1. 系统编程

嵌入式linux操作系统的c语言编程，主要学习的是如何利用linux提供的API来进行linux上的应用程序设计。这部分的内容学好了仅仅凭此部分的内容都可以找到一份好的工作（可以在人才网站上输入**“linux c”进行职位搜索）**；主要分为以下几部分：

1. 文件编程；
2. 进程控制；
3. 进程通信
4. 多线程编程
5. 网络编程
6. 数据库编程（可选）

系统编程所学的内容归结到一点：

给出一个函数的原型(返回值、函数、形参类型)，能够正确的使用它。

1. API

Application program interface:应用程序编程接口——函数原型

应用程序（面向最终使用者）：一切应用程序程序最终都必须通过API来获得内核的服务，ANDROID也不例外

| 特殊的应用程序：系统命令——mkdir ls rm tar等

| |

API(函数原型)

系统调用（OS内核功能调用）

|

|

OS(操作系统)内核（进程管理、文件管理、设备管理、内存管理）

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

硬件（CPU、内存、IO接口等）

1. linux下的文件编程（底层IO和标准IO）

文件：在计算机系统中的一般用文件的方式来实现信息的永久存储。我们这里讲的文件编程是指普通文件的读写编程不是指设备文件、数据库文件。

根据我们在c语言中所学的文件操作，我们可以总结出对文件编程的步骤：

打开——>读/写/移动读写指针——>关闭（务必养成文件不再使用时把它关闭的习惯）

在linux中，一个进程会由系统给它自动打开三个文件：

标准输入(stdin) 标准输出（stdout） 标准出错(stderr)

1. linux提供的对文件操作的API(底层IO)
2. 打开文件

函数原型：

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

int open(const char \*pathname, int flags);

int open(const char \*pathname, int flags, mode\_t mode);

open是一个重载的方法。

参数说明：

pathname:表示要打开文件名（可以包含路径，如果没有则为当前目录）；

flags:表示打开文件的请求权限（O\_RDONLY O\_WRONLY O\_RDWR）以及一些其他的对文件的操，常用的如下：

O\_CREAT:打开的文件不存在则创建;

O\_TRUNC:打开的文件截短；

O\_APPEND:以追加的方式打开文件；

O\_EXECL:如果要创建的文件存在的则报错；

mode: 当flags包含O\_CREAT时，指明创建文件的权限（拥有者、同组用户、其他用户的权限），一般我们用8进制指定：

如: 0640-->(110 100 000)2,这里的三组二进制分别表示拥有者、同组用户、其他用户的权限，110表示拥有者可以对文件读、写，但不能执行

**这里要注意，一般当第二个参数出现O\_CREAT时，open才会出现第3个参数;flags的各种值可以通过按位或（|）运算符进行组合。**

返回值：当成功打开一个文件时，返回一个非负的整数——文件描述符，文件描述符和c标准IO中的成功打开一个文件返回文件流指针一样都是标识打开文件的；

如果打开失败，则返回-1；

在调用linux api失败时，我们可以通过perror函数来返回失败的原因：

#include <stdio.h>

void perror(const char \*s);

课堂小任务1：

利用不断重复打开同一已经存在的文件来判定linux下一个进程最多可以打开的文件数量。（利用open来做！！）

存在的问题：部分学员分不清函数的声明、函数的定义、函数的调用

参考代码：

#include <stdio.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

int main()

{

//int open(const char \*pathname, int flags);

int fd;

while(1)

{

fd=open("./gec.txt",O\_RDONLY);

if(fd<0)

{

perror("Failed to open:");

break;

}

printf("fd=%d\n",fd);

}

return 0;

}

课堂小任务2：

利用open函数创建一个文件tmp.txt，拥有者具有可读写权限，同组、其他用户没有权限；运行程序并通过ls -l命令查看文件是否创建成功。

参考代码：

#include <stdio.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

int main()

{

//int open(const char \*pathname, int flags);

int fd;

fd=open("./tmp.txt",O\_CREAT|O\_RDWR,0600);

if(fd<0)

{

perror("Failed to open:");

}

return 0;

}

1. 关闭文件

注意：当打开的文件不再操作时候最好能把它关闭，这样可以释放文件打开是系统分配的相关资源；如果不关闭文件，进程正常结束也会自动关闭。

#include <unistd.h>

int close(int fd);

参数说明：

fd: 要关闭的文件的文件描述符；

返回值：

成功关闭就返回0；出错就返回-1；

1. 读文件：向文件输出信息

#include <unistd.h>

ssize\_t read(int fd, void \*buf, size\_t count);

功能描述：从fd描述符所表示的文件读取count个字节保存到buf所开始的内存处；

参数说明：

fd: 要读的文件描述符；

buf: 读的字节要保存的内存起始地址；

count: 请求读取的字节数；

返回值：

小于0：出错；

大于0：成功读取的字节数，不一定和请求的count相等；

==0：文件结束

课堂小任务3：

利用read编程把文本文件中的内容读出并显示出来，运行并对结果进行分析。

参考代码：

#include <stdio.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <unistd.h>

int main()

