准错误（STDERR\_FILENO）。 UNIX允许一个进程能够同时打开的文件数理论上是有限的。

~open（）或openat（）可以打开或创建一个文件。其根据不同的参数可以实现各种权限的文件操作，详情请阅P50。由open和openat函数返回的文件描述符一定是最小的魏永描述符数值。

~create（）函数可创建一个新文件。其功能与open有等效。（早期版本中open不可以打开不存在的文件，后期加上了O\_CREAT和OTRUNC选项，可以创建新文件。）creat（）只能以只写方式打开文件。

~close（）函数可以关闭一个打开的文件。但是多数程序使用终止进程来关闭打开的文件而不是调用close（）。

~lseek（）函数可以显式的为一个打开文件设置偏移量，关于偏移量请阅P53，P54给出了一些常用的偏移量操作。

~read（）函数，read函数从打开文件中读数据，返回值为独到的字节数，若已到文件尾则返回0，出错返回-1.

~write（）函数可以向已打开的文件写数据。这种IO效率其实不是很高，后面会有更好的方法。

~多个进程可以打开同一个文件，由于一致性要求原子性操作。

~内核使用3种数据结构标识打开文件，分为 文件状态标志、当前文件偏移量、v节点指针。（v节点表项又分为：v节点信息和v\_data，v\_data又分为：i节点信息、当前文件长度和i\_vnode）

如果两个独立进程各自打开了同一个文件则如P61所示有如下关系。

~原子操作是多进程读写同一文件的前提。UNIX拓展标准中定义了pread（）和pwrite（）两个函数，用来原子性的读和写。pread相当于调用lseek后调用read。pwrite相当于调用lseek后调用了write。当然，还是有一些区别的，请阅P63.

~dup（）和dup2（）可以用来复制一个现有文件的文件描述符。dup返回可用最小文件描述符，dup2可以使用fd2参数执行新文件描述符。这两个函数都是原子操作。在某些时候，可以使用fcntl（）函数等效实现dup和dup2，具体方法和区别请阅P64.

~内核使用延迟写技术时，需要保证缓冲层和实际写入存储介质的信息相同，提供了3种函数：sync()（将所有修改过得块缓冲区排入写队列，不等待执行完成）fsync（）（将指定的fd文件编入写队列，等到写入完成时才返回），fdatasync（）（类似于fsync，但是只写入文件的数据部分，不管文件的属性）具体请阅P65

~fcntl（）函数前面说过，可以改变已经打开的文件的属性。一般使用fcntl（）进行一下几种操作：复制已有描述符、获取/设置文件描述符标志、获取/设置状态标志、获取/设置异步IO所有权、获取/设置记录锁，详情请阅P65.P66-P69展示了fcntl的几个常用实例和常规错误。

~P70简单介绍了ioctl（）函数的诞生原因和意义，具体实现请阅18.12节和19.7节。

~ /dev/fd 新系统都提供这个目录，里面放的是0、1、2、3等的文件，打开其中一个文件n就相当于复制文件描述符n，其具体使用方法请阅P71-P72。

# 第四章 文件和目录

~4.2介绍了一些查看文件信息结构的函数，stat（）（给出pathname，返回此命名文件有关的信息结构）、fstat（根据文件描述符fd，返回已打开的文件信息结构）、lstat（返回符号链接本身的信息结构）、fstatat（一个相对当前打开目录的文件统计信息）。stat用的最多的就在ls -l里。

~4.3介绍了文件的类型，这是基础，包括：普通文件、目录文件、块特殊文件、字符特殊文件、FIFO、套接字、符号链接。其具体属性请阅P76

P76-P78给出了打印文件属性的几个典型实例。

~一个进程相关联的ID有6个常用的：实际用户ID、实际组ID（实际上使用者）、有效用户ID、有效组ID（文件访问权限检查）、保存的设置用户ID、保存的设置组ID。（exec函数保存）。通常每个文件都有一个所有者和组所有者，所有者有stat结构中的st\_uid指定，组所有者则由st\_gid指定。具体的ID间的关系请阅P78

~在stat的st\_mode中包含了对文件的访问权限位，所有文件都有访问权限。每个文件有9个访问权限位，可分为三类：用户、组、其他，每类又分为：读、写、执行三种。具体权限含义和具体规则请阅P79。

~由open、creat、mkdir创建的文件和目录的权限如下：用户ID是进程的有效用户ID，组ID可以是两种情况：1.可以是进程有效组ID。2.可以是所在目录的组ID。

~对于一个文件的用户和组权限测试，可以使用access（）和faccessat（）两个函数进行访问权限测试。access通过制定pathname，faccessat通过指定文件描述符fd实现的。P81-P82给出了access的工作实例。

~前面介绍了文件权限相关内容，本节讲述了函数umask（）设置创建权限屏蔽字的用法。umask的参数cmask对应了文件的9个访问权限，置1则屏蔽该种权限。SUS标准规定了UNIX需要支持umask命令来查看当前默认权限屏蔽字。具体方法请阅P84

~改变现有文件的访问权限可以用：chmod（）、fchmod（）和fchmodat（）。chmod在指定文件上进行操作，fchmod函数则对已经打开文件进行操作。fchmodat提供pathname和fd。为了改变文件权限，进程的有效用户ID必须等于文件所有者ID，或者进程拥有超级用户权限。P86呈现了修改文件的模式的实例。

~黏着位，拥有粘着位的文件或目录要想被删除，必须满足以下条件：1.拥有此文件、2.拥有此目录、3.是超级用户，否则，即使拥有写权限也不可删除。

~更改文件的用户用户ID和组ID主要是用以下几个函数：chown（）、fchown（）、fchownat（）、lchown（），非符号向量下，四个函数类似，符号链接时，lchown和fchownat更改符号链接本身的所有者而不是指向文件。fchown改变fd指向打开文件的所有者。具体的实例请阅P88

~文件长度，stat中st\_size记录了普通文件、符号链接目录的文件长度，长度可以是0，

~文件中的空洞：普通文件允许出现空洞，其是由所设置的偏移量超过文章尾端并且写入了数据导致的。如果使用实用工具如cat（1）复制这个文件，那么空洞会被填补，所有实际数据被置为0.

~可以调用函数truncate（）和ftruncate（）对文件进行截断。具体截断方法及参数见P90

~文件系统，指固定的告诉内核如何处理上层文件的一种规则实现，一个磁盘可以分为多个分区，每个分区可以包含一个文件系统，具体磁盘、分区、文件系统之间的关系请阅P91 图4-13。同时，本节也通过inode讲了关于硬链接和符号链接的区别。讲述了i节点的包含信息、特有属性和意义。具体请阅P92。

~上节讲述了i节点的特性和链接，本节讲述创建链接相关的函数：link（）（由existingPATH和newPATH确定要链接的文件和链接文件）、linkat（）（除了两个PATH，可以给出文件描述符来确定文件），linkat函数也可以通过flag参数指定如果文件是个符号链接时要做的操作。具体请阅P93

解除链接使用ulink（）、unlinkat（）、原理是删除现有的目录项，链接计数减一。一般会利用unlink的特性（不会删除已打开的文件的内容），配合open、creat等函数，可以实现创建进程的临时文件，在进程崩溃时不会有残留临时文件（一旦崩溃会立即解除链接）具体实例请见P94-P95

~文件重命名可以使用rename（）和renameat（）。这两个函数可以对各种文件进行重命名操作（除了.和..），当然，根据newname和oldname，有些特例需要提出，具体特例请见P96.

~符号链接，符号链接是一个对文件的间接指针，他可以跨文件系统，使用灵活，无需超级用户，谁都可以对一个文件创建符号链接。P97 图4-17展示了各个常用函数对符号链接的处理。P98给出了几个函数与符号链接联动实现特定功能的实例。

~创建和和读取符号链接：可以使用symlink（）、symlinkat（）创建符号链接，用法与link和linkat相似，但是可以跨系统。

~文件的时间，文件一般会在文件属性中保留三个有关时间的字段：st\_atim（文件数据的最后访问时间），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）。具体时间相关的用法P100。

~文件的时间戳可以使用函数：futimens（）、utimensat（）和utimes（）修改。前两个函数文件由参数文件描述符fd定，其times数组参数包含：访问时间、修改时间两个元素。单位是纳秒。第三个函数文件由路径定，times参数包括两个时间戳。具体数据结构请阅P102.

~之前讲述了创建文件相关的函数，创建目录和删除目录使用函数：mkdir（）、mkdirat（）、rmdir（）。前两个函数创建目录，.和..由系统自动创建，文件访问权限有umask设定的屏蔽字确定。一般需要对目录设置 执行 权限来获取目录中的文件名。

~读目录，目录的写、执行权限是指对目录中的文件而不是目录文件本身，只有内核对目录文件有写权限。fdopendir（）函数可以把打开文件的文件描述符转化成目录处理函数需要的DIR结构。集体的函数规则和用法请阅P105.

~一般来说，每个进程都有自己的工作目录，此目录是搜索所有相对路径的起点。如果要更改此进程的当前工作目录，可以使用函数：chdir（）和fchdir（），两者区别为一个参数是路径名，一个是文件描述符。如果需要将一个相对路径解析成一个从根目录开始的绝对路径，可以使用函数：getcwd（），此函数必须要两个参数：缓冲区地址buf、缓冲区长度size，该缓冲区必须有足够长度容纳绝对路径加一个null字符。

~st\_dev字段和st\_rdev的区别：主要在于前者是与每个文件名关联的文件系统设备号，而后者只有字符特殊文件和块特殊文件才有，包括实际设备的设备号。

~P112-P113 图4-26展示了文件访问权限位的小结。

# 第五章 标准I/O库

本章讲述标准IO库。

~对于一个标准IO操作围绕 流 （stream）进行。当用标准IO库打开或创建一个文件时，我们已经使一个流和一个文件关联，“流”可用于单字节或多字节字符集。流的定向决定了所读写的字符是单字节还是多字节的。流的定向由函数fwide（）设置，由函数freopen（）清除。具体详细请阅P115-

P116

~和之前对应上的，对每个进程预定义了3个流，他们是标准输入、标准输出和标准错误。，这三个定义在头文件 stdio.h 里。

~缓冲：缓冲的存在是为了减少系统调用read和write的次数，将要写或读的流存进缓冲区，之后根据某种规则将缓冲区的数据按批次进行实际IO操作。

标准IO主要提供3种缓冲方式：1.全缓冲：填满缓冲区后才进行实际的IO操作，通常驻留在磁盘的文件一般使用这种IO缓冲。2.行缓冲：遇到换行符时才将之前缓冲区冲洗，一般涉及到终端输入输出时使用此方式。3.不带缓冲：标准IO库不对字符进行缓冲，一般标准错误会这样做，尽快将错误信息打印。

~将缓冲区的字符写入磁盘的操作叫做 冲洗（flush）。除了进程的自动冲洗外，可以使用函数：fflush（）手动冲洗缓冲区内容。

~我们可以使用函数：setbuf（）和setvbuf（）来手动更改缓冲类型。具体注意事项请阅P117-P118

~打开流：可以使用函数打开流：fopen（）、freopen（）和fdopen（），三者的区别是：fopen根据路径名打开文件，freopen根据在指定流上打开指定文件，该流不能打开，不能定向。fdopen取已有文件描述符与流结合，具体的注意事项请阅P118-P119.

~对于打开流函数的type参数，有15种不同的值，其表格和实例请阅P119。

~关闭流：使用函数 fclose（），文件被关闭前会冲洗缓冲区中输出数据，放弃输入数据。当进程正常终止，冲洗所有标准IO流，关闭所有打开的标准IO流。

~读和写流：打开流后，可选3种读写操作：每次一个字符的IO、每次一行的IO、直接IO。具体区别和联系请阅P120.

~读流函数，以下函数一次读一个字符：getc（）、fgetc（）和getchar（）。getchar等同于getc，前两个的区别是getc可以实现为宏而fgetc不行，导致的结论请阅P121.

