补充：

API：Application Programming Interface，[应用程序](http://baike.baidu.com/view/330120.htm" \t "http://baike.baidu.com/subview/16068/_blank)编程接口。API基于系统，可以是对系统调用的封装。

----------------

系统调用：

系统调用是指操作系统内核对外提供的一系列接口函数。当外部函数调用系统调用时,会通过软中断的方式把地址空间由用户空间切换到内核空间,执行系统调用函数，从而来操作内核和驱动硬件等,系统调用函数执行后,地址空间由内核空间切换回到用户空间。

-------------

库函数(广义包括API)与系统调用的区别：

库函数是对系统调用(或API)的封装（不是所有的库函数都是），为的是解决一些公共的问题和提供统一的系统调用的接口。

他和系统调用的优缺点就是：系统调用速度是明显要快于库函数（并不一定全部是，但绝大部分是），但系统调用缺乏移植性。库函数速度要慢，但解决了移植问题。这些在开发过程中要根据自己的实际情况来决定使用哪一个。

常见库函数：标C库，VC++、MFC、VB和Qt自带的库函数。

=========================

一．开发环境

1.常见的C程序相关文件后缀

.h - 头文件

.c - 源文件

.i - 预处理文件

.s - 汇编文件

.o - 目标文件

.a - 静态库文件

.so - 共享库文件

---------------------

2.从源代码到可执行文件的构建过程：

a)预编译(编译预处理)：去注释、头文件扩展、宏替换

注解：在C程序的源代码中以#(sharp)开头的，都是预处理指令。

gcc -E hello.c -o hello.i -> hello.i

#include <> 是在系统指定的路径下去找

#include "" 先在当前路径下去找，没有再去系统指定路径去找

b)编译：将高级语言翻译成汇编语言，得到汇编文件

gcc -S hello.i -> hello.s

c)汇编：将汇编语言翻译成机器指令，得到目标文件

gcc -c hello.s -> hello.o

d)链接：将目标文件和运行时库文件进行绑定

gcc hello.o -o hello -> hello

e)一步到位：

gcc hello.c

gcc -v hello.c 显示所有的编译过程

---------------------

补充：链接本质就是代码拷贝(包括库文件和头文件加载)

(命令:nm /usr/lib/gcc/i686-linux-gnu/4.6/../../../i386-linux-gnu/crt1.o)

start是程序入口,main函数是C语言入口。

-------------------------

3.gcc的编译选项

(1)熟悉

-v 查看编译器的版本等信息

-std 表示指定执行的C标准(gcc -std=c99)

-Wall 尽可能产生警告信息

-Werror 将警告当作错误进行处理

-I 指定头文件的附加搜索路径（优先于其它方式）

-L路径 告诉连接器到指定的路径下找库文件

-l库名 告诉链接器指定库的名字

(2)了解

-g 产生调试信息,采用GDB进行单步调试

-O 表示进行具体的优化

-x 显示指定源代码的具体编程语言

-pedantic对非标准的扩展语法产生警告

-------------------

补充：gdb调试工具的使用

在编译的时候加上-g/-ggdb选项，意思是将调试信息加入到可执行文件里

1)gcc test.c tmath.c -g -> a.out

2)gdb a.out

3)gdb调试命令

list列出程序的代码

b 函数的名字（或者行号） 设置断点

r 运行程序

n 执行下一条指令

p 变量的名字 输出变量的内容

s 切入到函数的内部执行step by step(调用函数时立刻输入s)

q 退出gdb调试器

set args 可以设置参数

Show args 可以显示设置的参数

-------------------

补充：

1）echo命令

echo 字符串：在显示器上显示字符串

$变量名：取变量的值

echo $变量的名字：将变量的内容显示在屏幕上

echo $?：将上一个命令的执行结果显示到显示器上

2）man帮助

man 1 command 查看linux命令的帮助信息

man 2 system call 查看系统调用的帮助信息

man 3 标准C库函数名 查看标准C库中的函数的帮助信息

3）ldd 可执行文件 查看可执行文件依赖的库文件

4）nm命令的使用

nm 二进制文件 察看文件的导出符号 (例如: nm tmath.o)

T表示： 外部函数，可以在外部调用，有函数实现的代码

U表示： 没有函数实现的代码

-------------------

4.多文件结构的编程(头文件中写什么)

(1)头文件的卫士

#ifndef xxx\_H //采用头文件的文件名,小写换大写,点换下划线

#define xxx\_H

#endif

(2)包含其他头文件

(3)宏定义

(4)自定义类型声明(结构体，联合，枚举)

