Linux 环境编程

1．编译器 gcc

gcc(GNU C Compiler)是 GNU 推出的功能强大、性能优越的多平台编译器，是 GNU 的代表作品之一。gcc 编译器能将 C、C++语言源程序和目标程序编译、连接成可执行文件。

在 Linux 系统中，可执行文件没有统一的后缀，系统根据文件属性来区分文件，gcc 通过后缀来区别输入文件的类别。gcc 遵循的部分约定规则如下表所示。

gcc 遵循的部分约定规则

文件后缀	文件内容
.c	C 语言源代码文件
.a	由目标文件构成的档案库文件
.C，.cc 或.cxx	C++源代码文件
.h	程序所包含的头文件
.i	已经预处理过的 C 源代码文件
.ii	已经预处理过的 C++源代码文件
.m	Objective-C 源代码文件
.o	编译后的目标文件
.s	汇编语言源代码文件
.S	经过预编译的汇编语言源代码文件

在使用 gcc 编译器的时候，必须给出一系列必要的调用参数和文件名称。gcc 命令提供了非常多的命令选项，大约有 100 多个，但并不是所有都要熟悉，多数参数可能根本就用不到，初次接触时掌握常用的几个，后面随着实验的深入再慢慢学习并掌握。

（1） gcc 最基本的用法

gcc 最基本的用法是：gcc [options] [filenames]

其中 options 是编译需要的参数，filenames 给出相关的文件名称。其中[options]的值可以为下列值：

-c，只编译，不连接成可执行文件，只是由输入的.c 等源代码文件生成.o 后缀的目标文件，通常用于编译不包含主程序的子程序文件。

-o output_filename，确定输出文件的名称为 output_filename，该名称不能和源文件同名。如果不带这个选项，gcc 就给出预设的可执行文件 a.out。

-O，对程序进行优化编译、连接，采用该选项，整个源代码会在编译、连接过程中进行优化处理，提高可执行文件的执行效率，但编译、连接的速度就相应地慢一些。

-Idirname，在预编译过程中使用，将 dirname 所指出的目录加到程序头文件目录列表中。

-Ldirname，在连接过程中使用，将 dirname 所指出的目录加到程序函数档案库文件的目录列表中。

-lname，在连接时，装载名为“libname.a”的函数库，该函数库位于系统预设的目录或者由-L 选项确定的目录下。例如，-lm 表示连接名为“libm.a”的数学函数库。

2．文件操作

在 Unix/Linux 系统中，文件概念涉及的内容很广，既包含普通文件、也包含目录、硬件设备。对普通文件的操作包括创建、打开、读/写、关闭；对目录的操作包括创建、删除、改变目录、链接、更改权限；对设备的操作包括打开、读/写、关闭等。

有关文件操作的系统调用有 creat、open、close、read、write、lseek、fstat, stat、lstat、dup、dup2 等。

(1) write 系统调用

size_t write(int filedes, const void *buf, size_t nbytes);

把缓冲区的前 n bytes 个字节写入到文件描述符 filedes 关联的文件中。它返回实际写入的字节数。如果返回值为 0，就表示未写出任何的数据；如果是−1，就表示在 write 调用中出现了错误，对应的错误代码保存在全局变量 errno 中。

(2) read 系统调用

size_t read(int filedes, void *buf, size_t nbytes)

从文件描述符 filedes 相关联的文件里读入 nbytes 个字节的数据，并把它们放到数据区 buf 中。它返回实际读入的字节数，可能会小于请求的字节数。如果是 0，表示未读入任何数据就到了文件尾。如果是−1，表示 read 调用出现了错误，并在 errno 中设置错误代码。

(3) open 系统调用

int open(const char *path, int oflags);

int open(const char *path, int oflags, mode_t mode);

open 建立一条到文件或设备的访问路径。如果操作成功，将返回一个文件描述符。如果两个程序同时打开同一个文件，会得到两个不同的文件描述符，如果它们对文件进行写操作，数据不是交织在一起，而是彼此相互覆盖。为了防止出现这种冲突的局面，可以使用文件锁功能。open 调用成功时返回一个新的文件描述符，失败时返回−1，并设置全局变量 errno 以指明失败的原因。新文件描述符总是使用未用描述符的最小值。