~要注意的是，以上三个函数对于出错和读到结束符的返回值相同，要获取状态，需要调用函数：ferror和feof来判断。

~从流中读取数据后，可以调用ungetc（）将字符压回流中。

~流输出函数：对应以上的输入函数，对应了三个输出函数：putc（）、fputc（）和putchar（）。

~每次一行IO：以下函数提供每次输入一行的功能：fgets（）、gets（），这两个函数指定了buffer地址，读入的行直接送入其中，gets从标准输入读，fgets从指定流读。具体的读入规则请阅{P122

~P123-P124讲述了使用标准IO的效率比较。

~二进制IO，如果对一个结构体进行读写操作，无论是一次一个字符还是一次一行，都十分不便，所以提供了两个函数进行类结构体的二进制IO：

fread（）、fwrite（）。常用于读写二进制数组或结构体。当然这种方法也有限制：只能读写同一系统上的已写数据。具体原因请阅P126.

~定位流，有时候需要定位一个流正被什么文件使用，UNIX给出了三种方法定位：1.ftell和fseek函数。2.ftello和fseeko函数。3.fgetpos和fsetpos函数。具体区别和联系请阅P126.

~UNIX提供多种格式化IO，格式化输出：printf（）（格式化数据写到标准输出），fprintf（写到指定的流），dprintf（写到指定文件描述符），sprintf（讲格式化数据写到数组buf），sprintf（在数组末尾自动加null字节，但不包括到返回值）。P128-P129说明了标准输出的相关实例和参数说明。

~格式化输入:scanf(输入到标准输入)，fscanf，sscanf，其作用是 分析输入字符串，并将字符序列转换成指定类型的变量。具体的参数说明请阅P130

~每个标准IO流都有一个与其相关联的文件描述符，可以使用fileno（）获得其描述符。

~从P131-P133给出了一个对各个标准IO流打印缓冲状态信息的实例。

~临时文件，ISO C提供了两个函数创建临时文件。tmpnam函数产生一个与现有文件名不同的有效的路径名字符串，每次调用都会产生一个不同路径名，做多调用TMP\_MAX 次。TMP\_MAX定义在 stdio.h中。tmpfile函数会创建一个临时二进制文件，进程结束或文件关闭时自动销毁。P135给出了应用实例。

还有一种mkstemp函数，这个函数会以唯一的名字创建一个普通文件并打开此文件，但是此函数创建的文件不会自动删除。P136给出了此函数的使用实例。

~内存流，标准IO的缓冲区默认在内存中，我们可以使用setbuf和setvbuf来更改成自己的缓冲区。fmemopen函数支持调用缓冲区用于内存流。具体的调用参数和注意事项请阅P137-P139.P138给出了一个内存流写入操作的实例。

还有两个函数能够创建内存流：open\_memstream（）和open\_wmemstream，前者用于创建流是面向字节的，后者是用来创建流是面向宽字节的。其与fmemopen有一定区别，具体请阅P193.

# 第六章 系统数据文件和信息

本章主要讲述了系统运行有关的数据文件和可移植接口。

~口令文件是类UNIX系统共有的，一把在/etc/passwd，其字段请阅P142，都包括在 pwd.h的怕是passwd结构中。

~P143说明了一些常用的口令文件的相关操作和登录项的说明。提出了finger、vipw等命令的用法。

~获取口令文件项的函数：getpwuid（）和getpwnam（），前一个由ls程序使用，后一个由login程序使用。这两个函数都返回指向passwd结构体的指针。还有三个函数可用来查看整个口令文件：getpwent（）、setpwent（）和endpwent（）。成功则返回口令文件的指针。

~上节说道，会把用户口令放入一个单项加密的文件中，一般为/etc/shadow，称为阴影口令。其字段见P145.有一组函数可用于访问阴影口令：getspnam（）、getspent（）、setspent（）和endspent（）。

~和用户口令文件类似的，UNIX有组文件，字段如P146 图6-4所示，提供两个函数查看组名或数值ID：getgrgid（）、getgrnam（）。提供一组函数用来搜索整个组文件：getgrent（）、setgrent（）和endgrent（）。详情请阅P147.

~附属组ID，类UNIX系统提供了附属组概念，一个用户可以加入多个附属组，来获得这些组相应的文件权限，提供以下函数获取和设置附属组ID：getgroups（）、setgruops（）、initgroups（），其具体的工作呢原理和字段说明请阅P147.

~P148给出了不同的UNIX系统对于存储加密口令字的不同实现方式，请阅P148

~6.7节给出了一些常用的其他数据文件，如：passwd,group,shadow,hosts,networks,serveices等，每个数据文件都至少对应三个函数：get、put和end，具体功能说明请阅P149.

~大多数系统都会提供utmp和wtmp文件，utmp用来记录当前登录到系统的各个用户，wtmp用来跟踪各个登录和注销事件。具体的系统调用请阅P149

~系统标识主要由各种与当前主机有关的操作系统信息组成，常用命令有：uname：获取主机和操作系统有关信息，gethostname：获取主机名，hostname：设置和获取主机名。P151给出了返回值的最大长度和相关设置。

~对于系统的日历时间，可以用time（）函数来获得。clock\_gettime（）函数可以获得指定时钟的时间。其他与时间有关的函数还有：localtime（日历时间转换成本地时间）gmtime（日历时间转换成协调统一时间），mktime（以本地时间为参数，将其变成time\_t的值），strftime（复杂的时间值函数，内含多个参数），strftime\_l（允许将区域指定为参数，其他和strftime相同），P151-P154给出了以上函数及参数的详细解释。P155给出了strptime函数实例。P156给出了strptime函数的转换说明。

# 第七章 进程环境

~ 1．进程如何启动和退出的（main函数，初启函数，exit函数，atexit函数）。

（1）Linux创建进程的唯一方法就是利用shell的fork系统调用。

（2）编译C程序时，连接器会将初启函数和main函数一起装入内存形成可执行文件，指定该初起函数为该初启函数的开始执行点。

（3）初启函数负责从UNIX SHELL获得命令行参数和系统的环境变量，将它们组织成main函数需要的形式，并调用main函数。

（4）当C程序从main返回时，初启函数负责调用exit函数中止进程。

（5）初启函数也是由shell创建的。也是shell的子进程。

~2．命令行参数

（1）命令行参数分为两种，一种是选项参数，如-l –u 。一种是非选项参数，大多是文件名或路径名。

（2．对于系统，程序得到命令行参数有两种办法。

通过初起函数得到。

由父进程传递给子进程。

(3)对于程序，程序得到命令行参数的唯一方法是通过argc,argv.

(4)当参数个数固定或所以的参数都可按相同的方式理解，可直接用argc,argv获得。

(5)当参数有选项参数和非选项参数混合时，可使用getopt函数对选项参数进行逐一扫描，实现区分。但必须对选项参数进行语法约定。

(6)选项参数的约定有：

选项参数必须以”-“开头。

当选项不需要值的时候可以连着写。

选项参数必须写在它的非选项参数之前。

二、命令行参数

1.

|  |  |
| --- | --- |
| 名称:： | getopt |
| 功能： | 扫描选项参数 |
| 头文件： | #include <unistd.h> |
| 函数原形： | int getopt(int argc,char \*const argv[],const char \*optstring); |
| 参数： | argc  argv  opestring 选项参数 |
| 返回值： |  |

~getopt( )用来分析命令行参数。参数argc和argv是由main( )传递的参数个数和内容。参数optstring 则代表欲处理的选项字符串。

返回值:如果找到符合的参数,则返回此参数字母.如果参数不包含在参数optstring 的选项字母,则返回字符，分析结束则返回-1

~getopt(argc,argv,”islf”);

./a.out –i –x –l –s file1

~如果在一个选项后面有“：”，那么该选项后面必须带有值。

getopt(argc,argv,”is:lf”);

~和getopt有关的4个全局变量

1．opterr

当opterr等于0的时候，遇到非法的选项字符时，或者选项要求有值而缺省时，它会打印“？”。缺省时opterr是等于0的。

当opterr不等于0的时候，遇到非法的选项字符时，或者选项要求有值而缺省时，它会打印一条出错信息到标准输出。

还有一种情况时，第三个参数的开头是一个“：”，可以通过opterr区分它是因为遇到非法的选项字符（返回”？”），还是选项要求有值而缺省出错(返回“:”)。

2.optopt 但出现两种异常的时候，optopt会把它出错信息存储起来。

3.optind argv存放下一个要处理元素的索引，初值为1。

4.optarg 若选项要求有值需接收值，optarg指向选项的值。

~三、终止进程

1．

|  |  |
| --- | --- |
| 名称:： | exit/\_exit |
| 功能： | 终止一个进程 |
| 头文件： | #include <stdlib.h> (exit)  #include <unistd.h>（\_exit） |
| 函数原形： | void exit(int status)；  void \_exit(int status); |
| 参数： | status  终止状态 |
| 返回值： | 无 |

exit和\_exit函数用于正常终止一个程序：\_exit立即进入内核，exit则先执行一些清除处理，然后进入内核。

exit函数总是执行一个标准I/O库的清除关闭操作，对于所有打开流调用fclose函数，把所缓存中的所有数据都写到文件中。

exit和\_exit函数都带一个参数，称为终止状态，大多数UNIX Shell都提供检查一个进程终止状态的方法。

2．

|  |  |
| --- | --- |
| 名称:： | atexit |
| 功能： | register a function to be called at normal program termination. |
| 头文件： | #include <stdlib.h> |
| 函数原形： | int atexit(void（\*function）(void)); |
| 参数： | function   函数的地址 |
| 返回值： | 若成功则为0，否则不为0。 |

~一个函数可以最多登记32个函数，这些函数将由exit自动调用，我们称这些函数为终止处理函数，用stexit登记它们。

其中，atexit的参数是一个函数地址，当调用此函数时无需向它传递任何参数，也不期望它返回一个值。exit以登记这些函数的相反顺序调用它们。同一函数如若登记多次，则也被调用多次。

~exit首先调用各终止处理函数，然后按需多次调用fclose关闭所有打开流。

~四、进程空间

|  |  |
| --- | --- |
| 名称:： | malloc/calloc/realloc/free |
| 功能： | allocate and free dynamic memory |
| 头文件： | #include <stdlib.h> |
| 函数原形： | void \*calloc(size\_t nmemb,size\_t size);  void \*malloc(size\_t size)  void \*realloc(void \*ptr,size\_t size);  void free(void \*ptr); |
| 参数： | size   存储区大小  nmemb 指定的对象  ptr     指向存储区的指针 |
| 返回值： | 若成功则返回非空指针，否则返回NULL |

malloc分配制定字节的存储区，此存储区的初始值不能确定。

calloc为指定的对象分配能容纳其指定个数的存储空间。

realloc更改以前分配的存储区大小。

free释放分配的存储区。

realloc使我们可以增减以前分配区的长度。如果在该存储区后有主够的空间可供扩充，则可在原存储区位置上向高地址方向扩充，如果在原存储区没有足够的空间，则realloc分配另一个足够大的存储区。通常分配存储空间比实际的要大一些，额外的空间用来记录管理信息——分配块的长度。指向下一个分配块的指针等等。

|  |  |
| --- | --- |
| 名称:： | alloca |
| 功能： | memory allocator |
| 头文件： | #include <alloca.h> |
| 函数原形： | void \*alloca(size\_t size); |
| 参数： | size   存储区大小 |
| 返回值： | 若成功则返回非空指针，否则返回NULL |

~alloca也是用来分配存储空间的，它和malloc的区别是它是在当前函数的栈上分配存储空间，而不是在堆中。其优点是：当函数返回时，自动释放它所使用的栈。

~环境变量顾名思义就是存储进程运行环境的变量.每个程序都接收到一张环境表。环境表是一个字符数组，其中每个指针包含一个以NULL结束的字符串的地址。全局变量environ则包含了该指针数组的地址。

五、进程的环境变量

5．

|  |  |
| --- | --- |
| 名称:： | getenv |
| 功能： | 获取环境变量 |
| 头文件： | #include <stdlib.h> |
| 函数原形： | char \*getenv（const char \*name）; |
| 参数： | name  变量名 |
| 返回值： | 若成功则返回与name相关的value指针，否则返回NULL |

~此函数返回一个指针，它指向name=value字符串的value。我们应当使用getenv从环境中取一个特定的环境变量值，而不是直接访问environ.