{

int fd;

int count;

char buf[256];

fd=open("./task1.c",O\_RDONLY);

if(fd<0)

{

perror("Failed to open :");

return -1;

}

//ssize\_t read(int fd, void \*buf, size\_t count);

while((count=read(fd,buf,10))>0)

{

buf[count]=0;//buf[count]='\0'

printf("%s",buf);

}

close(fd);

return 0;

}

1. 移动文件读写指针函数

#include <sys/types.h>

#include <unistd.h>

off\_t lseek(int fd, off\_t offset, int whence);

功能描述：

让打开文件的读写指针移动到指定位置以实现文件随机读写：想从哪个字节开始读就从哪个自己开始读，默认情况随着对文件的读写其内部的读写指针会自动向后移动。

参数说明：

fd： 要操作的文件描述符；

offset: 读写指针需移动的距离（以字节为单位）,正数则向文件尾部移动，负数向文件头部移动；

whence:

SEEK\_END: 相对于文件尾部移动；

SEEK\_SET: 相对于文件头部移动；

SEEK\_CUR: 相对于当前位置移动；

返回值：

成功时返回移动后读写指针的位置；

失败时值为小于0；

举例：

将文件描述符为fd的文件的最后2个字符读出来

lseek(fd,-2,SEEK\_END);

read(fd,buf,2);

课堂小任务4：

将某一个文本文件的字符倒序输出。

提示：

先通过length=lseek(fd,0,SEEK\_END)来获取文件的长度；再通过lseek(fd，i,SEEK\_END)来从尾到头逐个移动指针，再读一个字节显示一个字节的方式来完成全部的倒序输出。

参考代码：

#include <stdio.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <unistd.h>

int main()

{

int fd;

int length;

int i;

char buf;

fd=open("./task1.c",O\_RDONLY);

if(fd<0)

{

perror("Failed to open:");

return -1;

}

length=lseek(fd,0,SEEK\_END);

printf("The lenght of task1.c is: %d\n",length);

for(i=-1;i>=-length;i--)

{

lseek(fd,i,SEEK\_END);

read(fd,&buf,1);

printf("%c",buf);

}

printf("\n");

close(fd);

return 0;

}

1. 文件写操作函数

#include <unistd.h>

ssize\_t write(int fd, const void \*buf, size\_t count);

功能描述：把从内存buf开始的count个字节写入到文件描述符fd所表示的文件中；

参数说明：

fd: 要操作的文件；

buf : 被写道文件中的数据的内存起始地址；

count: 要写的字节数；

返回值：

<0:失败;

>0:成功写入的字节数

课程小任务5：

利用write函数把一个长度大于20字节的文本文件的第7到第10字节修改为“8888”。

课后作业：

1. 温习所讲函数的原型；
2. 本次课的5个小任务做到“乱熟于心”
3. 提前做本周作业：利用linux 底层文件IO和链表实现一个基于命令行方式的学生信息管理系统:学生信息的增删改查（要求arm版本和pc机版本均实现）。

S5pv210 GEC的使用

1. linux内核、文件系统的烧写参照文档；
2. 嵌入式linux的会话方式：
3. Serial

用直连的串口线把开发板的串口0和pc机的串口连接起来，在pc机上以serial协议的方式开超级终端，并设置为8位数据位、没有奇偶校验位、1位停止位、没有流控、波特率为115200，即可在超级终端上执行linux系统命令来和板子进行交互；

1. Telnet

用网线把开发接入到和你的pc机的同一个网络中。先以serial登录到板子上，执行如下命令(有些参数需要针对实际情况修改，这里仅仅这对培训课室的环境)：

ifconfig eth0 192.168.1.2+序号 netmask 255.255.255.0

在执行命令检测开发板是否和你的pc机连通了：

ping 你pc的ip地址

修改板子的启动脚本，使得板子运行时能够自动运行telnet服务:telned:

执行命名：

vi /etc/profile

在最后一行“source upiot.sh”前面插入下面两行（同时在source upiot.sh前面加个“#”）：

ifconfig eth0 192.168.1.2+序号 netmask 255.255.255.0

/usr/sbin/telnetd &

#source upiot.sh

保存，退出

重启开发板，在超级终端里选择telnet协议，其他参数设置好，连接，如果登录账户root就可以了！

如果你的桌面linux和板子的linux够相互ping通，则可以在桌面linux上执行命令：

telnet 板子的ip地址

来登录到板子上。

1. 如何生成210板上的c语言可执行文件并在其上运行
2. 安装交叉编译工具链

可以从网上很多的开源网站上下载到arm-linux的gcc交叉编译工具链，一般购买的arm开发板的配套资料里面也自带了gcc交叉工具链。在培训课室的教师机的“F:\20151108\TOOL\arm 工具”下的arm.tar.bz2就是一个交叉编译工具链。