(5)起类型别名

(6)外部变量声明

extern int num;

(7)函数声明

4.常见的预定义宏

\_\_FILE\_\_ 获取所在的文件名 %s

\_\_BASE\_FILE\_\_ 获取正在编译的源文件名 %s

\_\_LINE\_\_ 获取所在的行号 %d

\_\_FUNCTION\_\_/\_\_func\_\_ 获取所在的函数名 %s

\_\_DATE\_\_ 获取日期信息 %s

\_\_TIME\_\_ 获取时间信息 %s

======================================

二、环境变量

1. 环境变量的概念

环境变量：存放和编程环境/系统环境相关信息的变量叫做环境变量。

--》env(1) 察看系统的环境变量表

--》echo $环境变量：将环境变量的值输出到显示器

一些具体的环境变量：

USER 当前用户的名字

PATH 指定命令(ls,cp,pwd等)的查找路径

(按路径顺序，先访问到的路径里的同名命令就执行，自己可以编写命令放到PATH以使用)

-----------------------------

2. 环境变量的格式

环境变量的格式：在等号的两边坚决不允许出现空格。

name=value

name是环境变量的名字

value是环境变量的值

----------------------

3. 环境变量的配置

(1) Windows Xp系统中的配置

我的电脑 => 右键，选择属性 => 高级 => 环境变量 => 系统变量 => 选中Path,点击编辑 => 在Path变量值的最后增加分号(;),再增加新的路径(千万不要改动之前的变量值) => 一路点击确定即可

(2) Linux系统下的配置

打开任意一个终端，执行以下命令：

export PATH=$PATH:.

----------------------

注解：$环境变量： 获取环境变量的值

$PATH:. 在$PATH环境变量的后边添加上当前路径。

PATH=$PATH:. 在$PATH环境变量的后边添加上当前路径，然后更新PATH的值。

: 路径的分隔符，类似于Windows系统中的分号

export 表示导入/修改的意思，就是将后边的变量导出为环境变量。

----------------------

如果需要设置环境变量为永久有效，则执行以下指令：

vi ~/.bashrc文件，在文件末尾增加代码：export PATH=$PATH:.

source ~/.bashrc(或者bash)表示让文件的配置立即生效

~代表当前用户的工作主目录

----------------------------------------

补充：

export PS1="\W\$ " (只显示当前目录的省略，参考《鸟哥私房菜》)

bash有外部命令和内部命令

跟bash独立的应用程序称为外部命令

命令如果是bash中的程序，称为内部命令

type command 用于察看命令的内部还是外部。

----------------------------------

4. 编程相关的环境变量

PATH - 存放不需要再增加路径就可以直接运行可执行文件的目录

CPATH - 主要表示C语言中头文件所在的路径

CPLUS\_INCLUDE\_PATH - 主要表示C++头文件所在的路径

LIBRARY\_PATH - 编译链接时查找静态库和共享库的路径

LD\_LIBRARY\_PATH - 运行时查找共享库的路径

5. 头文件的定位方式

(1)#include <>

表示去系统默认的路径中查找指定的头文件 /usr/include

(2)#include "<目录>/xxx.h"

缺点：头文件路径发生变化，必须修改源程序

(3)export CPATH=$CPATH:<目录>

缺点：同时构建多个工程时，容易引发冲突

1. gcc -I编译选项：gcc -I<目录>

优点：既不会修改源程序，也不会有任何冲突，推荐使用此方法

=======================================

1. 静态库和动态库

1. 静态库和动态库

什么是库？什么是静态库？

函数的仓库，就称为函数库。函数库分为静态和动态两种

静态库，静态库中的函数，在编译阶段链接。称为静态库。

动态库，动态库中的函数，在程序执行的时候链接。成为动态库。

2. 动态库和静态库的区别

1）使用静态库链接的可执行文件：文件比较大，不需要跳转，执行的时候就不再依赖于静态库文件。

2）使用动态库链接的可执行文件：文件比较小，需要跳转，执行的时候依赖于动态库文件。

————————

补充：

静态链接：发生在编译阶段(把函数代码完全拷贝到可执行文件中,变成T)