(4) close 系统调用

int close(int filedes);

close 调用终止一个文件描述符 filedes 与其对应文件之间的关联。文件描述符被释放并能够重新使用。close 调用成功就返回 0，出错就返回−1。

(5) lseek 系统调用

off_t lseek(int filedes, off_t offset, int whence);

lseek 调用对文件描述符 filedes 的读写指针进行设置。Lseek 返回从文件头到文件指针被设置处的字节偏移量，失败时返回−1。

(6) fstat, stat 和 lstat 系统调用

int fstat(int filedes, struct stat *buf);

int stat(const char* path, struct stat *buf);

int lstat(const char *path, struct stat *buf)fstat;

系统调用返回与打开的文件描述符相关的文件状态信息，该信息将会写入 buf 结构。

(7) dup 和 dup2 的系统调用

int dup(int filedes); int dup2(int filedes, int filedes2);

dup 系统调用提供了复制文件描述符的一种方法。

3．进程控制

Linux 是一个多用户、多任务的操作系统。在 Linux 系统中，对各种计算机资源(如文件、主存、CPU 等)的分配和管理都以进程为单位。为了协调多个进程对这些共享资源的访问，操作系统要跟踪所有进程的活动，以及它们对系统资源的使用情况，从而实施对进程和资源的动态管理。

有关进程控制的系统调用有 fork、exec、wait、exit、getpid、nice 等。

(1)创建和修改进程

Linux 系统中有两个基本的操作用于创建和修改进程：函数 fork()用来创建一个新的进程，该进程几乎是当前进程的一个完全拷贝；函数族 exec()用来启动另外的进程以取代当前运行的进程。

fork()用来创建一个子进程。该函数被调用一次，但返回两次。两次返回的区别是子进程的返回值是 0，而父进程的返回值则是新进程(子进程)的进程 id。将子进程 id 返回给父进程的理由是：因为一个进程的子进程可以多于一个，没有一个函数使一个进程可以获得其所有子进程的进程 id。对子进程来说，之所以调用 fork 后返回 0，是因为子进程随时可以调用 getpid()来获取自己的 pid；也可以调用 getppid()来获取父进程的 id。

fork 之后，操作系统会复制一个与父进程完全相同的子进程，这两个进程共享代码空间，但数据空间互相独立，子进程数据空间中的内容是父进程的完整拷贝，指令指针也完全相同，子进程拥有父进程当前运行的位置(两进程的程序计数器 pc 值相同，即子进程从 fork 返回处开始执行)。但有一点不同，如果 fork 成功，子进程中 fork 的返回值是 0，父进程中 fork 的返回值是子进程的进程号，如果 fork 不成功，父进程会返回错误。

fork 创建一个新的进程就产生了一个新的 PID，exec 启动一个新程序替换原有的进程，因此这个新的被 exec 执行的进程的 PID 不会改变，和调用 exec 函数的进程一样。

exec()不是函数，而是一个由六个调用函数组成的函数族，描述如下。

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

exec()根据所指定的路径或文件名找到要加载的程序，将其加载并执行，这样函数就通过 exec()替换了原来的内容，变成了所加载的内容，除了外部接口比如 ID 保持一致外，子进程已经变成了与父进程内容不同的新进程。在 exec()函数中还可以添加命令行参数用来传递给子进程。

(2)设置进程属性

设置进程属性主要是修改进程 PCB 中的进程属性。

nice()函数用来改变进程的执行优先级，其参数顺序越大优先级越低。只有超级用户才能使用负的优先级。函数 setpgid()将 pid 所指定的进程的组进程设置为 pgid 指定的组识别码，若 pid 为 0，则设置当前进程的组进程识别码， 若 pgid 为 0，则会以目前进程的进程识别码来取代。函数 setpgrp()用来将目前进程的组进程识别码设置为目前进程的进程识别码,等价于 setpgid(0,0)。setpriority()用来设置进程，进程组和用户的进程执行优先权。

(3)获取进程属性

获取进程信息的函数其功能是读取进程控制块 PCB 中的信息，与属性设置函数相对应。

getpid()用来获取目前进程的进程标识。getpgid()用来获得参数 pid 指令进程所属于的组

识别号，若参数为 0，则返回当前进程的组识别码。getpgrp()用来获得目前进程所属于的组识别号，等价于 getpgid(0)。getpriotity(void)用来获得进程，进程组和用户的进程执行优先权。