先把arm.tar.bz2复制到linux当前用户工作目录下，再执行命令：

cd ~

mkdir crosstooltain

tar xvf arm.tar.bz2 -C crosstoolchain/

注意：**我们在做嵌入式开发时，最好能够保证编译bootloader linux内核 文件系统 应用的程序的为同一版本的工具链，否则可能会出现问题！！**

将解压缩工具链的bin目录添加到环境变量PAHT里面，执行如下命令：

cd crosstoolchain/arm/bin

echo export PATH=$PWD:\$PATH>>~/.bashrc

重新开启终端，执行如下命令：

which arm-linux-gcc

会发现此时的arm-linux-gcc使我们刚才自己安装的工具链了！！

用下面的命令可以查看工具链的bin目录是否添加到环境变量PATH中。

echo $PATH

1. 交叉编译工具链的使用

**arm-linux-gcc -o 输出可执行文件名 以空格分开的源文件列表**

如果你想确认你是否生成了arm-linux上的可执行文件，可以通过以下命令来查看：

**file 输出可执行文件名**

举例：

//armhello.c

#include <stdio.h>

int main( )

{

printf(“Hello,world!\n”);

return 0;

}

arm-linux-gcc -o armhello armhello.c

file armhello

1. 交叉编译生成的可执行文件的下载、运行

在桌面linux的终端下执行如下命令：

cd ~

mkdir armshare

sudo gedit /etc/exports

用gedit打开exports后在该文件添加如下行并保存以设定nfs的共享目录

/home/gec/armshare \*(rw,sync,no\_root\_squash)

执行命令重启nfs服务器：sudo service nfs-kernel-server restart

在板子的终端上执行如下命令：

mount -t nfs 你的linux的IP地址:/home/gec/armshare /mnt -o nolock

到此，板子上的linux就可以远程访问桌面linux的/home/gec/armshare了！！

把前面生成的armhello复制到/home/gec/armshare中去，然后再板子的Linux上想对armhello做任何事情了！！

用tftp的方式下载文件到板子的文件系统中：

1. 运行tftp服务器并设定共享目录为桌面；
2. 把需要下载的文件复制到tftp服务器的共享目录中；
3. 在板子的终端上执行如下命令：

tftp -g -r 下载文件名 tftp服务器ip

如果是可执行文件，还需要在板子的终端上执行命令更改下载文件的权限：

chmod 777 下载文件名

工欲善其事必先利其器——Linux下进行c c++ qt开发的集成开发环境qtcreator的使用

交叉开发环境的搭建：

1. 把共享目录下的armqt5.5-gec.ok.tar.bz2复制到你的机器上的ｌｉｎｕｘ上面的当前用户家目录下：／ｈｏｍｅ／ｇｅｃ；
2. 在终端下执行如下命令：

ｃｄ　　~

sudo tar xvf armqt5.5-gec.ok.tar.bz2 -C /opt

Linux下对目录这个特殊文件的编程

1. opendir

原型：

#include <sys/types.h>

#include <dirent.h>

DIR \*opendir(const char \*name);

参数说明：

name：执行要打开目录的字符串；

返回值：

成功：返回指向目录流的一个指针；

失败：返回一个空指针

1. readdir

原型：

#include <dirent.h>

struct dirent \*readdir(DIR \*dirp);

参数：

dirp： 指向一个成功打开的目录流指针；

返回值：

每成功调用一次都会返回目录流中一个目录项——struct dirent的指针，如果目录流中的全部目录项已经读取完毕则返回一个NULL指针；

struct dirent {

ino\_t d\_ino; /\* inode number \*/

off\_t d\_off; /\* offset to the next dirent \*/

unsigned short d\_reclen; /\* length of this record \*/

unsigned char d\_type; /\* type of file; not supported

by all file system types \*/

char d\_name[256]; /\* filename \*/

};

Eg:

DIR \*pDir;

struct dirent \*pDirent;

pDir=opendir(“/”);

while((pDirent=readdir(pDir))!=NULL)

{

printf(“%s\n”,pDirent->d\_name);

}

1. closedir

#include <sys/types.h>

#include <dirent.h>

int closedir(DIR \*dirp);

当一个成功打开的目录流不再使用时调用该函数关闭目录流以释放资源。

课堂小任务：

利用对目录操作的相关函数实现linux系统命令ls的功能——myls

对目录进行操作的API还有：

rmdir mkdir

课堂小任务：

利用目录相关的api和普通文件相关的api，编写程序实现对目录进行复制的命令，格式如下（简单起见，先不考虑子文件夹的情况）：

cpdir 源目录名 目标目录名

1. 分解“动作”1：把源目录下的所有文件名（包含路径，除掉“.” “..”）打印出来。

参考代码：

#include <stdio.h>

#include <sys/types.h>

#include <fcntl.h>

#include <dirent.h>

#include <string.h>

int main(int argc,char \*\*argv)

{

char \*psourceDir,\*pdestDir;

DIR \*pdir;

struct dirent \*pdirent;

char sourcefile[128];

if(argc<3)

{

printf("Usage: mycopydir source\_dir dest\_dir");

return -1;

}

psourceDir=argv[1];

pdestDir=argv[2];

pdir=opendir(psourceDir);

if(pdir==NULL)

{

perror("Failed to open dir:");

return -1;

}

while((pdirent=readdir(pdir))!=NULL)