动态链接：程序运行的时候，才发生的链接(例如：printf)，使用时需要链接库文件

静态库和动态库的命名规则：

静态库：lib库名.a

动态库：lib库名.so

---------------------

3.静态库的创建：

a) 先编译源文件为目标文件

gcc -c tmath.c process.c

b) 将目标文件打包到静态库文件

ar -r libtmath.a tmath.o process.o

ar -t libtmath.a 察看静态库里包含的目标文件

c) 使用静态库编译链接文件，生成可执行文件

gcc test.o -L. -ltmath -o tt

-L路径 库文件所在路径

-l库名 找仓库的名字(不需要后缀)

=========================

4.动态库的制作：

a) 将源文件编译为目标文件

gcc -fPIC -c process.c tmath.c

-fPIC表示多个进程共享库文件时，只加载一份

b) 生成动态库

gcc -shared \*.o -o libtmath.so

c) 动态库的使用

gcc test.c -L. -ltmath

d) 链接时，失败

原因：加载器找不到库文件，与静态库的链接器不同

e) 解决办法：需要让加载器知道库文件的所在路径。

1)通过配置环境变量LD\_LIBRARY\_PATH告诉加载器的搜索路径

export LD\_LIBRARY\_PATH=$LD\_LIBRARY\_PATH:.

2)将库文件移动到/usr/lib或者/lib文件夹下

sudo mv libtmath.so /usr/lib

=============================

5.动态加载库函数

在程序中，根据自己的需求加载动态库，使用到了一系列的函数

dlopen(3)、dlerror(3)、dlsym(3)、dlclose(3)

注意：编译链接的时候记得使用： -ldl，表示链接指定库文件

===================================

四、C语言中的错误处理

1.错误编号和错误信息

(1)错误编号

errno: 系统设置的全局变量

(2)错误信息

a.strerror(3)

char \*strerror(int errnum);

功能：根据错误号返回错误描述信息

b.perror(3)

void perror(const char \*s);

功能： 打印系统错误信息

stdin 标准输入 键盘

stdout 标准输出 显示器

stderr 标准错误输出 显示器

======================================

2.环境表的概念和使用

(1) 环境表的概念

每个进程都有一张独立的环境变量表，来保存专属于该进程的所有环境变量。

环境变量表是以NULL结尾的字符指针数组，每个指针指向一个格式为"变量名=变量值"的环境变量字符串，该指针数组的首地址保存在全局指针(char\*\*)environ中，所以访问环境表则需要借助变量environ。

代码参见：env.c

(2) 主函数的原型

int main(int argc,char\* argv[],char\* envp[]){}

第一个参数：命令行参数的个数

第二个参数：命令行参数的地址信息

第三个参数：环境表的首地址

(3) 环境表相关的处理函数

getenv(3)、putenv(3)、setenv(3)、unsetenv(3)、clearenv(3)

====================================

五、内存管理

1. 程序和进程的概念

程序：计算机指令集合形成的文件。程序是静态的。

进程：程序运行的一个实例。进程是动态的。

2. 进程中的内存区域划分，从低地址到高地址依次为：

(1)代码段(text) - 存放可执行指令和功能代码的区域，比如：函数名

(2)只读常量区 - 存放字符串常量和const修饰的且已初始化的全局和静态局部变量

(3)数据段(data) - 存放没有常属性且已初始化的全局和静态局部变量

(4)BSS段(bss) - 存放没有常属性且未初始化的全局和静态局部变量，BSS段会在main函数执行之前自动清零

(5)堆区(heap) - 表示动态内存空间

(6)栈区(stack) - 存放非静态局部变量(包括具有常属性的)

(7)参数和环境区 - 命令行参数和环境变量

--------------

简单记忆：

内存地址从小到大分别是：

(代码区 只读常量区)(全局区 BSS段) 堆 栈

其中,堆区和栈区没有明确的分割线，可以适当地调整

-------------------------

3. 字符串存储形式之间的比较

对于一个存储常量字符串的字符指针和字符数组来说，字符指针可以改变指向，不可以改变指向的内容；而字符数组可以改变指向的内容，不可以改变指向;

对于一个存储常量字符串的动态内存空间来说，其中指向该内存空间的指针,既可以改变指向，又可以改变内容

-------------------------

补充：

1. 通过命令查看进程内存映射：

--》使用命令ps -aux找到进程号

--》cat /proc/进程号/maps：查看进程内存映射

2. 通过size命令可以查看特定可执行文件的代码段、数据区和BSS区的大小。

size a.out

===============================

4. 虚拟内存管理技术

Unix/Linux系统中的内存都是采用虚拟内存管理技术来进行管理的,即：每个进程都有0~4G的内存地址(虚拟的，并不是真实存在的),由操作系统负责把内存地址和真实的物理内存映射起来。因此，不同进程的内存地址看起来是一样的，但是所对应的物理内存是不一样的