注意程序中的USER环境变量是系统用于用户自定义的环境变量。

~6．

|  |  |
| --- | --- |
| 名称:： | putenv |
| 功能： | 增加环境变量 |
| 头文件： | #include <stdlib.h> |
| 函数原形： | int putenv（const char \*str）; |
| 参数： | str  变量名 |
| 返回值： | 若成功则返回0，否则返回非0 |

 ~qutenv取形式为name=value的字符串，将其放到环境表中。如果name已存在则先删除其原来的定义。

7．

|  |  |
| --- | --- |
| 名称:： | setenv |
| 功能： | 修改环境变量 |
| 头文件： | #include <stdlib.h> |
| 函数原形： | int setenv(const char \*name,const char \*value,int rewrite); |
| 参数： | str  变量名 |
| 返回值： | 若成功则返回0，否则返回非0 |

~setenv将name设置为value.如果在环境中name已经存在，那么

(1)    若rewrite非0，则首先删除其现存的定义；

(2)    若rewrite为0，则不删除其现存定义（name不设置为新的value，而且也不出错）。

~8.

|  |  |
| --- | --- |
| 名称:： | unstenv |
| 功能： | 删除环境变量 |
| 头文件： | #include <stdlib.h> |
| 函数原形： | void unstenv(const char \*name); |
| 参数： | name  变量名 |
| 返回值： | 无 |

unsetenv删除name的定义。即使不存在这种定义也不算出错。

~9．

|  |  |
| --- | --- |
| 名称:： | clearenv |
| 功能： | 清除环境变量 |
| 头文件： | #include <stdlib.h> |
| 函数原形： | int clearenv(void); |
| 参数： | 无 |
| 返回值： | 若成功则返回0，否则返回非0 |

~环境表和环境字符串通常存放在进程空间的顶部，删除一个字符串很简单，但是增加一个字符串或修改一个现在的字符串就很困难。我们不能增加该空间的长度，如果我们想修改现有的name,并且新value的长度大于原长度，则必须调用malloc为新字符串分配空间，然后将新字符串复制到该空间中，接着使环境表中针对name的指针向新分配区。如果我们想增加一个新的name，并且实第一次增加，则必须调用malloc为新的指针表分配空间。接着将原来的分配表复制到新分配区，并将指向新name=value字符串的指针存放在指针表的表尾。如果不是第一次增加一个新name，可以调用realloc分配比原先多方一个指针的空间。然后将指向新name=value字符串的指针存放在该表表尾，后面跟着一个空指针。

六、跳转函数

~10．

|  |  |
| --- | --- |
| 名称:： | setjmp |
| 功能： | save stack context for non-local goto |
| 头文件： | #include <stjmp.h> |
| 函数原形： | int setjmp(jmp\_buf env); |
| 参数： | env  要跳转的位置 |
| 返回值： | 若直接调用则返回0，若从longjmp调用返回则返回非0。 |

~11．

|  |  |
| --- | --- |
| 名称:： | longjmp |
| 功能： | non-local jump to a saved stack context |
| 头文件： | #include <setjmp.h> |
| 函数原形： | void longjmp(jmp\_buf env,int val); |
| 参数： | env  从函数setjmp传递过来的要跳转的位置  val  返回给setjmp用于判断函数跳转 |
| 返回值： | 无 |

~setjmp和longjmp都是对栈操作函数。setjmp和longjmp函数用于函数调转，setjmp用于标记跳转后开始执行程序的位置。longjmp用于实现跳转。sejmp的参数env的类型是一个特殊类型jmp\_buf。这一数据类型是某种形式的数组，其中存放在调用longjmp时能用来恢复栈状态的所有信息。因为需要在另外一个函数中引用env变量，所有规范的处理方式是将env变量定义为全局变量。因为一个程序可以实现多次跳转，setjmp不知道是哪个longjmp跳转的，所有需要一个参数来标记，这个参数就是val，它将成为从setjmp处返回的值。这样我们可以通过测试返回值的方法来判断是那一个longjmp跳转的。

七、进程的资源限制

~12.

|  |  |
| --- | --- |
| 名称:： | getrlimit/setrlimit |
| 功能： | 获得和修改进程资源限制 |
| 头文件： | #include <sys/resoutce.h> |
| 函数原形： | int getrlimit(int resource,struct rlimit \*rlptr);  int setrlimit(int resource,const struct rlimit \*rlptr); |
| 参数： |  |
| 返回值： | 若成功返回0，若失败返回非0值。 |

~进程的资源现在通常是在系统初始化时由进程0建立，然后由后续进程继承。

rlimit 结构定义如下：

struct rlimit{

       rlim\_t rlim\_cur;

       rlim\_t rlim\_max;

};

更改资源限制时必须遵循下列限制：

一 .任何一个进程都可以将一个软限制值改为小于或等于其硬限制值。

二 .任何进程可以降低其硬限制值，但必须大于或等于其软限制值。这种操作对普通用户是不可逆的。

三 .只有超级用户进程可以提高硬限制值。

# 第八章进程控制

一、进程ID

~1．

|  |  |
| --- | --- |
| 名称:： | getpid |
| 功能： | 获得进程id. |
| 头文件： | #include <unistd.h> |
| 函数原形： | pid\_t getpid(void); |
| 参数： | 无 |
| 返回值： | 进程id. |

每个进程都有一个非负整数表示的唯一进程id。系统中有一些专用的进程该进程是内核的一部分，它并不执行任何磁盘上的程序。因此也被称为系统进程。

进程1通常是init进程，在自举过程完成时有内核调用。此进程负责在自举内核后启动一个UNIX系统。init通常读与系统有关的初始化文件（/ect/re\*文件或/ect/inittab文件，以及/ect/init.d中的文件）,并将系统引导到一个状态。但要注意init并不是一个系统进程，它是一个普通用户进程，但是它以超级用户特权运行。

~2.

|  |  |
| --- | --- |
| 名称:： | getppid |
| 功能： | 获得进程的父进程id |
| 头文件： | #include <unistd.h> |
| 函数原形： | pid\_t getppid(void); |
| 参数： | 无 |
| 返回值： | 父进程id. |

~3.

|  |  |
| --- | --- |
| 名称:： | getuid/geteuid |
| 功能： | 获得调用进程用户的实际用户id和有效用户id. |
| 头文件： | #include <unistd.h> |
| 函数原形： | uid\_t getuid(void);  uid\_t geteuid(void) |
| 参数： | 无 |
| 返回值： | 用户id. |

~4．

|  |  |
| --- | --- |
| 名称:： | getgid/getegid |
| 功能： | 获得调用进程用户的实际组id和有效组id. |
| 头文件： | #include <unistd.h> |
| 函数原形： | gid\_t getgid(void);  gid\_t getegid(void); |
| 参数： | 无 |
| 返回值： | 用户组id. |

与一个进程相关联的用户id用6个或更多,他们是：实际用户id，实际组id，有效用户id，有效组id，保存的设置用户id，保存的设置组id

实际用户id和实际组id标识我们究竟是谁。这两个字段在登陆时取自口令文件中的登陆项。通常，在一个登陆会话间这些值并不改变，但是超级用户进程又方法改变它们。

有效用户id和有效组id以及附加组id决定了我们的文件访问权限。

保存的设置用户id和保存的设置组id在执行一个程序是包含了有效用户id和有效组id的副本。

通常有效用户id等于实际用户id，有效组id等于实际组id。它们不同的情况通过发生在变更用户权限，但是不切换用户的情况下。

linux中有许多id，我们要注意这些id 的区别。getpid,getuid是获得当前进程和进程组的id，而getuid,geteuid,getgid,getegid是获取当前用户和用户组的id 。

二、创建进程

~5．

|  |  |
| --- | --- |
| 名称:： | fork |
| 功能： | 创建一个进程 |
| 头文件： | #include <unistd.h>  #include <sys/types.h> |
| 函数原形： | pid\_t fork(void); |
| 参数： | 无 |
| 返回值： | 子进程返回0，父进程返回子进程id，出错返回1。 |

~由fork创建的进程称为子进程，fork函数被调用一次，但返回两次。两次返回唯一区别是子进程返回的是0，父进程返回的是子进程的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。父、子进程并不共享这些存储空间，父、子进程只共享正文段。由于在fork之后经常跟着exec，所以现在很多实现并不执行一个父进程数据段、堆和栈的完全复制，它们采用一种叫做写时复制技术。

 ~fork有两种用法:

(1)一个父进程希望复制自己，使父、子进程同时执行不同的代码段。

(2)一个进程要执行一个不同的程序。这对shell是常见的情况。

~6．

|  |  |
| --- | --- |
| 名称:： | vfork |
| 功能： | 创建一个进程 |
| 头文件： | #include <unistd.h>  #include <sys/types.h> |
| 函数原形： | pid\_t vfork(void); |
| 参数： | 无 |
| 返回值： | 进程id. |

vfork与fork一样都创建一个进程，但是它并不是将父进程的地址空间完全复制到子进程中，在子进程调用exec或exit之前，它在父进程的空间中运行。这要就提高了效率。

vfork和fork的另一个区别是：vfork保证子进程先运行，在它调度exex或exit后父进程才可能被调度运行。

三、等待进程终止

~7．

|  |  |
| --- | --- |
| 名称:： | wait/waitpid |
| 功能： | 等待一个进程的终止 |
| 头文件： | #include <sys/types.h>  #include <sys/wait.h> |
| 函数原形： | pid\_t wait(int \*statloc);  pid\_t waitpid(pid\_t pid,int \*statloc,int options); |
| 参数： | statloc 指向存放终止状态单元的指针  pid    options  控制wait的操作 |
| 返回值： | 若成功返回进程id，若出错返回0。 |

~两个函数都是的功能都是获取子进程终止状态，但在一个子进程终止前，wait使其调用者阻塞，而waitpid有一个选项，可使调用者不阻塞。

如果一个之进程已经终止，并且是一个僵尸进程，则wait立即返回并取得该子进程的状态，否则wait使其调用者阻塞直到一个子进程终止。

options参数使我们能进一步控制waitpid的操作。此参数可以是0，或者是:

waitpid并不等待在其调用之后的第一个终止进程，它有若干个选项，可以控制它所等待的进程。

~对于waitpid函数的参数pid的解释如下：

pid==-1   等待任一子进程。

pid>0    等待其进程与pid相等的子进程。

pid<-1    等待其组id等于pid绝对值的任一子进程。

options参数使我们能进一步控制waitpid的操作。此参数可以是0，或者是:

WCONTINUED  用于作用控制

WNOHANG    若pid指定的子进程并不是立即可用的，则waitpid不阻塞，此时其返回值为0。

WUNTRACED   用于作业控制。

~对于僵尸进程的解释：

在一个进程调用了exit之后，该进程并非马上就消失掉，而是留下一个称为僵尸进程（Zombie）的数据结构。

在Linux进程的5种状态中，僵尸进程是非常特殊的一种，它已经放弃了几乎所有内存空间，没有任何可执行代码，也不能被调度，仅仅在进程列表中保留一个位置，记载该进程的退出状态等信息供其他进程收集，除此之外，僵尸进程不再占有任何内存空间。

~程序先打印子、父进程的id.然后子进程调用sleep函数，这时父进程阻塞，直到子进程调用exit终止退出，父进程获得子进程终止信息。

~8．

|  |  |
| --- | --- |
| 名称:： | wait3/wait4 |
| 功能： | 等待一个进程的终止 |
| 头文件： | #include <sys/types.h>  #include <sys/wait.h>  #include <sys/time.h>  #include <sys/resource.h> |
| 函数原形： | pid\_t wait3(int \*stacloc,int options,struct rusage \*rusage);  pid\_t wait4(pid\_t pid,int \*statloc,int options,struct rusage \*rusage); |
| 参数： |  |
| 返回值： | 若成功返回进程id ，若出错返回-1。 |

~wait3,wait4的功能更强大，这与附加参数rusage有关。该函数要求内核返回由终止进程及其所有子进程使用的资源汇总。资源统计信息包括用户CPU时间总量，系统CPU时间总量，页面出错次数，接收到信号的次数等。

四、执行新程序

~9．

|  |  |
| --- | --- |
| 名称:： | exec |
| 功能： | 执行一个新程序 |
| 头文件： | #include <unistd.h> |
| 函数原形： | int execl(const char \*path,const char \*arg,…);  int execlp（const char \*file,const char \*arg,…）;  int execle(const char \*path,const char \*arg,…,char \*const envp[]);  int execv(const char \*fath,char \*const argv[]);  int execvp(const char \*file,char \*const argv[]);  int execve(const char \*file,char \*const argv[],char \*const envp[]); |
| 参数： | path 路径名  file  文件名  arg  单独的命令参数，以NULL结束。  argv 指向命令参数列表  envp 指向环境字符串指针数组的指针 |
| 返回值： | 若出错返回-1。若成功不返回值 |

~用fork函数创建子进程后，子进程往往要调用一种exec函数以执行另一个程序。当进程调用一种exec函数时，该进程完全由新程序代换，而新程序则从其main函数开始执行。因为调用exec并不创建新进程，所以前后的进程ID并未改变。exec只是用另一个新程序替换了当前进程的正文，数据，堆和栈段。

~有六种不同的exec函数可供使用，它们常常被统称为exec函数。在这些函数中字母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表示该函数去一个参数表，它与字符v互斥。v表示该函数去一个argv[]矢量。最后，字母e表示该函数取envp[]数组，而不使用当前环境。

~在执行exec前后实际用户ID和实际组ID保持不变，而有效ID是否改变则取决于所执行程序文件的设置用户ID位和设置组ID位是否设置。如果新程序的设置用户ID位已设置，则有效用户ID变成程序文件所以者的ID，否则有效用户ID不变。对组ID的处理方式与此相同。

~程序先利用argv[1]调用./main函数，然后把argv传给main函数，把参数打印到屏幕上。要注意main函数必须和本程序在同一目录下。

五、更改ID

~10．

|  |  |
| --- | --- |
| 名称:： | setuid/setgid |
| 功能： | 更改用户ＩＤ和组ID |
| 头文件： | #include <unistd.h>  #include <sys/types.h> |
| 函数原形： | int setuid(uid\_t uid);  int setgid(uid\_t gid); |
| 参数： | uid 用户ID  gid 用户组ID |
| 返回值： | 若成功返回进程0，若出错返回-1。 |

（1）若进程具有超级用户特权，则setuid函数将实际用户ID，有效用户ID，以及保存的设置-用户-ID设置为uid.