{

if((strcmp(pdirent->d\_name,".")==0)||

(strcmp(pdirent->d\_name,"..")==0))

continue;

strcpy(sourcefile,psourceDir);

strcat(sourcefile,"/");

strcat(sourcefile,pdirent->d\_name);

printf("%s\n",sourcefile);

}

closedir(pdir);

return 0;

}

1. 分解动作2：实现文件复制

#include <stdio.h>

#include <sys/types.h>

#include <fcntl.h>

#include <dirent.h>

#include <string.h>

#include <sys/stat.h>

#include <unistd.h>

void mycopyfile(const char \*pSource,const char \*pDest);

int main(int argc,char \*\*argv)

{

char \*psourceDir,\*pdestDir;

DIR \*pdir;

struct dirent \*pdirent;

char sourcefile[128];

int ret;

if(argc<3)

{

printf("Usage: mycopydir source\_dir dest\_dir");

return -1;

}

psourceDir=argv[1];

pdestDir=argv[2];

ret=mkdir(pdestDir,S\_IRWXU | S\_IRWXG | S\_IROTH | S\_IXOTH);

if(ret<0)

{

perror("Failed to mkdir:");

return -1;

}

pdir=opendir(psourceDir);

if(pdir==NULL)

{

perror("Failed to open dir:");

return -1;

}

while((pdirent=readdir(pdir))!=NULL)

{

char destbuf[128];

if((strcmp(pdirent->d\_name,".")==0)||

(strcmp(pdirent->d\_name,"..")==0))

continue;

strcpy(sourcefile,psourceDir);

strcat(sourcefile,"/");

strcat(sourcefile,pdirent->d\_name);

// printf("%s\n",sourcefile);

strcpy(destbuf,pdestDir);

strcat(destbuf,"/");

strcat(destbuf,pdirent->d\_name);

mycopyfile(sourcefile,destbuf);

}

closedir(pdir);

return 0;

}

void mycopyfile(const char \*pSource,const char \*pDest)

{

int srcfd;

int destfd;

int count;

char buf[512];

srcfd=open(pSource,O\_RDONLY);

if(srcfd<0)

{

printf("source file:%s\n",pSource);

perror("Failed to open source file:\n");

return;

}

destfd=open(pDest,O\_RDWR|O\_CREAT,0666);

if(destfd<0)

{

perror("Failed to open source file\n");

return;

}

while((count=read(srcfd,buf,512))>0)

{

//write(destfd,buf,512);//error!!!!!

write(destfd,buf,count);

}

close(srcfd);

close(destfd);

}

对文件属性的操作：

1. stat

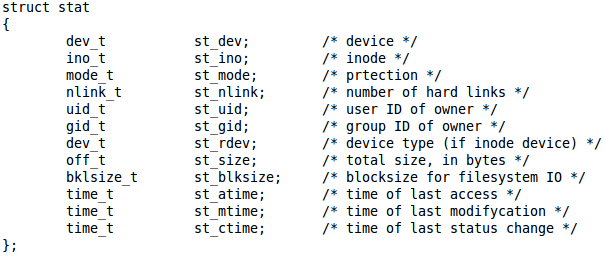
#include <sys/stat.h>

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

参数说明：

path: 表示需要获取属性的文件（含路径）；

buf: 成功获取的文件属性保存在buf所指向的struct stat结构体对象中。



其中，通过st\_mode成员判断该文件的类型（通过宏S\_ISREG S\_ISDIR S\_ISFIFO S\_ISCHAR等）；通过st\_size获取文件的长度；通过st\_atime st\_mtime st\_ctime获取文件相关的时间信息

在linux下关于时间操作的函数

1. 获取系统时间

#include <time.h>

time\_t time(time\_t \*tloc);

功能描述：该函数获取相对于1970年1月1日0时0分0秒的个时间

#include <time.h>

char \*asctime(const struct tm \*tm);

char \*asctime\_r(const struct tm \*tm, char \*buf);

char \*ctime(const time\_t \*timep);

char \*ctime\_r(const time\_t \*timep, char \*buf);

struct tm \*gmtime(const time\_t \*timep);

struct tm \*gmtime\_r(const time\_t \*timep, struct tm \*result);

struct tm \*localtime(const time\_t \*timep);

struct tm \*localtime\_r(const time\_t \*timep, struct tm \*result);

其中,struct tm结构体的定义如下：

struct tm {

int tm\_sec; /\* seconds \*/

int tm\_min; /\* minutes \*/

int tm\_hour; /\* hours \*/

int tm\_mday; /\* day of the month \*/

int tm\_mon; /\* month \*/

int tm\_year; /\* year \*/

int tm\_wday; /\* day of the week \*/

int tm\_yday; /\* day in the year \*/

int tm\_isdst; /\* daylight saving time \*/

};

在上面几个与时间相关的函数中，time localtime用得比较多；

明天内容：

1. linux下的串口编程；
2. Linux动态库、静态库的使用；
3. 阶段项目——数码相框的关键技术。
4. linux下的串口编程

我们这里说的串口是uart串口。在linux系统里面，“一切皆文件”，我们这里的uart串口在linux系统里面也是以文件的方式存在——串口设备文件，一般位于目录/dev/目录下。