其中0~4G的虚拟地址空间分为：用户空间和内核空间;用户空间指0~3G的虚拟地址空间,而内核空间指3G~4G的虚拟地址空间;用户程序运行在用户空间,内核空间只有系统内核才能访问,用户程序不能直接访问内核空间，不过系统内核提供了一些系统函数负责从用户空间切换到内核空间

内存地址的基本单位是字节,内存映射的基本单位是内存页,目前主流的操作系统中一个内存页是4Kb(4096字节)

1Kb = 1024byte(字节)

1byte = 8个bit(二进制位)

----------------------

1. 内存壁垒与段错误

(1)不同进程中，内核空间的映射是同一块物理内存，而用户空间的映射必须是不同物理内存

(2)一切对虚拟内存的越权访问，都将导致段错误

--》试图访问没有映射到物理内存的虚拟内存

例如：scanf函数缺少&，空指针/野指针的使用

int\* pi = NULL; //空指针

int\* pi; //野指针

scanf("%d",pi); //error，极可能引发段错误

--》试图以非法方式访问虚拟内存，如对只读内存做写操作等

============================

6. 标准内存分配函数

(1)使用malloc内存的注意事项

--》使用malloc申请动态内存时，会附加若干(一般12个)字节，存放控制信息(MCB，内存控制块，包括内存的大小、前后内存块的地址)

--》malloc函数底层采用链表的形式去处理多个内存块。

--》malloc分配的动态内存,千万不要越界访问,因为极有可能破坏MCB，从而引发段错误

(2)使用malloc申请内存的一般映射规则

--》使用malloc申请比较小的动态内存时,操作系统会一次性分配33个内存页,从而提高效率

--》使用malloc申请比较大的内存时,系统会默认分配34个内存页，当申请的内存超过34个内存页时，则系统会再次33个内存页(也就是按照33个内存页的整数倍进行分配)。

--》使用free释放动态内存时，释放多少则在动态内存总数中减去多少,当所有的内存释放完毕时，进程还保留33个内存页备用，直到进程结束,主要是为了提高效率。

------------------------------------------

7. 内存管理方式的比较

STL(标准模版库) - 自动申请/释放动态内存

new/delete - 申请/释放动态内存,C++语言中的运算符

malloc()/free() - 申请/释放动态内存,标C函数

sbrk()/brk()函数 - 申请/释放动态内存,UC函数

mmap()/munmap函数 - 建立/解除 到内存的映射

8. 映射的建立和解除(物理内存映射到进程中内存的堆区)

mmap(2)、munmap(2)

===============================

六、文件操作

1. 文件的基本概念：

（1）在Unix/linux系统中,几乎所有的一切都可以统称为文件

（2）对于文件的操作几乎适用于所有的设备等等

（3）目录也可以看作文件处理

-----------------------------

举例：文件重定向

/dev/null - 空设备(掌握)

echo 字符串 - 原样输出字符串内容

echo 字符串 > 文件名 - 写入字符串到文件(没有换行符)

echo 字符串 > /dev/null - 表示丢弃结果信息

cat /dev/null > 文件名 - 表示清空文件

/dev/tty - 输入输出设备,一般默认为终端(了解)

echo 字符串 > /dev/tty - 输出到输入输出设备

cat /dev/tty - 读取/打印输入输出设备内容

-----------------------------

2. 文件操作的相关函数

UC系统调用：open(2)、close(2)、read(2)、write(2)、lseek(2)

标C库函数：fopen(3)、fclose(3)、fwrite(3)、fread(3)、fseek(3)

===============================

七、文件描述符的拷贝

1. 文件描述符

文件描述符本质就是一个整数,可以代表一个打开的文件。但是文件的信息并不是保存在文件描述符中，而是存在文件表等结构中,使用open函数打开一个文件时,会把文件的信息放入文件表等结构中,但是出于安全和效率等因素的考虑，文件表等结构不适合直接操作，而是给文件表对应一个编号，使用编号进行操作，这个编号就是文件描述符

系统会管理文件描述符，在每个进程中都有一张描述符总表,当有新的文件描述符需求时,会去总表中查找未使用的最小值并且返回。文件描述符本质就是非负整数，也就是从0开始，一直到OPEN\_MAX(在linux系统中一般是255),其中0 1 2被系统占用,分别代表标准输入、标准输出以及标准错误