（2） 若进程没有超级用户特权，但是uid等于实际用户ID或保存的设置-用户-ID，则setuid只有效用户ID设置为uid。不改变实际用户ID和保存的设置-用户-ID。

（3） 如果上面两个条件都不满足，则errno设置-用户-ID。

~11．

|  |  |
| --- | --- |
| 名称:： | setreuid/setregid |
| 功能： | 交换实际用户ID和有效用户ＩＤ． |
| 头文件： | #include <unistd.h> |
| 函数原形： | int setreuid(uid\_t ruid,uid\_t euid);  int setregid(gid\_t rgid,uid\_t egid); |
| 参数： | euid 有效用户id  ruid　实际用户id  egid 有效组id  rgid实际组ｉｄ |
| 返回值： | 若成功返回进程0，若出错返回-1。 |

可以用setuid函数设置实际用户ID和有效用户ID。与此类似，可以用setgid函数设置实际组ID和有效ID。

~12.

|  |  |
| --- | --- |
| 名称:： | seteuid/setegid |
| 功能： | 更改用户有效ID和有效组ID |
| 头文件： | #include <unistd.h> |
| 函数原形： | int setuid(uid\_t uid);  int setgid(gid\_t gid); |
| 参数： | uid 用户id  gid 组id |
| 返回值： | 若成功返回进程0，若出错返回-1。 |

# 第九章 进程关系

# 第十章 信号

~概念

信号是软件中断，很多重要的应用程序需要处理信号

信号提供了一种处理异步事件的方法，例如终端用户键入中断键，则会通过信号机制停止一个程序

 每个信号都有一个名字，都以 SIG 开头，例如 SIGABRT 是夭折信号头 <signal.h> 中，这些信号被定义为正整数（信号编号）

unix中不存在信号编号为0的信号，kill函数对信号0有特殊的应概念

信号是软件中断，很多重要的应用程序需要处理信号

信号提供了一种处理异步事件的方法，例如终端用户键入中断键，则会通过信号机制停止一个程序

每个信号都有一个名字，都以 SIG 开头，例如 SIGABRT 是夭折信号头 <signal.h> 中，这些信号被定义为正整数（信号编号）

unix中不存在信号编号为0的信号，kill函数对信号0有特殊的应用，POSIX.1称此种信号为空信号

信号是异步事件，产生信号的时间对进程而言是随机出现的，进程通知内核该信号出现的时候执行什么操作，当信号出现，内核执行相关操作用，POSIX.1称此种信号为空信号

信号是异步事件，产生信号的时间对进程而言是随机出现的，进程通知内核该信号出现的时候执行什么操作，当信号出现，内核执行相关操作

~进程的处理

可以要求内核按照下列三种方式之一进行处理：

1忽略此信号

大多数信号可以使用这种方式处理，但是有两种不能被忽略：SIGKILL 和 SIGSTOP，这两种信号不能被忽略，因为他们向超级用户提供了使进程终止或停止的可靠方法，另外，非法内存引用或被0除这些由硬件异常产生的信号如果被忽略，则进程的行为是未定义的

2捕捉信号

进程通知内核在某种信号时调用一个函数，执行对事件的处理，但是注意，不能捕捉 SIGKILL 和 SIGSTOP 信号

3执行系统默认动作

针对大多数信号的系统默认动作是终止进程

信号列表

执行 kill -l 可以获得系统支持的全部信号

信号说明

1.SIGHUP

本信号在用户终端连接(正常或非正常)结束时发出, 通常是在终端的控制进程结束时, 通知同一session内的各个作业, 这时它们与控制终端不再关联。

登录Linux时，系统会分配给登录用户一个终端(Session)。在这个终端运行的所有程序，包括前台进程组和后台进程组，一般都属于这个Session。当用户退出Linux登录时，前台进程组和后台有对终端输出的进程将会收到SIGHUP信号。这个信号的默认操作为终止进程，因此前台进程组和后台有终端输出的进程就会中止。不过可以捕获这个信号，比如wget能捕获SIGHUP信号，并忽略它，这样就算退出了Linux登录，wget也能继续下载。

此外，对于与终端脱离关系的守护进程，这个信号用于通知它重新读取配置文件。

2.SIGINT

程序终止(interrupt)信号, 在用户键入INTR字符(通常是Ctrl-C)时发出，用于通知前台进程组终止进程。

3 SIGQUIT

和SIGINT类似, 但由QUIT字符(通常是Ctrl-/)来控制. 进程在因收到SIGQUIT退出时会产生core文件, 在这个意义上类似于一个程序错误信号。

4 SIGILL

执行了非法指令. 通常是因为可执行文件本身出现错误, 或者试图执行数据段. 堆栈溢出时也有可能产生这个信号。

5 SIGTRAP

由断点指令或其它trap指令产生. 由debugger使用。

6 SIGABRT

调用abort函数生成的信号。

7 SIGBUS

非法地址, 包括内存地址对齐(alignment)出错。比如访问一个四个字长的整数, 但其地址不是4的倍数。它与SIGSEGV的区别在于后者是由于对合法存储地址的非法访问触发的(如访问不属于自己存储空间或只读存储空间)。

8 SIGFPE

在发生致命的算术运算错误时发出. 不仅包括浮点运算错误, 还包括溢出及除数为0等其它所有的算术的错误。

9 SIGKILL

用来立即结束程序的运行. 本信号不能被阻塞、处理和忽略。如果管理员发现某个进程终止不了，可尝试发送这个信号。

10 SIGUSR1

留给用户使用

11 SIGSEGV

试图访问未分配给自己的内存, 或试图往没有写权限的内存地址写数据.

12 SIGUSR2

留给用户使用

13 SIGPIPE

管道破裂。这个信号通常在进程间通信产生，比如采用FIFO(管道)通信的两个进程，读管道没打开或者意外终止就往管道写，写进程会收到SIGPIPE信号。此外用Socket通信的两个进程，写进程在写Socket的时候，读进程已经终止。

14 SIGALRM

时钟定时信号, 计算的是实际的时间或时钟时间. alarm函数使用该信号.

15 SIGTERM

程序结束(terminate)信号, 与SIGKILL不同的是该信号可以被阻塞和处理。通常用来要求程序自己正常退出，shell命令kill缺省产生这个信号。如果进程终止不了，我们才会尝试SIGKILL。

16 SIGCHLD

子进程结束时, 父进程会收到这个信号。

如果父进程没有处理这个信号，也没有等待(wait)子进程，子进程虽然终止，但是还会在内核进程表中占有表项，这时的子进程称为僵尸进程。这种情况我们应该避免(父进程或者忽略SIGCHILD信号，或者捕捉它，或者wait它派生的子进程，或者父进程先终止，这时子进程的终止自动由init进程来接管)。

17 SIGCONT

让一个停止(stopped)的进程继续执行. 本信号不能被阻塞. 可以用一个handler来让程序在由stopped状态变为继续执行时完成特定的工作. 例如, 重新显示提示符

18 SIGSTOP

停止(stopped)进程的执行. 注意它和terminate以及interrupt的区别:该进程还未结束, 只是暂停执行. 本信号不能被阻塞, 处理或忽略.

19 SIGTSTP

停止进程的运行, 但该信号可以被处理和忽略. 用户键入SUSP字符时(通常是Ctrl-Z)发出这个信号

20 SIGTTIN

当后台作业要从用户终端读数据时, 该作业中的所有进程会收到SIGTTIN信号. 缺省时这些进程会停止执行.

21 SIGTTOU

类似于SIGTTIN, 但在写终端(或修改终端模式)时收到.

22 SIGURG

有"紧急"数据或out-of-band数据到达socket时产生.

23 SIGXCPU

超过CPU时间资源限制. 这个限制可以由getrlimit/setrlimit来读取/改变。

24 SIGXFSZ

当进程企图扩大文件以至于超过文件大小资源限制。

25 SIGVTALRM

虚拟时钟信号. 类似于SIGALRM, 但是计算的是该进程占用的CPU时间.

26 SIGPROF

类似于SIGALRM/SIGVTALRM, 但包括该进程用的CPU时间以及系统调用的时间.

27 SIGWINCH

窗口大小改变时发出.

28 SIGIO

文件描述符准备就绪, 可以开始进行输入/输出操作.

29 SIGPWR

Power failure

30 SIGSYS

非法的系统调用。

在以上列出的信号中，程序不可捕获、阻塞或忽略的信号有：SIGKILL，SIGSTOP

不能恢复至默认动作的信号有：SIGILL，SIGTRAP

默认会导致进程流产的信号有：SIGABRT，SIGBUS，SIGFPE，SIGILL，SIGIOT，SIGQUIT，SIGSEGV，SIGTRAP，SIGXCPU，SIGXFSZ

默认会导致进程退出的信号有：SIGALRM，SIGHUP，SIGINT，SIGKILL，SIGPIPE，SIGPOLL，SIGPROF，SIGSYS，SIGTERM，SIGUSR1，SIGUSR2，SIGVTALRM

默认会导致进程停止的信号有：SIGSTOP，SIGTSTP，SIGTTIN，SIGTTOU

默认进程忽略的信号有：SIGCHLD，SIGPWR，SIGURG，SIGWINCH

此外，SIGIO在SVR4是退出，在4.3BSD中是忽略；SIGCONT在进程挂起时是继续，否则是忽略，不能被阻塞。

UNIX 系统信号机制最简单的接口是 signal 函数，定义于 signal.h 头文件中

1

void (\*signal(int signo, void (\*func)(int)))(int);

调用成功返回信号以前的处理配置，出错返回 SIG\_ERR

signal 函数是由 ISO C 定义的，不涉及多进程、进程组以及终端IO等，所以他对信号的定义非常含糊，所以对UNIX而言几乎毫无用处

因为 signal 的语义与具体实现有关，所以不建议使用

signo 参数是信号名，func 的值是常量 SIG\_IGN（忽略信号）、SIG\_DFL（执行系统默认信号）或当接到此信号后要调用的函数的地址

用指定函数处理信号称为“捕捉”该信号，此函数称为“信号处理函数”或“信号捕捉函数”

signal 函数最大的缺点是如果想要获取一个信号的当前处理方式，则必须首先改变他的处理方式

系统调用的重新启动

信号发生的时刻，内核会中断正在执行的指令序列，跳转到信号处理函数中执行

信号处理函数中如果没有使用exit或longjmp，当函数执行后会跳回捕捉到信号时正在执行的正常指令序列中，类似于硬件中断的处理方式

但是被中断的系统调用并不一定是可以完全恢复的，比如对低速设备进行操作的时候被中断，因此在unix中引入了自动重启机制

会被自动重启的系统调用有：ioctl、read、readv、write、writev、wait、waitpid

因为应用程序并不是总是希望去重新启动这些系统调用，所以在不同的系统中的处理是不同的

可重入函数

在进程中，如果正在执行malloc操作进行堆空间的存储分配，此时插入执行信号处理函数会发生什么呢？又例如某些将结果存放到静态存储单元中的函数，期间插入执行信号处理程序，在程序中再次在静态存储单元中存放数据，这时会发生什么呢？