对uart串口编程的基本步骤：

1. 打开；
2. **配置（普通文件一般无此步，不同的设备文件期配置方式也不同）**；
3. 读/写；
4. 关闭

Linux下串口的工作模式：

1. 规范模式：底层驱动会对传输过程中的一些控制符进行处理
2. **原始模式：**底层驱动会把串口所接收到的数据原封不动的上传的应用程序（在嵌入式开发中几乎均采用此中模式）

Linux下串口编程的模版：

#include <termios.h>

int fd;

struct termios newtermios;//struct termios结构体的实例用来描述串口的配置

//xxxxx表示你要使用的串口是设备文件名

//O\_NOCTTY表示此串口不会成为进程的控制终端，这样如果串口收到“ctrl+c”不会把进程结束掉

//O\_NDELAY表示此串口不关心DCD线的信号

fd=open(“/dev/xxxxx”,O\_RDWR|O\_NOCTTY|O\_NDELAY);

if(fd<0)

{

perror(“Failed to open uart:”);

return -1;

}

//设配置为原始工作模式

cfmakeraw(&newtermios);

//设置串口波特率，YYYYY可为B9600 B115200等

cfsetspeed(&newtermios,YYYYY);

//设置数据位

Newtermios.c\_cflag&=~CSIZE;

Newtermios.c\_cflag|=CS8;

//设置奇偶校验位:没有校验位

Newtermios.c\_cflag&=~PARENB;

//设置停止为：1位停止位

Newtermios.c\_cflag&=~CSTOPB;

//清除串口残留的数据:即将使用的串口的应用不需要串口收发缓冲区的残余数据

Tcflush（fd,TCIOFLUSH）；

//使串口配置生效

Tcsetattr(&newtermios,TCSANOW);

//将串口工作方式设为阻塞方式：当没有数据可以ｒｅａｄ是就阻塞，与ｓｃａｎｆ类似

Fcntl(fd,F\_SETFL,0);

//根据实际情况读写串口

Read/write;

//关闭串口

Close(fd);

课堂小任务：

在GEC210的linux上编写一个程序，每隔一秒钟向串口打印信息“Hello,young man!”

因为我们gec210板上唯一一个引出来的RS232接口被用来作用linux内核启动默认的linux控制终端：linux内核的输出信息会从这里输出，shell解释器（/bin/sh）也会从这里接收系统命令；但我们现在需要把这个串口作为普通的通信接口，为了避免干扰，需要重新设置linux的默认控制终端：

Fd=open(“/dev/s3c2410\_serial3”,O\_RDONLY);

If(fd>0)

{

Ioctl(fd,TIOCCONS);//把fd设为linux的默认控制中断

Close(fd);

}

Gec210的引出来的rs232串口在linux文件系统中的文件名（含路径）：/dev/s3c2410\_serial0

我们把gec210引出的串口作为通信接口使用，用telnet的方式来远程登录到板子的linux上

参考代码：

#include <stdio.h>

#include <unistd.h>

#include <fcntl.h>

#include <sys/ioctl.h>

#include <termios.h>

#include <string.h>

int main(void)

{

int fd;

struct termios newtermios;

char buf[]="Hello,young man!\r\n";

fd=open("/dev/s3c2410\_serial3",O\_RDWR);

if(fd<0)

{

perror("Failed to open /dev/s3c2410\_serial3");

return -1;

}

else

{

ioctl(fd,TIOCCONS);

close(fd);

}

fd=open("/dev/s3c2410\_serial0",O\_RDWR|O\_NOCTTY|O\_NDELAY);

if(fd<0)

{

perror("Failed to open /dev/s3c2410\_serial0");

return -1;

}

cfmakeraw(&newtermios);

cfsetspeed(&newtermios,B115200);

newtermios.c\_cflag&=~CSIZE;

newtermios.c\_cflag|=CS8;

newtermios.c\_cflag&=~PARENB;

newtermios.c\_cflag&=~CSTOPB;

tcflush(fd,TCIOFLUSH);

tcsetattr(fd,TCSANOW,&newtermios);

fcntl(fd,F\_SETFL,0);

while(1)

{

write(fd,buf,strlen(buf));

sleep(1);

}

return 0;

}

课堂小任务：

在gec210上编写一个应用程序：从串口0接收的数据发送到当前进程的终端（printf）

gcc工具链和make工程管理器的使用（补充）

一个源代码编程最终的可执行文件需要经历一下4步：

源文件（用纯文本的编辑器编辑）——>(预处理：gcc -E，处理掉文件中以#开头的预处理指令，预处理后的文件依然是文本文件 )——>(编译：gcc -S，把源文件编程对应平台的汇编语言文件)

——>(汇编：gcc -c,把汇编语言文件编程目标平台的二进制文件)——>(gcc,把多个而进制文件和外部的库链接组装成最后的一个可执行文件)

预处理例子：

//gccExp.c

#include <stdio.h>

int main()