(4)进程的退出

进程的退出表示进程即将结束运行，Linux 系统中分为正常退出和异常退出两种。其中正常退出的方法有三种，分别是在 main()函数中执行 return、调用 exit()函数和调用_exit()函数。异常退出的方法有调用 abort()函数和进程收到某个信号而使进程终止两种。不管是哪种方式退出，最终都会执行内核中同一段代码。这段代码用来关闭进程所有已打开的文件描述符，释放它所占用的主存和其他资源。

父子进程终止的先后顺序不同会产生不同的结果。在子进程退出前父进程退出，则系统会让 init 进程接管子进程。当子进程先于父进程终止，而父进程又没有调用 wait()函数等待子进程结束，子进程进入僵死状态，并且会一直保持下去除非系统重启。子进程处于僵死状态时，内核只保存该进程的一些必要信息以备父进程所需。此时子进程始终占用着资源，同时也减少了系统可以创建的最大进程数。如果子进程先于父进程终止，且父进程调用了 wait()或 waitpid()函数，则父进程会等待子进程结束。

4．进程通信

进程间通信就是在不同进程之间传播或交换信息，进程间通信的目的在于实现数据传输、数据共享、事件通知、资源共享等。Linux 进程间通信(IPC)主要由早期 UNIX 进程间通信、基于 System V 进程间通信和 POSIX 进程间通信等几部分发展而来。

现在 Linux 使用的进程间通信方式主要有以下几种。

(1)管道(pipe)和有名管道(named Pipe)

1)无名管道的使用

系统调用 pipe 用来创建无名管道，与之相关的 pipe()函数定义在头文件 unistd.h 中，它的一般形式为：int pipe(int filedes[2])。

一个进程在由 pipe()创建管道后，一般再创建一个子进程，然后通过管道实现父子进程间的通信。pipe 系统调用需要打开两个文件，文件标识符通过参数传递给 pipe()函数。管道两端固定了任务，即一段只能用于读，用文件描述符 filedes[0]表示，称为管道读端；另一端只能用于写，用 filedes[1]表示，称为管道写端。调用成功时，返回值为 0，错误时返回−1。

管道的工作方式可以总结为以下 3 个步骤：

① 将数据写入管道

将数据写入管道利用文件系统的系统调用 write(pipeID[1], buf, size)，把 buf 中长度为 size 的消息写入管道写入口 pipeID[1]。与文件不同的是，管道长度受到限制，管道满时写入操作将被阻塞。

执行写操作的进程进入睡眠状态，直到管道中的数据被读取。fcntl()函数可将管道设置为非阻塞模式，管道满时，write()函数的返回值为 0。如果写入数据长度小于管道长度，则要求一次写入完成。如果写入数据长度大于管道长度，在写完管道长度的数据时，write()函数将被阻塞。

② 从管道读取数据

读取数据利用文件系统的系统调用 read(pipeID[0]，buf，size)，从管道读出口 pipeID[0]中把长度为 size 的消息读出到 buf 中。读取的顺序与写入顺序相同。当数据被读取后，这些数据将自动被管道清除。因此，使用管道通信的方式只能是一对一，不能由一个进程同时向多个进程传递同一数据。如果读取的管道为空，并且管道写入端口是打开的，read()函数将被阻塞。读取操作的进程进入睡眠状态，直到有数据写入管道为止。fcntl()函数也可将管道读取模式设置为非阻塞。

③ 关闭管道

管道虽然有 2 个端口，但只有一个端口能被打开，这样避免了同时对管道进行读和写的操作。关闭端口使用的是 close()函数，关闭读端口时，在管道上进行写操作的进程将收到 SIGPIPE 信号。关闭写端口时，进行读操作的 read()函数将返回 0。

管道为一临界资源，使用过程中父子进程之间除了需要读写同步以外，在对管道进行读写操作时还需要互斥进入，可以使用对文件上锁和开锁的系统调用 lockf(files, function, size)。其中，files 是管道的读写端口；function 是功能选择，为 1 表示上锁，为 0 表示开锁；size 表示锁定或开锁的字节数，其值为 0 则表示文件全部内容。

2)命名管道的创建与读写

创建命名管道的系统函数有两个：mknod 和 mkfifo。两个函数均定义在 sys/stat.h 中，函数原型如下：

int mknod(const char *path, mode_t mod, dev_t dev)；

int mkfifo(const char *path, mode_t mode)；

函数 mknod 参数中 path 为创建的命名管道的全路径名；mod 为创建的命名管道的模式，指明其存取权限；dev 为设备值，该值取决于文件创建的种类，只在创建设备文件时才会用