上述的两种情况是不可重入的，因为内核维护的堆存储区链表可能正在更改，而第二种情况中存储区可能被覆盖

其他大多数函数是不可重入的，主要原因是：

1.他们使用静态数据结构

2.他们调用 malloc 或 free

3.他们是标准IO函数，而标准IO的很多事先都以不可重入方式使用全局数据结构

由于线程只有一个 errno 变量，所以信号处理程序可能会修改他的值，因此，作为一个通用规则，信号处理函数应在执行可重入函数之前先保存 errno 的值，并在其后恢复

若在信号处理程序中调用一个不可重入函数，其结果是不可预见的

信号的产生、阻塞和递送

当引发信号的时间发生时，内核向进程发送一个信号，事件可以是硬件异常（如除以0）、软件条件（如alarm计时器超时）、终端产生的信号或调用kill函数产生的信号

信号产生时，内核通常在进程表中设置一个某种形式的标志，这一动作，我们称之为“向进程递送了一个信号”，信号产生和递送之间，称信号是“未决的”

进程可以选用信号递送阻塞，如果为进程产生了一个选择为阻塞的信号，而且对该信号的动作是系统默认动作或捕捉该信号，则为该进程将此信号保持未决状态，知道进程对此信号解除阻塞，或将此信号的动作改为忽略

需要注意的是，内核是在递送信号的时候决定信号的动作的，而不是在信号产生时，因此进程可以在信号递送给他之前改变信号的动作

通过调用 sigpending 函数可以判定哪些信号被设置为阻塞并处于未决状态

在进程解除对某个信号的阻塞之前，大多数UNIX实现不允许系统递送该信号多次，相同信号只会递送一次

信号屏蔽字

每个进程都有一个信号屏蔽字，规定了当前要阻塞递送到进程的信号集，每种可能的信号，该屏蔽字中都有一位与之对应，进程可以调用 sigprocmask 函数来检测和更改其当前信号屏蔽字

信号的发送 -- kill 和 raise 函数

1

2

int kill(pid\_t pid,int signo);

int raise(int signo);

定义于 signal.h 中

成功调用返回0，出错返回-1

kill 函数将信号发送给进程或进程组，raise 函数则向自身

调用 raise(signo) 等价于调用 kill(getpid(), signo)

kill 函数的 pid 参数有以下四种情况

1.pid > 0

将该信号发送给进程ID为 pid 的进程

2.pid == 0

将该信号发送给与发送进程属于同一进程组的所有进程（发送进程需要具有向这些进程发送信号的权限）

3.pid < 0

将该信号发送给进程组 ID 为 pid 的绝对值的所有进程

4.pid == -1

将该信号发送给系统上的所有进程（发送进程需要具有向这些进程发送信号的权限）

一般来说，内核进程以及init(pid 1)被称为系统进程集，是不包括在这里所说的“所有进程”中的

除超级用户外，发送进程只能将信号发送给实际用户ID或有效用户ID相同的接受者（SIGCOUNT 信号可以发送给属于同一会话的任何其他进程）

POSIX 将编号为 0 的信号定义为空信号，如果 signo 为0，则只进行正常的错误检查，而不发送信号，这种用法常常被用来检查指定的进程是否存在

如果向一个不存在的进程发送信号，则 kill 函数返回1，errno 被置为 ESRCH，但是必须了解的是，当一个进程结束以后，unix很可能将该PID用于新的进程，并且由于这种测试并不是原子操作，很可能在kill返回结果时原进程已经终止，所以这种测试并没有多大的意义

计时器的使用 -- alarm 和 pause 函数

alarm

使用 alarm 函数可以设置一个计时器，在将来某个指定的时间该计时器会超时，产生 SIGALRM 信号，其默认动作是终止该进程

1

unsigned int alarm（unsigned int seconds);

定义于 unistd.h 中

返回 0 或以前设置的闹钟时间的余留秒数

需要注意的是，经过 seconds 秒之后，内核会产生 SIGALRM 信号，但是由于进程调度的延迟，所以并不能保证进程处理信号的时间

每个进程只能有一个闹钟时钟，如果在调用 alarm 时以前已经为该进程设置过闹钟时钟且尚未超时，则将该闹钟的余留值作为本次 alarm 函数调用的返回值，并将以前登记的闹钟时钟用新值代替

如果我们想捕捉 SIGALRM 信号，则必须在调用 alarm 之前设置该信号的处理程序，因为如果我们先调用 alarm 函数之后，可能在设置处理程序之前已经捕捉到到 SIGALRM 信号，则进程将根据默认动作终止

pause

pause 函数使调用进程挂起直至捕捉到一个信号

1

int pause(void);

定义于 unistd.h 中

返回 -1，并将 errno 设置为 EINTR

信号集

信号集是一个用来表示多个信号的数据类型

作为很多信号处理函数的参数，用来告诉内核不允许发生信号集中的信号

因为信号数目可能超过一个整型量所能表示的最大值，因此，POSIX定义了数据类型 sigset\_t 以包含一个信号集

以下是五个信号集处理函数：

int sigemptyset(sigset\_t \*set);

int sigfillset(sigset\_t \* set);

int sigaddset(sigset\_t \*set,int signum);

int sigdelset(sigset\_t \* set,int signum);

int sigismember(const sigset\_t \*set,int signum);

前四个函数若调用成功则返回0，否则返回-1

第五个函数如果信号在该信号集中，则返回1，否则返回0，调用出错返回-1

所有应用程序在使用信号集前，需要首先通过调用 sigemptyset 或 sigfillset 函数初始化信号集，让信号集变成空集或包含所有信号

然后，可以调用 sigaddset 或 sigdelset 函数在信号集中增加或删除信号，调用 sigismember 测试信号是否存在于信号集中

检测更改信号屏蔽字 -- sigprocmask 函数

调用 sigprocmask 函数可以检测或更改进程的信号屏蔽字

int sigprocmask(int how, const sigset\_t \*restrict set, sigset\_t \*restrict oset);

定义于 signal.h 中，调用成功返回0，否则返回-1

该函数的参数和返回值有如下几种情况

1.若 oset 为非空指针，则当前进程的信号屏蔽字会通过 oset 返回

2若 set 为非空指针，则参数 how 指示如何修改当前信号屏蔽字

3.若 set 为空指针，则不改变该进程的信号屏蔽字，how参数也没有意义

返回未决信号集 -- sigpending 函数

sigpending 返回未决信号集

int sigpending(sigset\_t \*set);

调用成功则返回0，信号集由 set 参数返回，出错返回 -1

设定信号处理函数 -- sigaction 函数

sigaction 函数的功能是检查或修改与指定信号相关联的处理动作（或同时执行这两种操作）

int sigaction(int signo, const struct sigaction \*restrict act, struct sigaction \*restrict oact);

定义于 signal.h 中

调用成功返回0，否则返回-1

参数 signo 是要检测或修改的具体动作的信号编号，若 act 指针为非空，则要修改其动作，若 oact 指针为非空，则系统经由 oact 指针返回该信号的上一动作

sigsetjmp 和 siglongjmp 函数

之前我们看到，调用 setjmp 和 longjmp 函数在信号处理程序中是非常有用的，用来直接返回到程序的主循环中，而不是从信号处理程序正常返回

但是，当进入信号捕捉函数时，此时当前I型您好被自动地加到进程的信号屏蔽字中，如果信号处理程序没有正常返回，则进程将一直屏蔽该信号的产生

因此，POSIX 提供了 sigsetjmp 和 siglongjmp 两个函数

int sigsetjmp(sigjmp\_buf env, int savemask);

void siglongjmp(sigjmp\_buf env, int val);

定义于 setjmp.h 中

第一个函数如果直接调用返回0，若从 siglongjmp 调用返回则返回非0

这两个函数与 setjmp 和 longjmp 函数相比，sigsetjmp 增加了一个参数 -- savemask

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的 savemask 的 sigsetjmp 调用已经保存了 env，则 siglongjmp 从其中恢复保存的信号屏蔽字

sigsuspen 函数

int sigsuspend(const sigset\_t \*sigmask);

返回-1，并将 errno 设置为 EINTR

将进程的信号屏蔽字设置为 sigmask 指向的值，并挂起进程直至有信号产生

system 函数是由 fork 和 exec 函数实现的

system 和信号处理

考虑通过 system 函数调用 ed 编辑器，此时，终端按下 CTRL+C 产生 SIGINT 信号

由父进程和子进程所构成的进程组接到这一信号，由于父进程没有捕获该信号，于是执行默认操作，父进程结束，随之关闭了标准输入输出，子进程由于无法从标准输入读入数据，于是也相继退出

另外，无论如何，在 system 执行过程中，不应该让调用进程去获得此时产生的一些信号，比如 SIGCHLD，调用进程会误以为自己的子进程结束，而因此作出处理

在信号的发送与捕捉中，我们对 sleep 函数进行了实现，但是最终，我们的实现仍有问题，那就是在处理我们的 SIGALRM 信号的过程中，如果开始执行其他的信号，之后回到我们的信号处理函数，然后做函数间跳转，进程行为会不可靠、

 sleep 函数

unsigned int sleep(unsigned int seconds);

定义于 unistd.h 中

返回未休眠够的秒数

执行 sleep 后进程被挂起，直到满足下列条件之一：

1.已经过了 seconds 的秒数

2.调用进程捕捉到一个信号并从信号处理程序返回

在 linux 下，使用 alarm 和 signal 或 sigsuspend 可以实现系统的 sleep 功能，可以参看：

[sleep 函数的实现](http://112.126.74.142/article/list/article.php?id=10182537)

但是如果使用 setittimer 与 sigsuspend 可以实现精度较高的定时功能、

函数原型

int setitimer(int which, const struct itimerval \*value,

struct itimerval \*ovalue);

定义于 sys/time.h 中

调用成功返回 0 否则返回 -1

value

value 是一个 itimerval 结构的类型

itimerval 结构如下：

struct itimerval {

    struct timeval it\_interval;

    struct timeval it\_value;

};

it\_interval 指定间隔时间，it\_value 指定初始定时时间，如果只指定 it\_value，则只实现一次定时，否则会在超时后用 it\_value 初始化为 it\_interval 实现重复定时

timeval 结构如下：

struct timeval {

    long tv\_sec;

    long tv\_usec;

};

tv\_sec 提供秒级精度，tv\_usec 提供微秒级精度（1s = 1000000us）

ovalue

ovalue 参数也是一个 itimerval 结构类型参数，用来保存先前的值，通常设为 NULL

信号名字

extern char \*sys\_siglist[];

定义于 signal.h 中

数组下标是信号编号，值是信号名

有一个函数类似于这个数组的作用，用来根据信号编号返回信号名字符串

char \*strsignal(int signo);

信号映射

信号映射函数用来实现在信号编号和信号名之间的转换

int sig2str(int signo, char \*str);

int str2sig(const char \*str, int \*signo);

定义于 signal.h 中

调用成功返回0，否则返回-1

# 第十一章 线程

概念

典型的UNIX进程可以看做只有一个控制线程，即一个进程同一时刻只做一件事

如果进程拥有多个控制线程，则在同一时刻各个线程可以处理各自独立的任务

优势

1.简化编程

2.多进程实现内存和文件描述符共享需要使用操作系统提供的复杂机制，而多个线程则自动的共享地址空间和描述符

3.有些问题中将问题分解可以有效改善程序的吞吐量

4.多线程可以将交互与其他部分分开，有效利用等待的时间片

进程包含了表示进程内执行环境的必要信息，其中包含进程中线程的线程ID、一组寄存器的值、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，进程的所有信息对所有线程都是共享的，包括程序文本、全局内存和堆内存、栈以及文件描述符

线程标识与相关函数

和每个进程都有一个进程ID一样，每个线程也有自己的唯一标识，系统中的每个进程的ID都是不同的，同一进程中的线程ID也是不同的，但是不同进程的线程ID可能相同

进程使用一个整数作为ID（pid\_t类型），而线程则使用一个结构体 pthread\_t 类型作为ID

比较两个线程ID1

int pthread\_equal(pthread\_t tid1, pthread\_t tid2);