{

printf("Hello,world!\n");

return 0;

}

执行命令：

gcc -E -o gccExp.i gccExp.c

用文本编辑器观察gccExp.i的内容

编译的例子：

执行命令：

gcc -S -o gccExp.s gccExp.i

打开gccExp.s观察生成汇编语言代码

汇编的例子：

gcc -c -o gccExp.o gccExp.s

连接的例子：

gcc -o gccExp gccExp.o

gcc对输入进行处理如果发现该文件某些步骤没有做，则gcc会自动调用对应的命令来处理，如：

gcc -o gccExp gccExp.c

gcc会自动对其预处理、编译、汇编、链接

make工程管理器：

假设某一个工程的代码共有100个源文件：1.c 2.c 3.c 。。。。。。100.c,你想如何把它们生成最终的可执行文件？

make工程管理器是以makefile为脚本来对整个工程进行处理的。

Makefile文件编写的规则：

1. 每行以#开头则为注释；
2. 有多条规则组成，每条规则的编写格式：

目标：依赖

[Tab键] 命令（描述依赖如何生成目标）

make的执行格式

**make [目标] [-C 目录] [-f makefile文件名]**

如果目标省略，则从makefile文件的第一目标开始处理；

如果[-f makefile文件名省略]，则makefile文件名默认为makefile或Makefile;

如果[-C 目录]省略，则在当前目录下查找makefile

关于makefile如下知识点请自行在网上查阅资料学习：

1. makefile变量：预定义变量、自动变量、自定以变量；
2. makefile隐式规则
3. makefile的模式规则

Linux下的库的创建和使用：

1. 库是包含大量函数的二进制代码，这些函数可以被外部调用，这样实现了在二进制层次的代码的复用；
2. 库的分类：

静态库：在windows系统中，扩展名.lib;在linux系统中为.a;静态库中被调用函数的代码会编译时包含在调用者中；

动态库：在windows系统中，扩展名.dll;在linux系统中为.so;动态库中被调用函数在调用者运行时才会和其建立连接

1. linux中静态库的创建

第一步：生成.o文件

gcc -c -o 目标文件名.o 源文件

ar crs lib库名.a 目标文件列表

举例：

//add.c

int add(int x,int y)

{

return x+y;

}

//sub.c

int sub(int x,int y)

{

return x-y;

}

执行命令：

gcc -c add.c

gcc -c sub.c

ar crs libmymath.a add.o sub.o

libmymath.a就是我们的静态库！！

我们可以自己写一个关于自己库函数的头文件，别人使用该文件只用包含头就可以了：

//mymath.h

#ifndef \_\_MYMATH\_\_

#define \_\_MYMATH\_\_

int add(int,int);

int sub(int,int);

#endif

静态库的使用：

gcc -o 输出可执行文件 输入源文件列表 -I 第三方头文件目录 -L 第三方库文件目录 -l库名

举例：

//main.c

#include <stdio.h>

#include <mymath.h>

int main()

{

int x,y;

scanf("%d%d",&x,&y);

printf("%d,%d\n",add(x,y),sub(x,y));

return 0;

}

假设mymath.h 和libmymath.a均与main.c都放在同一目录下：

gcc -o staticLibExe main.c -I./ -lmymath -L ./

1. linux中动态库的创建和使用

gcc -fpic -c add.c

gcc -fpic -c sub.c

gcc -fpic -shared -o libmymath.so add.o sub.o

或者一步到位：

gcc -fpic -shared -o l**ib**mymath.**so** add.c sub.c

动态库的使用与静态库一样，当动态库和静态库都存在是优先链接动态库！！

gcc -o sharelibExe main.c -lmymath -I./ -L./

这里，main.c 和libmymath一起生成最终的可执行文件sharelibExe!!

执行如下命令可以查看sharelibExe所依赖的动态库文件：

ldd sharelibExe

当程序运行依赖动态库运行时条件（任一种都可以）：

1. 把相应的动态库复制到/lib或者/usr/lib目录；
2. 把相应动态库的目录路径添加到环境变量LD\_LIBRARY\_PATH;
3. 把相应动态库的的目录路径添加到/etc/ld.so.conf文件中，并执行ldconfig
4. 在链接时通过加入参数：

-Wl,-rpath,库目录

课堂小任务：

将上面的add.c sub.c生成arm-linux版的动态库，并写一个测试程序在arm-linux上运行。

本周作业：

1. 继续完善上周的学生信息管理系统，重要模块能够生成动态库，这个系统的源码通过make工程管理器管理；
2. 在网上查阅linux下帧缓冲区设备文件（/dev/fb0）的读写方法

阶段项目：

嵌入式数码相框：能够对指定目录的jpeg图片按照一定的速度在GEC210的屏幕上播放。

1. 嵌入式linux下lcd屏显示的原理
2. 屏幕的分辨率

以像素为单位宽X以像素为单位的高,gec210的lcd屏的分辨率是800X480

(2) 色深

一个像素的颜色值由多少二进制位来表示。一般的色深为1 8 16 24 32

1. RGB值

RGB是比较常用的表示颜色的一种方法：任何一种颜色可以通过R(红色) G(绿色) B（蓝色）的分量值来表示，例如16为色的RGB的分量所站的位一般可以分为：5:5:6或5：6：5;24位色一般表示为：8：8：8