注意：

打开不同的文件时，对应的文件表和v节点表信息都不同,而多次打开相同的文件时,v节点表信息相同,文件表信息不同

close()函数的工作方式：先把文件描述符与文件表的对应关系解除,不一定会删除文件表,只有当文件表没有与其他描述符对应时才会删除文件表(一个文件表可以对应多个描述符),close()函数也不会修改描述符的整数值,但是会让一个描述符无法代表一个文件

-------------------------------

2. 拷贝文件描述符的函数

dup(2)、dup2(2)

注意：文件描述符的复制本质就是让多个文件描述符对应同一个文件表，也就是对应同一个文件，多一个途径操作文件

================================

八、文件锁

fcntl(2)(重点)

#include <unistd.h>

#include <fcntl.h>

int fcntl(int fd, int cmd, ... /\* arg \*/ );

功能：根据文件描述符对文件执行的操作

(1)复制文件描述符(了解)

(2)设置/获取文件描述符的标志(了解)

(3)设置/获取文件状态的标志(了解)

(4)实现文件锁的功能(掌握)

2. 使用fcntl函数实现文件锁的功能

当使用多个进程同时读写文件时，可能会引发文件的混乱，如果所有的进程都是读文件，则可以同时进行；但是只要有一个进程执行写操作，则多个进程应该串行操作而不是并行,使用文件锁实现上面的想法,文件锁就是读写锁：一把读锁和一把写锁,其中读锁是一把共享锁，允许其他进程加读锁,不允许加写锁;写锁是一把互斥锁，不允许其他进程加读锁和写锁

-----------------------

总结：

1.如果加了一把锁，只能导致第二个加锁失败(两个读锁除外)，文件依然可以修改

2.写锁与任何别的(读或写)锁互斥，上读锁之后，也不能上写锁

3.我们讲的是协同锁/君子协定，但是有强制锁

4.约定：只读，上读锁；读写时，上写锁。

===============================

九、文件元数据

在一个文件里，有文件的内容和文件的属性。文件的元数据，又叫文件的属性，包括大小，权限，创建时间等。

1. stat/fstat函数获取文件元数据

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

int fstat(int fd, struct stat \*buf);

功能：获取指定文件的元数据

----------------------------------------

2. 一些获取或修改文件元数据的函数

access(2) - 判断文件是否存在以及对应的权限

chmod(2) - 修改文件权限

truncate(2) - 修改文件大小，如果文件变小则多余数据丢弃，如果文件变大则扩展出来的区域用'\0'填充

umask(2) - 设置创建文件时需要屏蔽的权限，返回之前屏蔽的权限

----------------------------------------------------------------

扩展：2个查看文件元数据的命令（ls -l、stat）

ls -l a.txt结果如下:

- rwx rwx r-x 1 tarena tarena 456 5月4 14:49 a.txt

文件类型 属主 属组 其它 硬链接数 属主名称 属组名称 大小 最后修改时间 文件名

-------------------------------------------------------------

1. 硬链接和软链接
2. 同一个文件的硬链接指向同一个inode，inode为文件存储时的索引节点号
3. 硬链接数：对应文件存储的接口数，硬链接数为0的时候文件可能被系统删除。
4. 软链接中有自己的inode，存放了某个硬链接文件名(以便于找到inode)，相当于一个快捷方式

(4)建立硬链接和软链接的命令

ln hello a 为h文件建立硬链接文件a

ln -s hello b 为h文件建立软链接文件b

(5)文件硬链接和软链接的处理函数

link(2) - 创建硬连接

unlink(2) - 删除硬连接

symlink(2) - 创建软链接

rename(2) - 修改硬链接(名字或路径)

remove(3) - 删除硬链接(包括普通文件和空目录)

========================================

十、目录管理

文件夹的内容是什么？

文件夹的内容就是文件夹中的文件或文件夹。

文件夹权限：-rwx：x表示进入，w表示创建或修改文件或文件夹

1. 常用的目录管理函数

opendir(3)、readdir(3)、closedir(3)

-----------------------------------

2. 其他目录管理函数

mkdir(2) - 创建一个目录

rmdir(2) - 删除一个目录

===============================

十一、进程的管理

1. 程序和进程的概念

程序：计算机指令集合形成的文件。程序是静态的。

进程：程序运行的一个实例。进程是动态的。

1. 查看进程信息的基本命令：

ps： 表示查看当前终端的进程信息(进程的快照)

ps命令的执行结果如下：

PID - 进程号(重点)

TTY - 终端的编号

TIME - 消耗CPU的时间

CMD - 进程启动的命令名称及参数选项(重点)