定义于 pthread.h 中

若参数中的两个线程ID相同则返回非0值，否则返回0

获取自身的线程ID1

pthread\_t pthread\_self();

定义于 pthread.h 中

一般来说，pthread\_equal 与 pthread\_self 两个函数是一起使用的

进程的主线程常常把新的工作放到工作队列中，由各个线程去工作队列中取出工作并执行，但是线程只应该处理主线程分配给他的工作，于是线程去对比控制线程放在工作中的线程ID去判断是否应该去处理

线程的创建

每个进程开始运行时，都只包含一个控制线程

新增线程需要调用 pthread\_create 函数创建

int pthread\_create(pthread\_t \*restrict tidp,

        const pthread\_attr\_t \*restrict attr,

        void \*(\*start\_rtn)(void), void \*restrict arg);

定义于 pthread.h 中

调用成功返回0，否则返回错误编号

当 pthread\_create 成功返回时，由 tidp 指向的内存单元被设置为新创建的线程ID

attr 用来指定各种不同的线程属性，为 NULL 则创建默认属性线程

新创建的线程从 start\_rtn 回调函数开始执行，并以 arg 作为回调参数，参数虽然只有一个，但是可以为任何类型的指针类型，所以可以创建参数结构体传参

进程在创建线程时无法指定哪个线程先执行

新的线程继承调用线程的浮点环境和信号屏蔽字，但是会清除未决信号集

需要注意的是，pthread\_create 函数调用失败时并不设置 errno，而是返回错误码

每个线程都有自己的 errno 副本，这只是为了与现有函数兼容

如果线程调用 \_exit、exit 或 \_Exit 函数中的某个，则整个进程都会被终止，同样，对于用来终止进程的信号，如果执行其默认动作，则整个进程也会被终止

如果想要结束单个进程，有下列三种方法：

1.线程从启动函数中返回

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 可以主动退出

主动退出 -- pthread\_exit 与 pthread\_join

pthread\_exit

1

void pthread\_exit(void \*rval\_ptr);

定义于 pthread.h 中

rval\_ptr 是一个无类型指针，与传给启动例程的单个参数类似，同一进程中的其他线程可以通过 pthread\_join 函数访问这个指针

pthread\_join

int pthread\_join(pthread\_t thread, void \*\*rval\_ptr);

定义于 pthread.h 中

调用成功返回0，否则返回错误编号

调用该函数的线程将一直阻塞，直到参数中指定的线程调用 pthread\_exit 函数退出

需要注意的是，pthread\_exit 要返回全局结构体变量或者malloc函数分配结构，如果返回局部变量，则栈可能会被其他线程覆盖

取消线程 -- pthread\_cancle 与线程清理处理程序

线程可以请求取消同一进程中的其他线程

int pthread\_cancle(pthread\_t tid);

定义于 pthread.h 中

若成功则返回0，否则返回错误编号

在默认情况下，pthread\_cancle 函数会使由 tid 标识的线程的行为表现为如同调用了参数为 PTHREAD\_CANCLE 的 pthread\_exit 函数

线程也可以选择忽略或控制取消方式

线程可以安排在他退出时需要调用的函数，和进程调用 atexit 函数安排进程退出时需要调用的函数类似

下面的程序就是线程清理处理程序，处理程序记录在栈中，因此，它们的执行顺序与注册顺序相反

void pthread\_cleanup\_push(void (\*rfn)(void \*), void \*arg);

void pthread\_cleanup\_pop(int execute);

当下列情况时，会调用清理函数：

1.调用 pthread\_exit 退出

2.响应取消请求

3.用非0的execute参数调用pthread\_cleanup\_pop

不过 execute 参数如果被设置为0，则不会调用清理函数，不过无论哪种情况，由于 pthread\_cleanup\_pop 函数的调用，都将删除上次 pthread\_cleanup\_push 调用建立的清理处理程序

**分离线程 -- pthread\_detach**

在默认情况下，线程的终止状态会一直保存到对该线程调用 pthread\_join，除非线程已经处于分离状态

处于分离状态的线程在终止时资源会被立即回收

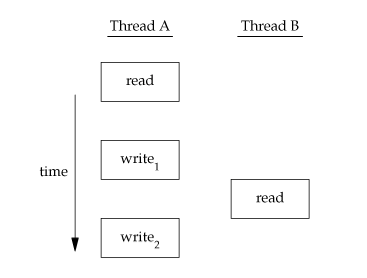
线程可以通过调用 pthread\_detach 使线程处于分离状态

int pthread\_detach(pthread\_t tid);

如若调用成功则会返回0，否则返回错误编号

当多个控制线成共享相同的内存时，需要确保每个线程看到一致的数据视图

下图描述了两个线程读写相同变量的例子：

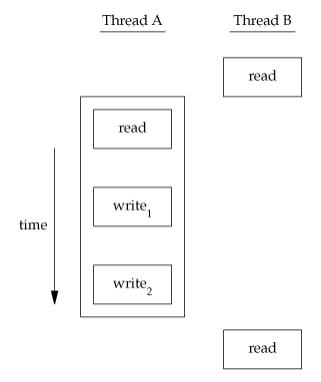


线程A读取变量然后给变量赋予一个新值，然后写入内存

但是，与此同时，B从内存中读取相同变量，由于A尚未将改变后的变量写入内存，所以B读到的是原值

如果此时线程A与线程B都要对内存或寄存器中的变量进行加1操作，由于线程A读到的是原值，所以A写入了原值加1，线程B在线程A写入前读取到该值，也进行加一操作，写入内存，则虽然两个线程对该变量进行了加1操作，事实上改变量却只加了1

解决这个问题的方案是加锁，让线程A在对该变量操作的过程中，不允许线程B对该变量进行操作，如下图所示：



互斥量

如果修改操作是原子性的，则不存在竞争，但是在现代操作系统中，存储器访问总是需要多个总线周期，多处理器系统的总线周期常常是在多个处理器上交叉进行的

因此在线程中使用、修改变量的时候，需要对变量加锁

互斥量从本质上说就是一把锁，在访问资源前对互斥量加锁，访问后释放该锁

当锁释放时，所有等待该锁释放的线程都会变成可运行状态，第一个变为运行状态的线程可以对互斥量加锁，其他线程将会因此而继续等待，直到他继续变为可用，因此，在这种方式下，每次只会有一个线程可以向前执行

互斥量的初始化与销毁

互斥量用 pthread\_mutex\_t 数据类型表示，在使用互斥量以前，必须首先对它进行初始化

可以直接把互斥量置为 PTHREAD\_MUTEX\_INITIALIZER （只对静态分配的互斥量），这种方式称为静态初始化

也可以通过调用 pthread\_mutex\_init 函数进行初始化，这种方式称为动态初始化

动态初始化的互斥量必须在释放内存前调用 pthread\_mutex\_destroy

int pthread\_mutex\_init(pthread\_mutex\_t \*restrict mutex,

        const pthread\_mutex\_t \*restrict attr);

int pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex);

pthread\_mutex\_init 函数的第二个参数 attr 是互斥量属性，使用默认的属性则将 attr 置为 NULL

互斥量的加锁和解锁

int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex);

int pthread\_mutex\_trylock(pthread\_mutex\_t \*mutex);

int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex);

pthread\_mutex\_unlock 函数会在互斥量已经加锁的情况下阻塞线程直到互斥量被释放

pthread\_mutex\_trylock 函数则不会将线程阻塞，如果互斥量是加锁状态，则会返回 EBUSY

避免死锁

如果线程试图对同一个互斥量加锁两次，那么他自身就会陷入死锁

但是，还是有很多更不明显的方式也能产生死锁

当程序中使用多个互斥量，如果允许一个线程一直占有第一个互斥量，并且在试图锁住第二个互斥量时处于阻塞状态，而与此同时，拥有第二个互斥量的线程也在试图锁住第一个互斥量，此时，由于两个线程在相互请求另一个线程占用的资源，导致两个线程都无法向前继续运行，于是就会产生死锁

当然，死锁是可以避免的，例如，如果程序需要对互斥量A和B同时加锁，如果所有线程总是在对互斥量B加锁之前锁住互斥量A，那么使用这两个互斥量不会产生死锁，只有当一个线程试图以另一个线程相反的顺序锁住互斥量时，才可能出现死锁。

读写锁

读写锁也称为“共享-独占锁”

读写锁与互斥量类似，但是读写锁允许更高的并行性，互斥量只有加锁和解锁两种状态，而且每次只能有一个线程对互斥量加锁

而读写锁具有三种状态：读模式加锁、写模式加锁、不加锁，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

写加锁状态的读写锁与互斥量一样，在解锁前所有试图对这个锁加锁的线程都会被阻塞，但读加锁状态的读写锁则允许所有线程对读写锁读加锁

读写锁很适合用在对于数据结构的读次数远大于写次数的情况中

条件变量

条件变量是线程可以使用的另一种同步机制，用于让线程以武警正的方式等待某个特定的条件发生

条件变量也是需要使用互斥量保护的，线程在改变条件状态前，必须先锁住互斥量，其他线程在获得条件变量之前也必须锁住互斥量，这样才不会察觉到这种改

条件变量的初始化与销毁

与互斥量一样，条件变量可以用两种方式初始化

1.静态初始化 -- 直接把常量 PTHREAD\_COND\_INITIALIZER 付给静态分配的条件变量

2.动态初始化 -- 使用 pthread\_cond\_init 函数

对于动态初始化的条件变量，在释放底层的内存空间之前，需要使用 pthread\_cond\_destroy 函数销毁

int pthread\_cond\_init(pthread\_cond\_t \*restrict cond,

        pthread\_condattr\_t \*restrict attr);

int pthread\_cond\_destroy(pthread\_cond\_t \*cond);

定义于 pthread.h 中

调用成功返回0，否则返回错误编号

attr 为 NULL 时创建默认树形的条件变量

等待条件变量

int pthread\_cond\_wait(pthread\_cond\_t \*restrict cond,

pthread\_mutex\_t \*restrict mutex);

int pthread\_cond\_timedwaite(pthread\_cond\_t \*restrict cond,

pthread\_mutex\_t \*restrict mutex,

const struct timespec \*restrict timeout);

struct timespec

{

    time\_t  tv\_sec;     /\* seconds \*/

    time\_t  tv\_nsec;    /\* nanoseconds \*/

}

定义于 pthread.h 中

调用成功返回0，否则返回错误编号

pthread\_cond\_wait 函数会等待直到条件变为真，如果超时，则返回一个出错码 -- ETIMEDOUT

pthread\_cond\_timedwaite 允许设定等待超时

1.等待函数有下列几点需要注意

2.传递给等待函数的互斥量必须是锁住状态的，函数原子性的解锁、测试，这样线程就不会错过条件的任何变化，当等待函数返回时，互斥量将被再次锁住

3.pthread\_cond\_timedwaite 的参数 timespec 结构是绝对时间而不是相对时间，比如需要等待三分钟，则需要将当前时间加上3分钟传递给 timespec 结构

下面的函数实现了到相对时间到 timespec 结构的转化

void maketimeout(struct timespec \*tsp, long minutes)

{

    struct timeval now;

    gettimeofday(&now);

    tsp->tv\_sec = now.tv\_sec;

    tsp->tv\_nsec = now.tv\_usec \* 1000;

    tsp->tv\_sec += minutes \* 60;

}

唤醒线程

int pthread\_cond\_signal(pthread\_cond\_t \*cond);

int pthread\_cond\_broadcast(pthread\_cond\_t \*cond);

定义于 pthread.h 中

调用成功返回0，否则返回错误编号

这两个函数用于通知线程条件已经满足

pthread\_cond\_signal 函数将唤醒某个等待该条件的线程，pthread\_cond\_broadcast 则唤醒所有等待该条件的线程

需要注意的是，一定要在改变条件状态以后再给线程发送信号

# 第十三章 守护进程

概述

在Unix 环境下，守护进程是相当重要的，大多数提供服务的进程都是以这种方式运行的

他们独立于控制终端，并且循环、周期的执行工作，完成了很多系统日常性的管理功能和各种服务，例如 apache 的 httpd 服务、nginx 的 master 进程、执行日常工作的 crontab 进程、打印进程 lpd 等