1. 显存

保存屏幕像素颜色值的一块内存区域，显卡的特殊电路会按照一定的速度把显存中的数据传送到屏幕上去。显存的大小=以像素为单位宽X以像素为单位的高X一个像素的颜色值所占的字节数，要注意的是尽管24位色一个像素的颜色值只需要3个字节，但考虑到内存的存取速度，在显存中一个像素的颜色值会占4个字节

1. 嵌入式linux下的显存——帧缓冲区framebuffer

framebuffer已经映射到了用户空间，以设备文件的形式存在，一般为/dev/fb0，如果你要读写framebuffer的内容只需要读写/dev/fb0就可以了

课堂小任务：

利用文件编程的知识把gec210的屏幕编程白色。

提示：gec210的屏幕为24位为色，但实际一个像素在framebuffer中占4个字节，其framebuffer的大小为：800X480\*4

参考代码：

#include <stdio.h>

#include <unistd.h>

#include <fcntl.h>

int main()

{

int fd;

int x,y;

fd=open("/dev/fb0",O\_RDWR);

unsigned int pixelcolor=0xff00ff00;

if(fd<0)

{

perror("Cannot open lcd:");

return -1;

}

for(y=0;y<480;y++)

{

for(x=0;x<800;x++)

write(fd,&pixelcolor,4);

}

close(fd);

return 0;

}

提高：

每隔2秒钟交替循环显示红、绿、蓝三种颜色，间隔2秒用可调用sleep(2)

我们除了可以利用read write方法来对一个文件进行读写外，还以调用mmap方法来像读写内存的方法来读写文件：

#include <sys/mman.h>

void \*mmap(void \*addr, size\_t length, int prot, int flags,

int fd, off\_t offset);

参数说明：

addr:一般为NULL,表示让系统去给要映射文件分配一个用户空间地址；

length：要映射的长度，一般是指要映射的整个文件的长度；

prot: PROT\_READ PROT\_WRITE PROT\_EXEC,可以组合；

flags:一般为MAP\_SHARED，表示该文件可以被多个进程同时映射；

fd：要映射的文件描述符；

offset:要映射的文件偏移量

返回值：

映射后的地址

int munmap(void \*addr, size\_t length);

取消映射。

参考代码：

#include <stdio.h>

#include <unistd.h>

#include <fcntl.h>

#include <sys/mman.h>

int main()

{

int fd;

int x,y;

unsigned int \*pFrameBuffer;

fd=open("/dev/fb0",O\_RDWR);

unsigned int pixelcolor=0xff0000ff;

if(fd<0)

{

perror("Cannot open lcd:");

return -1;

}

pFrameBuffer=(unsigned int \*)mmap(NULL,800\*480\*4,PROT\_READ|PROT\_WRITE,MAP\_SHARED,fd,0);

for(y=0;y<480;y++)

{

for(x=0;x<800;x++)

{

\*(pFrameBuffer+y\*800+x)=pixelcolor;

}

}

munmap(pFrameBuffer,800\*480\*4);

close(fd);

return 0;

}

课堂小任务：

采用mmap的方法实现对/dev/fb0的读写实现在gec210的lcd（50，50）处画一个宽为50x50的蓝色正方形

提示：可以先实现一个画线的函数void drawLine(int x0,int y0,int x1,int y1);

我们已经实现在lcd屏上的画点、画线，如果要把一副图片显示在lcd屏上，最关键的就是把图片上像素的rgb值逐个提取出来，再把该该像素的rgb值写到/dev/fb0。

1. jpeg库的移植
2. 把共享目录下的jpegsrc.v9a.tar.gz复制到你的linux的当前用户的家目录下；
3. 在你的linux下打开终端并进入到当前用户的家目录，然后执行如下命令：

tar xvf jpegsrc.v9a.tar.gz

cd jpeg-9a/

(3)对jpeg源码进行配置

./configure --host=arm-linux --prefix=/home/gec/3rdlib

其中，

--host表示最终生成的jpeg库的运行目标平台；

--prefix表示jpeg库的安装目录，

如果没有--host和--prefix则默认生成x86+linux平台上的jpeg库，当然你也可以生成以x86+linux上的jpeg库

1. 编译jpeg

make

1. 安装jpeg

make install

1. 我们可以观察安装后的成果

cd ~/3rdlib

tree

gec@ubuntu:~/3rdlib$ tree

.结果如下：

├── bin

│   ├── cjpeg

│   ├── djpeg

│   ├── jpegtran

│   ├── rdjpgcom

│   └── wrjpgcom

├── include

│   ├── jconfig.h

│   ├── jerror.h

│   ├── jmorecfg.h

│   └── jpeglib.h

├── lib

│   ├── libjpeg.a

│   ├── libjpeg.la

│   ├── libjpeg.so -> libjpeg.so.9.1.0

│   ├── libjpeg.so.9 -> libjpeg.so.9.1.0

│   └── libjpeg.so.9.1.0

└── share

└── man

└── man1

├── cjpeg.1

├── djpeg.1

├── jpegtran.1

├── rdjpgcom.1

└── wrjpgcom.1

其中include和lib我们利用jpeg库是会用到的两个重要的文件夹！

（7）找一副或多副JPEG格式的图片放到你的gec210的文件系统中

参考代码：

#include <stdio.h>

#include <jpeglib.h>

#include <stdlib.h>

#include <sys/mman.h>

#include <fcntl.h>

#include <unistd.h>

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

{

FILE \*fp;

int fd;

unsigned char \*pLineColor;

unsigned int \*pFrameBuffer;

int image\_width,image\_height;