------------------------

ps -aux 表示显示所有包含其他使用者的进程

a - 显示所有用户拥有控制终端的进程信息

u - 也包括没有控制终端的进程

x - 以详尽方式显示

w - 以更大列宽显示

ps -efl 表示以全格式的方式显示当前所有的进程

-----------------------------------

3. 进程的相关ID

(1)目前主流的操作系统都支持多进程，如果进程A启动了进程B，那么进程A叫做进程B的父进程，而进程B就叫做进程A的子进程

(2)进程0是系统内部的进程，负责启动进程1(init进程)，也会启动进程2(页守护进程)，而其他所有的进程都是进程1/进程2直接/间接地启动起来

1. PID - 进程号，进程的身份证号称为进程号
2. PPID - 父进程号
3. 进程控制块(PCB)：存放进程的管理和控制信息的[数据结构](http://lib.csdn.net/base/datastructure" \o "算法与数据结构知识库" \t "http://blog.csdn.net/wyzxg/article/details/_blank)称为进程控制块。它是进程管理和控制的最重要的数据结构，每一个进程均有一个PCB，在创建进程时，建立PCB，伴随进程运行的全过程，直到进程撤消而撤消。

(6)获取相关ID的函数：

(int)pid\_t getpid() - 获取进程号

(int)pid\_t getppid() - 获取当前进程的父进程号

--------------------------

4. 进程的创建

fork(2)

功能：以复制正在调用进程的方式去创建一个新的进程，fork函数会复制父进程PCB。

返回值：父进程中返回子进程的pid，子进程中返回自己pid(0)，失败返回-1

(1)fork创建子进程的代码执行方式:

A. fork之前的代码，父进程执行一次

B. fork之后的代码，父子进程各自执行一次

C. fork函数的返回值，父子进程各自返回一次

\*\*\*\*\*\*\*\*\*

1. 父子进程的关系

a.父进程创建了子进程之后，父子进程各自独立运行，并没有明确的先后执行顺序

b.如果子进程先结束，则父进程负责帮助回收子进程的资源

c.如果父进程先结束，子进程会变成孤儿进程，并被1(init)进程收养，1进程也叫做孤儿院

d.如果子进程先结束，但是父进程由于各种原因没有回收子进程的资源，子进程变成僵尸进程(但是父进程结束后还是会回收)。

(3)父子进程中内存资源的关系(共享代码区)

a.使用fork创建出来的子进程会复制父进程中除了代码区之外的其他内存区域，代码区和父进程共享

b.由于PCB表的浅拷贝，文件表父子进程共享

\*\*\*\*\*\*\*\*\*

------------------------------

5. 进程的终止

\_exit(2)/\_Exit(2)

功能：立即终止正在调用的进程

exit(3)

功能：正常终止进程，该函数在终止进程期间会调用atexit()/on\_exit()函数注册过的函数

atexit(3)、on\_exit(3)

功能：注册一个函数，在进程退出的时候调用他

----------------------------------

6. 进程的等待

(1)wait(2)

功能：等待并回收任意僵尸子进程

两个预定义宏：

WIFEXITED(\*status) 判断子进程是否正常终止

WEXITSTATUS(\*status) 获取子进程退出码，即exit(100)中的100

(2)waitpid(2)

功能：等待并回收指定子进程或任意子进程资源

注意：wait(&status)等同于waitpid(-1, &status, 0);

-------------------------------------

7. 进程管理的其他处理函数

(1)vfork(2)

功能：功能与fork()函数基本一样，但这个函数几乎不用了

与fork区别：

1. 不拷贝父进程PCB，而是直接占用父进程的内存区域
2. 子进程先执行，父进程进入阻塞状态，直到子进程终止，或者调用exec系列函数跳出为止，对于子进程的终止建议使用\_exit()函数

(3)采用的是写时复制技术，只有在改变共享内容的时候才复制PCB，但是在该技术被应用到fork函数后，vfork函数基本没人用了

-------------------------

1. exec系列函数

功能：加载一个可执行文件，如果该函数不出错，该进程剩下代码都不执行，相当于结束

例子：ps -o pid,ppid,comm

execl("/bin/ps","ps","-o","pid,ppid,comm",NULL);

--------------------------------

补充：

(1)变长参数表，参数可以给任意多个，结束给参数-1/NULL表示终止，实现省略。

(2)exec系列函数，使用新的映象替换旧的映象。进程的pid并没有更换，只是PCB中的内容更换了。

--------------------------------------------

(3)system(3)