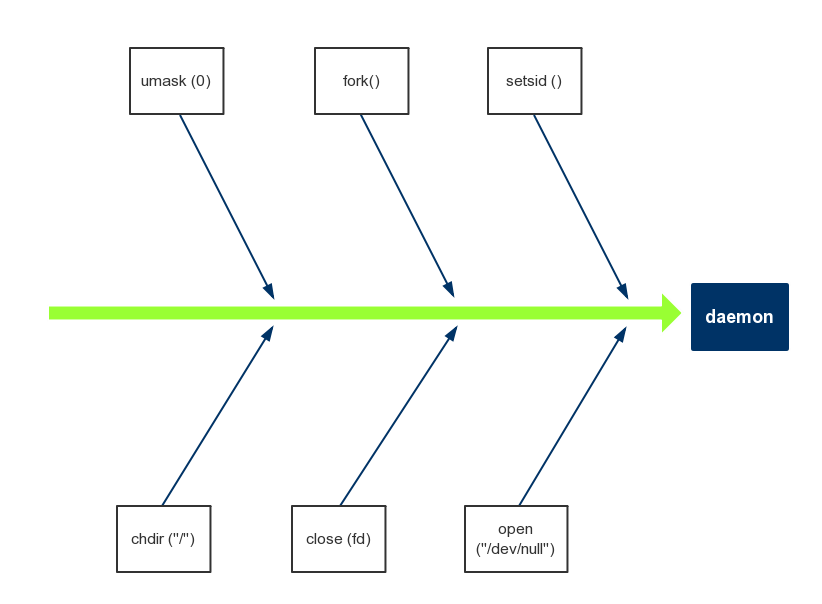
守护进程的特性

守护进程是一种纯粹的后台进程，与运行前环境完全隔离，包括未关闭的文件描述符、控制终端、会话、进程组、工作目录以及文件创建掩码等

很多守护进程是父进程 fork 产生，所以会继承所有的父进程地址空间中的环境，所以必须在守护进程诞生之初，断绝这些相关环境，当然，守护进程也可以在 linux 系统启动时从启动脚本 /etc/rc.d 中启动，也可以由 crontab 启动

事实上，守护进程与普通进程的编写并没有特别大的区别

守护进程的启动



后台运行

让守护进程在后台启动最简单的方式就是通过父进程 fork 然后让父进程退出

这样做的好处是容易实现，同时父进程还可以在此之前进行一些必要的初始化工作，最重要的是，子进程的进程组ID是父进程ID，这样保证了子进程不是组长进程，组长进程是无法通过调用 setsid 脱离控制终端的

nginx 就是采用这样的方式启动守护进程的

脱离终端

在linux中，进程属于一个进程组，进程组号（GID）就是进程组长的进程号 （PID）。登录会话可以包含多个进程组。这些进程组共享一个控制终端。这个控制终端通常是创建进程的登录终端。 控制终端，登录会话和进程组通常是从父进程继承下来的。

守护进程必须摆脱他们，不受他们的影响

因此，需要调用系统调用 setsid 使进程成为会话组组长

在此之后，进程成为新的会话组组长，与原来的会话组和进程组脱离关系

在 linux 中，终端会话组必须独占终端，因此，已经脱离终端会话组的进程也随即与控制终端脱离关系

当然，作为会话组长，还是可以再打开一个新的控制终端的

重设文件创建掩码

进程从创建他的父进程那里继承了文件创建掩码，它可能会修改守护进程创建的文件的存取位

因此需要调用：

umask(0)

如果需要在 fork 后立即改变其文件掩码，则最好在父进程 fork 前调用

关闭打开的文件描述符

进程从创建它的父进程那里继承了打开的文件描述符

如不关闭，将会浪费系统资源，造成进程所在的文件系统无法卸下以及引起无法预料的错误

复制标准输出、标准输入、错误输出文件描述符

同时，为了标准输入、标准输出、错误输出的正确使用，需要调用系统调用 dup2 复制这些文件描述符

if (dup2(fd, STDIN\_FILENO) == -1) {

    ngx\_log\_error(NGX\_LOG\_EMERG, log, ngx\_errno, "dup2(STDIN) failed");

    return NGX\_ERROR;

}

if (dup2(fd, STDOUT\_FILENO) == -1) {

    ngx\_log\_error(NGX\_LOG\_EMERG, log, ngx\_errno, "dup2(STDOUT) failed");

    return NGX\_ERROR;

}

也可以通过 open("/dev/null")，然后使用 dup2 让标准输入、标准输出、错误输出重定向到空文件

改变当前工作目录

守护进程一般还需要改变当前工作目录到根目录或特定目录，当然，也可以保持在原来目录，调用：

int chdir (const char \*path);处理信号

虽然信号处理不是必须的，但是为了让守护进程能够稳定运行，屏蔽一些信号还是很有必要的

同时，如果不对 SIGCHLD 信号进行处理，子进程退出后会变成僵尸进程，从而占用系统资源

在 nginx 中，守护进程在 fork 子进程前，屏蔽了 SIGCHLD、SIGALRM、SIGIO、SIGINT、SIGHUP、SIGINFO、SIGWINCH、SIGTERM、SIGQUIT、SIGXCPU 等信号

示例

#include <stdio.h>

#include <fcntl.h>

#include <unistd.h>

#include <stdlib.h>     // for exit

#include <signal.h>     // for sigaction

#include <syslog.h>     // for syslog

#include <sys/stat.h>   // for umask

#include <sys/resource.h>

void daemonize (const char \*cmd)

{

    int i, fd0, fd1, fd2;

    pid\_t  pid;

    struct rlimit rl;

    struct sigaction sa;

    // 更改文件权限屏蔽字 => 不屏蔽

    umask(0);

    if (getrlimit(RLIMIT\_NOFILE, &rl) < 0)

    {

        perror("getrlimit");

        exit(-1);

    }

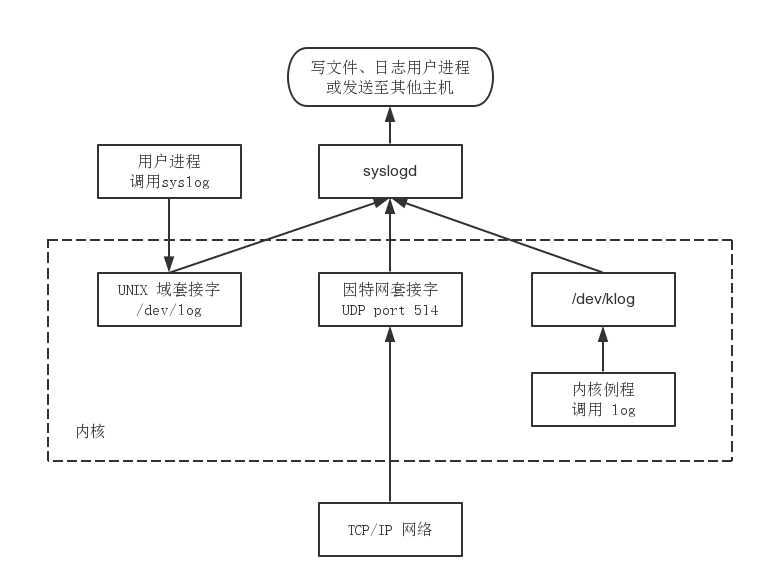
    if ((pid = fork()) < 0)

可以看到，TTY 一项中，值为?，这意味着，该进程是没有对应的终端的

同时，并不存在一个进程ID为 20121 的进程，这意味着，该进程为孤儿进程，也因此不会有任何机会被分配到一个新的控制终端，这得益于我们函数中的第二个 fork 函数

对于很多程序来说，我们非常关心他们的运行状态，特别是守护进程，他们无法通过标准输入、标准输出来反馈他们的运行状态，所以日志就显得格外重要

最早出现在BSD上的 syslog 完成了这样的工作，如今，这个函数已经成为 POSIX 的一部分



如图，有三种方式可以操作 syslog 打印日志

1.内核通过调用 log 函数写入日志到 /dev/klog

2.大多数用户进程可以通过调用 syslog 函数产生日志消息，该消息通过调用序列发送至 UNIX 域套接字 /dev/log

3.通过 TCP/IP 连接到本机的其他机器进程或本机进程可以通过 UDP 端口 514 发送日志消息

守护进程 syslogd 在启动时读取某个配置文件，决定不同类型的消息应送往何处

函数简介

void openlog(const char \*ident, int option, int facility);

void syslog(int priority, const char \*format, ... );

void vsyslog(int priority, const char \*format, va\_list arg);

void closelog(void);

int setlogmask(int maskpri); // 返回之前日志记录优先级屏蔽值

定义于 syslog.h 中

openlog 是可选的，同样 closelog 也是可选的，用来关闭曾被用于与 syslogd 通信的描述符

setlogmask 函数用于设置进程的记录优先级屏蔽字，返回调用之前的屏蔽字

只有大于当前优先级屏蔽字的消息才会被记录到日志，其他消息不会被记录

# 第十四章 高级IO

非阻塞IO

我们可以将系统调用分成低速系统调用和其他系统调用两类

低速系统调用有可能会使进程永远阻塞，低速系统调用包括下列情况：

1.如果某些文件类型的数据不存在，则读操作的调用者可能永远阻塞

2.如果数据不能立即被同样类型的文件接受（如管道中无空间、网络流控制等），则写操作的调用者可能永远阻塞

3.在某种条件发生之前，打开某些类型的文件会被阻塞（如打开一个终端设备可能需要等到与之连接的调制解调器应答，又例如在没有其他进程已用读模式打开该FIFO时若以致谢模式打开FIFO，则也需要等待）

4.对已经加上强制性记录锁的文件进行读写

5.某些 ioctl 操作

6.某些进程间通信函数

虽然读写磁盘文件会使调用者在短时间内被阻塞，但是并不能将与磁盘IO相关的系统调用视为“低速”系统调用

非阻塞调用

1.对于一个给定描述符，有两种方法可以指定为非阻塞IO

2.如果调用 open 获得描述符，则可制定 O\_NONBLOCK 标志

3.对于已经打开的一个描述符，则可调用 fcntl 打开 O\_NONBLOCK 标志

锁的隐含继承和释放

关于记录锁的自动继承和释放遵循下列三条规则：

1.锁与进程和文件两方面有关：当一个进程终止时，他所建立的所有锁将全部释放，而关闭一个描述符时，则该进程通过这一描述符所引用的文件上的所有的锁都将被释放（通过 dup 复制的描述符上设置的锁也将全部被释放）

2.由 fork 产生的子进程不继承父进程锁设置的锁

3.在执行 exec 后，新程序可以继承原执行程序的锁（如果文件描述符已经设置了 close-on-exec 标志，则相应文件上的所有锁均会被释放）

在文件尾端加锁

writew\_lock(fd, 0, SEEK\_END, 0);

write(fd, buf, 1);

un\_lock(fd, 0, SEEK\_END, 0);

write(fd, buf, 1);

上述代码做了什么呢？

首先，在文件尾端加了一把写锁，这样的锁包括以后可能添加到该文件的任何数据，这之后，写入一个字符后，该字符也随即被加上写锁

随后，进行了一次从尾端开始的解锁操作，从此以后新添加的字符都不会被加锁

但是，需要注意的是，此前锁住的 1 个字符，此时已然处于写锁定的状态

unix 与记录锁

当一些 unix 命令执行时，他们会对文件加锁，具体会出现下列情况：

1.使用 ed 编辑程序可以对已加锁文件进行编辑，因为 ed 的实际操作是写入临时文件，在保存时，用临时文件替换原文件，因此锁对 ed 命令毫无影响，同样的，锁也不会对 unlink 调用产生任何影响

2.不能使用 vi 编辑程序编辑已加锁文件，可读，但是不可写入

STREAMS 就是流

流在用户进程和设备驱动程序之间提供了一条全双工通道，从而无需和实际的硬件设备直接会话，也因此，流可以用来构造伪设备驱动程序

任意数量的处理模块可以被压入流，流类似于栈，模块以先入后出的策略被压入。

写到流首的数据顺流而下传送，由设备驱动程序读到的数据你流向上传送

整个 STREAMS 类似于设备驱动程序，是作为内核的一部分执行的

linux 以可选择包方式提供支持，而其他系统暂不支持

流的访问

正如之前介绍的，可以通过 open、close、read、write 和 ioctl 函数访问流，流设备通常存放在 /dev 目录下，但我们并不能直观的知道 /dev 目录下哪些设备是流设备

getmsg、putmsg 和 poll 是在 SVR3 内核中新增的流访问函数

getpmsg、putpmsg 是 SVR4 内核中新增的处理流内不同优先级波段消息的函数

存储映射 IO 使一个磁盘文件与存储空间中的一个缓冲区想映射，于是当从缓冲区中取数据时，就相当于读文件中的相应字节，同样，将数据写入缓冲区则相应字节被自动写入文件，这样就可以在不使用 read 和 write 的情况下执行 IO 操作了