到。这两个函数调用成功都返回 0，失败都返回−1。函数 mkfifo 前两个参数的含义和 mknod 相同。

命名管道创建后就可以使用了，命名管道和无名管道的使用方法基本相同。

(2)信号(Signal)

信号是比较复杂的通信方式，用于通知接受进程有某种事件发生。涉及的系统调用描述如下。

kill()

进程用 kill( )向一个进程或一组进程发送一个信号。系统调用格式为 int kill(pid，sig)。其中，pid 是一个或一组进程的标识符，sig 是要发送的软中断信号。信号的发送分如下三种情况。

① pid>0 时，核心将信号发送给进程 pid。

② pid=0 时，核心将信号发送给与发送进程同组的所有进程。

③ pid=-1 时，核心将信号发送给所有用户标识符真正等于发送进程的有效用户标识号的进程。

signal()

接收信号的程序用 signal()来实现对处理方式的预置，允许调用进程控制软中断信号。系统调用格式为 signal(sig, function)，此时需包含头文件 signal.h。其中 sig 用于指定信号的类型，sig 为 0 则表示没有收到任何信号。

function 是该进程中的一个函数地址，在核心返回用户态时，它以软中断信号的序号作为参数调用该函数，对除了信号 SIGKILL、SIGTRAP 和 SIGPWR 以外的信号，核心自动重新设置软中断信号处理程序的值为 SIG_DFL，进程不能捕获 SIGKILL 信号。

function 的解释如下。

1) function=1 时，进程对 sig 类信号不做任何处理便立即返回，亦即屏蔽该类信号。

2) function=0 时，缺省值，进程收到 sig 信号后终止自己。

3) function 为非 0、非 1 类整数时，执行用户设置的软中断处理程序。

(3)消息(Message)队列

消息队列也叫报文队列，是消息的链接表。消息队列用于运行于同一台机器上的进程间通信，和管道很相似，有足够权限的进程可以向消息队列中发送某种类型的消息，被赋予读权限的进程可以从消息队列中读取某种类型的消息。但消息主存可以根据需要自行定义，从而使消息队列克服了信号承载信息量少、管道只能承载无格式字节流以及缓冲区大小受限等缺点，在实际编程中应用较广。

(4)共享主存

共享主存是在系统内核分配的一块缓冲区，多个进程都可以访问该缓冲区。采用共享主存通信的一个显而易见的好处是效率高，因为进程可以直接读写主存，而不需要任何数据的拷贝，避免了在内核空间与用户空间的切换。对于像管道和消息队列等通信方式，则需要在内核和用户空间进行四次的数据拷贝，而共享主存则只拷贝两次数据：从输入文件到共享主存区和从共享主存区到输出文件。实际上，进程之间在共享主存时，并不总是读写少量数据后就解除映射，有新的通信时，再重新建立共享主存区域。而是保持共享区域，直到通信完毕为止，这样，数据内容一直保存在共享主存中，并没有写回文件。共享主存中的内容往往是在解除映射时才写回文件的。因此，采用共享主存的通信方式效率是非常高的。

在 Linux 系统中，共享主存是针对其他通信机制运行效率较低而设计的，多用于存储应用程序的配置信息。这个机制的不利方面是其同步和协议都不受程序员控制，必须确保将句柄传递给子进程和线程，因此往往与其它通信机制，如信号量结合使用，来达到进程间的同步及互斥。

(5)信号量(semaphore)

信号量及信号量上的操作是一种解决同步、互斥问题的较通用的方法，并在很多操作系统中得以实现，Linux 改进并实现了这种机制。信号量用来协调不同进程间的数据对象，最主要的应用就是共享主存方式的进程间通信。本质上，信号量是一个计数器，用来记录对某个资源(如共享主存)的存取情况。

维护信号量状态的是 Linux 内核操作系统而不是用户进程。信号量主要提供对进程间共享资源访问控制的手段，用来保护共享资源。除了用于访问控制外，还可用于进程间及同一进程不同线程间的进程同步。信号量有以下两种类型：

① 二值信号量。最简单的信号量形式，信号量的值只能取 0 或 1，类似于互斥锁，但两者关注的内容不同。信号量强调共享资源，只要共享资源可用，其他进程同样可以修改信号量的值；互斥锁更强调进程，占用资源的进程使用完资源后，必须由进程本身来解锁。

② 计算信号量。信号量的值可以取任意非负值(受内核本身的约束)。