功能：执行指定的shell命令或shell脚本(底层就是vfork加上exec系列函数搭配使用)

===============================

十二、信号

1.中断的概念

中断就是指暂时停止当前程序的执行转而去执行新的程序，或者处理出现的意外情况

中断分为：硬件中断(硬件发生故障)和软件中断(软件模拟的硬件中断)

2.信号的概念和分类

(1)基本概念

信号本质就是软件中断，大部分信号总是中断一个进程的正常运行。

(2)特点

a.信号是异步的，进程并不知道信号何时会到来

b.进程既可以发送信号，也可以处理信号

c.每个信号都有一个名字，这些名字以SIG开头

(3)信号的分类

使用命令：kill -l 查看当前操作系统所支持的信号

掌握几个常见的信号：

信号2 SIGINT 使用ctrl+c产生此信号或kill -2 进程号

信号3 SIGQUIT 使用ctrl+\产生此信号或kill -3 进程号

信号9 SIGKILL 该信号不能被用户捕获只能由系统默认处理

信号19 SIGSTOP 该信号不能被用户捕获只能由系统默认处理(放入后台)

注意：

1. 在linux系统中信号是从1~64排序，不保证连续，其中1~31之间的信号叫做不可靠信号，34~64之间的信号叫做可靠信号;
2. 不可靠信号又叫做非实时信号，不支持排队，屏蔽时发送多次只会显示一次；
3. 可靠信号又叫做实时信号，支持排队，即每发送一次都会被显示。

-------------------------------

3. 信号的3种处理方式

(1)默认处理,绝大多数信号的处理方式都是终止进程

(2)忽略处理

(3)自定义处理

============================

4. 信号的捕获函数

signal(2)

功能：设置指定信号的指定处理方式

(signal函数内部把第一个参数signum，传递给函数指针handler做为参数)

注意：

1. 信号是异步的，当捕获指定信号并自定义处理访问栈以外的变量时，可能会出错。
2. 所以，信号的处理函数必须是可重入函数。
3. 可重入函数：函数中所有变量都在栈里。

-------------------------------

5. 发送信号的方式

(1)键盘发送信号(只能发送一些比较特殊的信号)

ctrl+c => 信号2 SIGINT

ctrl+\ => 信号3 SIGQUIT

ctrl+z => 信号20 SIGTSTP

(2)程序出错（只能发送一些比较特殊的信号）

段错误 => 信号11 SIGSEGV

总线错误 => 信号7 SIGBUS

...

(3)采用kill命令发送信号(所有信号都能发)

kill -信号值 进程号

(4)采用系统函数发送信号

raise()/kill()/alarm()/sigqueue()

-------------------------------------

6. 发送信号的系统函数

raise(3)

功能：给自己的进程/线程发送指定的信号

kill(2)

功能：主要用于给指定的进程发送指定的信号

alarm(2)

功能：指定的秒数之后发送SIGALRM信号

pause(2)

功能：等待一个(未被忽略的)有效信号

sleep(3)

功能：让正在运行的线程进入睡眠状态，直到指定的秒数,或者被一个不能忽略的信号打断。

-------------------------------------------------------

扩充：

sigaction(2) => signal函数的增强版

注解：一般和sigqueue(3)联用，可以获取信号附加的数据

sigqueue(3)

功能：向指定的进程发送指定的信号和附加数据

扩充：

子进程对信号的处理方式和信号掩码的继承：

1.对于fork/vfork函数创建的父子进程来说：

(1)子进程可以继承父进程对信号的处理方式

(2)信号集的屏蔽可以被子进程继承

2.使用exec函数创建的新进程：

1. 子进程继承父进程的自定义处理方式，会被还原为默认处理

原因：位于原进程地址空间中的信号处理函数，此时已经被新进程的地址空间取代了

子进程结束的时候，会给父进程发送SIGCHLD(14)信号。

===============================

十三、信号集与信号阻塞

1. 信号集

信号集 - 信号的集合，用于存储多个信号

(1)底层存储方式类似：0101 0011 => 存在信号1 2 5 7

(2)信号集的数据类型：sigset\_t信号集类型名

(3)处理信号集的几个函数：

sigemptyset(3)、sigfillset(3)、sigaddset(3)、sigdelset(3)、sigismember(3)

--------------------------------

2. 信号阻塞

（1）在某些特殊情况下，程序的执行不允许被信号打断，此时需要将一些信号阻塞

（2）信号未决，是指信号产生而被阻塞的情况，阻塞期间来过的信号叫未决信号

（3）两个函数

sigprocmask(2)