存储映射

mmap 函数用来声明一个映射，即告知内核将一个给定的文件映射到一个存储区域中

1

void \*mmap(void \*addr, size\_t len, int prot, int flag, int filedes, off\_t off);

定义于 sys/mman.h 中

调用成功返回映射区域的首地址，否则返回 MAP\_FAILED

参数说明

addr

addr 参数用于指定映射存储区的起始地址，如果设为 0，则由系统选择该映射区的起始地址

len

映射字节数

off

要映射字节在文件中的起始偏移量

通常应当是系统虚存页表长度的倍数

虚存页表长度可以通过 \_SC\_PAGESIZE 或 \_SC\_PAGE\_SIZE 参数调用 sysconf 函数获取

filedes

被映射文件的文件描述符

prot

对映射存储区的保护要求

调用中，可以指定为 PROT\_NONE 或者 PROT\_READ、PROT\_WRITE、PROT\_EXEC 的任意组合的按位或

但是，需要注意的是，对指定映射存储区的保护要求不能超过文件 open 模式访问权限，如某文件是只读打开，则对映射存储区就不能指定 PROT\_WRITE

mmap 与信号 -- SIGSEGV & SIGBUS

当进程试图访问对它不可用的存储区时，会抛出 SIGSEGV 信号

如果进程访问的存储区不存在，则抛出 SIGBUS 信号

更改映射存储区的权限 -- mprotect

int mprotect(void \*addr, size\_t len, int prot);

定义于 sys/mman.h 中

调用成功返回 0，否则返回 -1

冲洗缓存到文件中 -- msync

int msync(void \*addr, size\_t len, int flags);

定义于 sys/mman.h 中

调用成功返回 0，否则返回 -1

如果是通过 MAP\_PRIVATE 标志创建的缓存，则调用 msync 不会改变文件内容

解除映射 -- munmap

int munmap(caddr\_t addr, size\_t len);

定义于 sys/mman.h 中

调用成功返回 0，否则返回 -1

munmap 不会影响被映射对象，即 munmap 函数不会将映射区内容写到磁盘文件上

POSIX AIO -- glibc 版本异步 IO 简介

linux 中最常用的 IO 模型是同步 IO，在这个模型中，请求发出后应用程序会阻塞直到满足条件（阻塞 IO），或在不满足条件的情况下立即返回出错（非阻塞 IO），这样做的好处是程序在等待 IO 请求完成时不会占用 CPU

POSIX 定义了异步 IO 应用程序接口（AIO API），linux 2.6 以上版本的内核也实现了内核级别的异步 IO 调用

异步 IO 的基本思想是允许进程发起很多 IO 操作，而不用阻塞任何一个，也不用等待任何操作的完成，直到 IO 操作完成时，进程可以检索 IO 操作的结果

linux 下主要有两套异步 IO，分别是 glibc 实现版本，和 linux 内核实现、libaio 封装的版本

IO 模型简介

write、read 如果没有设置 O\_NONBLOCK 标识则为同步阻塞式 IO，一旦 IO 发起，则进程一直等待直到操作完成

设置了 O\_NONBLOCK 标识后，write、read 成为非阻塞 IO，调用后如果资源可用则进行操作，并立即返回，如果资源不可用则直接返回出错，这样的情况下，程序通常需要进入忙等待状态，反复调用 IO 操作，直到正常返回

以上两种 IO 模型虽然可以很好地完成单机的 IO 操作，但是对于并发的请求则无法实现

对于并发的多个请求，可以使用 IO 复用模型，如 select、poll、epoll 等，但是进程必须阻塞直到操作完成

如果需要进行并发、非阻塞的 IO 操作，比如 CPU 密集型应用及较慢的 IO 操作应用场景下，使用异步 IO 是一个很好地选择

函数说明

aio\_read、aio\_write

12

int aio\_read( struct aiocb \*aiocbp );

int aio\_write( struct aiocb \*aiocbp );

调用成功返回 0，失败返回 -1 并设置 errno

将请求添加到 request\_queue

通过参数 aiocbp 指向的结构可以设置文件描述符、文件偏移量、缓冲区及大小等属性，函数执行后立即返回

对于 aio\_write，如果设置了 O\_APPEND，则文件偏移量属性会被忽略

aio\_error

1

int aio\_error( struct aiocb \*aiocbp );

用于查询请求的状态

aio\_return

ssize\_t aio\_return( struct aiocb \*aiocbp );

获取异步 IO 返回值

调用成功返回读写的字符数，出错返回 -1

aio\_suspend

int aio\_suspend( const struct aiocb \*const cblist[],

        int n,

        const struct timespec \*timeout );

阻塞进程，直到列表中的某个异步请求完成

cblist 中任何一个异步请求完成，函数都会返回 0，出错返回 -1

aio\_cancel

int aio\_cancel( int fd, struct aiocb \*aiocbp );

取消一个异步请求，第二个参数为 NULL 则取消所有该 fd 上的异步请求

成功取消返回 AIO\_CANCELED，请求已经完成则返回 AIO\_NOTCANCELED

在取消多个请求的情况下，如果至少有一个请求没有被取消，则返回 AIO\_NOT\_CANCELED，如果没有一个请求可以被取消，则返回 AIO\_ALLDONE

lio\_listio

int lio\_listio( int mode,

        struct aiocb \*list[],

        int nent,

        struct sigevent \*sig );

同时发起多个异步请求，可以很大程度上提高系统的性能

mode 参数可选 LIO\_WAIT 或 LIO\_NOWAIT 来声明该函数是否阻塞

nent 参数定义了 list 列表的最大元素个数

list 列表中可以有值为 NULL 的请求，则该请求被忽略

sigevent 的指针定义了在所有 IO 操作都完成时产生的信号或调用的回调函数

# 第十五章 进程间通信

进程间通信

最简单的进程之间传递信息的方式是通过 fork 或 exec 传送已打开的文件，或两进程通过打开相同的文件来进行数据的共享

进程之间通信还有其他技术 -- IPC：

1.半双工管道

2.FIFO

5.全双工管道

6.命名全双工管道

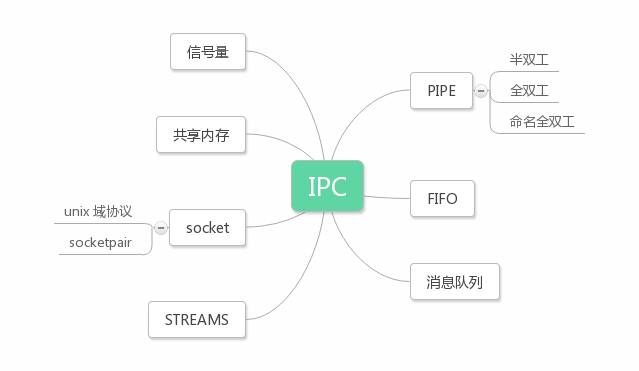
7.消息队列

8.信号量

9.共享内存

10.socket

11.STEAMS



管道

管道是 UNIX 系统 IPC 的最古老形式，所有的 UNIX 系统都支持这种通信机制，所以也是最常用的 IPC 形式

最简单的管道应用的例子就是：

1

ps aux | awk '{print $2}' | grep 'ngxin'

但管道有以下两个局限性：

1.管道是半双工的，虽然大部分系统已经经由域套接字支持了全双工管道和命名全双工管道，但是 linux 系统知识采取可选包方式提供支持

2.只能在具有公共祖先的进程之间使用

管道的用法

通常管道都是有一个进程创建，然后通过 fork 使父子进程通过该管道通信。

一个管道通常只有一个读进程与一个写进程

如果写一个读端已经关闭的管道，则会产生 SIGPIPE 信号，write 函数返回 -1，errno 被置为 EPIPE

在写管道（或 FIFO）时，常量 PIPE\_BUF 规定了内核中管道缓冲区的大小，可以通过 pathconf 或 fpathconf 函数确定 PIPE\_BUF 的值

具体使用的例子可以参考：

超轻量服务器 tinyhttpd 源码解析

在这个超轻量服务器中，主进程 fork 出 cgi 执行程序，就是通过管道接收 cgi 进程的输出的

管道的创建 -- pipe

1

int pipe(int filedes[2]);

调用成功返回 0，否则返回 -1

参数 filedes 用于返回

filedes[0] 用于读取管道中数据

filedes[1] 用于将数据写入管道

使用 fstat 函数对管道的每一端检测（使用 S\_ISFIFO 宏），会发现返回的是 FIFO 类型的文件描述符

popen 和 pclose 函数

函数说明

标准IO库提供了管道的创建和销毁的函数

1

2

FILE \*popen(const char \*cmdstring, const char \*type);

int pclose(FILE \*fp);

定义于 stdio.h

popen 返回创建的文件指针，如果调用失败返回空指针

返回 cmdstring 的终止状态，出错返回 -1

函数作用

函数 popen 先执行 fork，然后调用 exec 执行 cmdstring 保存的命令，并且返回一个标准 IO 文件指针

如果 type 参数为 "r"，则文件指针连接到 cmdstring 的标准输出，

如果 type 参数为 "w"，则文件指针连接到 cmdstring 的标准输入，

使用 popen 和 pclose 可以减少需要编写的代码量

协同进程

在具体的编程环境中，常常使用一种程序设计的方式，即某个进程向另一个进程的标准输入写入数据，同时又接收该进程的标准输出，则这里所说的“另一个”进程就是一个“协同进程”

典型的是在 http 服务器中，服务器 fork 子进程执行 cgi 程序，通过 cgi 程序的标准输入向他写入调用参数，并且通过 cgi 程序写入标准输出的数据返回调用主进程

可以参考 tinyhttpd 的源码：

[超轻量服务器 tinyhttpd 源码解析](http://112.126.74.142/article/list/article.php?id=10182680)

在这个例子中，进程创建了两个管道，一个是协同进程的标准输入，一个是协同进程的标准输出，然后 fork 出 cgi 进程，并通过这两个管道与 cgi 进程通信。

server端编程

#include "function.h"

void str\_echo (int sockfd)

{

ssize\_t n;

char buf[MAXLINE];

while (1)

{

while ((n = read (sockfd, buf, 1)) > 0)

{

printf("n:%d\n", n);

if (writen (sockfd, buf, n) < 0)

return;

sleep(1);

}

if (n < 0 && errno == EINTR)

continue;

else if (n < 0)

{

perror ("read error: ");

return;

}

}

}

int get\_listenfd()

{

struct sockaddr\_un servaddr;

int listenfd = socket(AF\_LOCAL, SOCK\_STREAM, 0);

if (listenfd < 0)

{

perror("can't create socket file");

exit(-1);

}

unlink(UNIXSTR\_PATH);

memset(&servaddr, 0, sizeof(servaddr));

servaddr.sun\_family = AF\_LOCAL;

memcpy(servaddr.sun\_path, UNIXSTR\_PATH, strlen(UNIXSTR\_PATH)+1);

if (bind(listenfd, (struct sockaddr \*)&servaddr, sizeof(struct sockaddr))

< 0)

{

perror("bind");

exit(-1);

}

signal(SIGCHLD, SIG\_IGN);

if (listen(listenfd, LISTENQ) < 0)

{

perror("listen");

exit(-1);

}

return listenfd;

}

ssize\_t writen (int fd, const void \*vptr, size\_t n)

{

size\_t nleft;

ssize\_t nwritten;

char \*ptr;

int i;

ptr = (char \*)vptr;

nleft = n;

for (i=0; i<n; ++i)

{

if (ptr[i] >= 'a' && ptr[i] <= 'z')

ptr[i] += 'A'-'a';

}

while (nleft > 0)

{

if ((nwritten = write (fd, ptr, nleft)) <= 0)

{

if (nwritten < 0 && errno == EINTR)

nwritten = 0;

else

{

perror ("write error: ");

return -1;

}

}

nleft -= nwritten;

ptr += nwritten;

}

return n;

}

待续......真的抱歉，学校事情太多了。我看不完了。公司是为了我们好，读读圣经确实很好，但是学校并不是清闲的。抱歉！