功能：检查和改变信号掩码

sigpending(2)

功能：获取信号阻塞时的未决信号

---------------------------------

总结：信号从产生到捕获的全过程：信号产生、信号阻塞、信号递达、信号捕获和处理(默认,忽略,自定义)

===============================

十四、进程间的通信

1. 概念：两个/多个进程之间的数据交换叫做进程间的通信

1. 进程间的通信方式

(1)文件、(2)信号、(3)管道、(4)环境变量

(5)共享内存、(6)消息队列(重点)、(7)信号量集

(8)套接字(重点) ...

---》其中(5)(6)(7)统称为XSI IPC通信方式(system V IPC - 系统IPC)

-------------------------------

3. 管道

（1）概念：本质上是以文件作为媒介来实现进程间的通信，该文件比较特殊，叫做管道文件

（2）分类：有名管道和无名管道

---》有名管道：使用命令(或函数)创建管道文件，实现任意两个进程之间的(双向)通信

---》无名管道：使用系统函数创建内核结构，用于有亲缘关系的进程间(单向)通信

----------------------------------

（3）创建有名管道的2种方法：

---》命令：mkfifo 管道文件名.pipe => 创建管道文件

---》函数：int mkfifo(const char \*pathname, mode\_t mode);

例子：

mkfifo a.pipe => 创建管道文件a.pipe

echo hello > a.pipe => 写入数据到管道文件中

cat a.pipe => 读取管道中的数据

注意：管道文件在磁盘上只有i节点没有数据块，也不保存数据

-------------------------------------

（4）无名管道(创建于内核中的结构)

---》函数：pipe(2)

注意：管道是比较古老的通信方式，现在很少使用(了解)

============================

---》以下为系统IPC(消息队列, 共享内存, 信号量集)

--------------------------

4. 消息队列(重点)

（1）概念

使用不同的进程将发送的数据打包成具体格式的消息，然后将消息放入到消息队列中，使用其他进程从消息队列中取出消息，从而实现进程间的通信

（2）使用消息队列通信的流程

---》获取key值(键值)，使用ftok函数

---》创建/获取消息队列，使用msgget函数

---》发送消息/接受消息，使用msgsnd和msgrcv函数

---》如果不再使用消息队列，使用msgctl函数删除消息队列

（3）相关的命令ipcs, ipcrm

ipcs 查看系统中所有的IPC结构

ipcs -q 表示查看当前系统中存在的消息队列

ipcrm -q ID 表示删除指定的消息队列

----------------------

5. 共享内存

（1）基本概念

本质上是由内核维护的一块内存区域，共享在两个/多个进程之间，然后两个/多个进程分别对该内存区域进行读写操作，从而实现通信,是最快的IPC通信方式

（2）使用共享内存通信的基本流程

---》获取key值，使用ftok函数

---》创建/获取共享内存，从而得到ID，使用shmget函数

---》挂接共享内存，使用shmat函数

---》使用共享内存，进行读写操作等

---》脱接共享内存，使用shmdt函数

---》如果不再使用，使用shmctl函数删除共享内存

（3）相关的基本命令

ipcs -m 查看系统中已存在的共享内存

ipcrm -m ID 删除指定的共享内存

=============================

6. 信号量集

（1）信号量和信号量集的概念

---》信号量：本质就是一个计数器，用于控制同时访问共享资源的进程数/线程数，解决有限资源的分配问题

---》信号量集：表示信号量的集合，也就是多个计数器的集合。

信号量集 <==> 信号量的集合(相当于信号量的数组)

（2）信号量的工作方式

---》先把信号量进行初始化，初始化为最大值

---》如果有进程申请到共享资源，那么信号量的值减1

---》当信号量的值为0时,终止进程对共享资源的申请，让申请共享资源的进程进入阻塞状态

---》如果有进程释放资源,那么信号量的值加1

---》当信号量的值大于0时,阻塞的进程可以继续抢占资源，抢占不到资源的进程继续进入阻塞状态等待

（3）使用信号量集通信的流程

---》获取key值,使用ftok函数

---》创建/获取信号量集，使用semget函数

---》初始化信号量集，使用semctl函数

---》操作信号量集,使用semop函数

---》如果不再使用则删除，使用semctl函数

（4）基本的命令

ipcs -s 查看系统中存在的信号量集

ipcrm -s ID 删除指定的信号量集

==========================

===============================