//声明一个jpeg解压对象

struct jpeg\_decompress\_struct dinfo;

//把jpeg的出错信息关联到进程的标准错误设备上，除非你不关心jpeg库的出错信息

struct jpeg\_error\_mgr err;

dinfo.err=jpeg\_std\_error(&err);

//创建一个jpeg解压对象

jpeg\_create\_decompress(&dinfo);

if(argc<2)

{

printf("Usage:digitalPhoto jpeg\_filename.\n");

return -1;

}

fp=fopen(argv[1],"r");

if(fp==NULL)

{

perror("Cannot open jpeg\_file:");

goto exit;

}

//将jpeg解压对象和jpeg文件流关联起来，意味这jpeg解压对象将从文件流中获取数据

jpeg\_stdio\_src(&dinfo,fp);

//获取头部信息

jpeg\_read\_header(&dinfo,TRUE);

image\_height=dinfo.image\_height;

image\_width=dinfo.image\_width;

printf("w=%d,h=%d,num=%d\n",image\_width,image\_height,dinfo.num\_components);

//打开framebuffer

fd=open("/dev/fb0",O\_RDWR);

pFrameBuffer=(unsigned int \*)mmap(NULL,800\*480\*4,PROT\_READ|PROT\_WRITE,MAP\_SHARED,fd,0);

//开始解压

jpeg\_start\_decompress(&dinfo);

//申请一个保存jpeg一行颜色数据的缓冲区

pLineColor=(unsigned char\*)malloc(image\_width\*dinfo.num\_components);

//按行读取jpeg图片的数据并处理

while(dinfo.output\_scanline<dinfo.output\_height &&

dinfo.output\_scanline<480)

{

int col;

unsigned char \*pTmpColor=pLineColor;

unsigned char red,green,blue;

//读取一行

jpeg\_read\_scanlines(&dinfo,&pLineColor,1);

//处理一行

for(col=0;col<800&&col<dinfo.image\_width;col++)

{

red=\*pTmpColor;

green=\*(pTmpColor+1);

blue=\*(pTmpColor+2);

pTmpColor+=3;

\*(pFrameBuffer+(dinfo.output\_scanline-1)\*800+col)=(red<<16)|(green<<8)|blue;

}

}

jpeg\_finish\_decompress(&dinfo);

munmap(pFrameBuffer,800\*480\*4);

free(pLineColor);

fclose(fp);

close(fd);

exit:

jpeg\_destroy\_decompress(&dinfo);

return 0;

}

1. 参照以上的代码，结合以前的目录操作完成对指定目录的jpeg图片的自动播放。
2. 如何通过/dev/fb0来获取lcd屏的相关信息？

#include <linux/fb.h>

#include <sys/ioctl.h>

....

struct fb\_var\_screeninfo screeninfo;//声明一个与屏幕信息相关的结构体对象

int screenWidth,screenHeight;

int bytesperpixel;

fd=open("/dev/fb0",O\_RDWR);

ioctl(fd,FBIOGET\_VSCREENINFO,&screeninfo);

screenWidth=screeninfo.xres;//获取屏幕的宽

screenHeight=screeninfo.yres;//获取屏幕的高

bytesperpixel=screeninfo.bits\_per\_pixel/8;//获取每个像素颜色所占的字节数

修改版：

#include <stdio.h>

#include <jpeglib.h>

#include <stdlib.h>

#include <sys/mman.h>

#include <fcntl.h>

#include <unistd.h>

#include <linux/fb.h>

#include <sys/ioctl.h>

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

{

FILE \*fp;

int fd;

struct fb\_var\_screeninfo screeninfo;//声明一个与屏幕信息相关的结构体对象

int screenWidth,screenHeight;

int bytesperpixel;

unsigned char \*pLineColor;

unsigned int (\*pFrameBuffer)[800];

int image\_width,image\_height;

//声明一个jpeg解压对象

struct jpeg\_decompress\_struct dinfo;

//把jpeg的出错信息关联到进程的标准错误设备上，除非你不关心jpeg库的出错信息

struct jpeg\_error\_mgr err;

dinfo.err=jpeg\_std\_error(&err);

//创建一个jpeg解压对象

jpeg\_create\_decompress(&dinfo);

if(argc<2)

{

printf("Usage:digitalPhoto jpeg\_filename.\n");

return -1;

}

fp=fopen(argv[1],"r");

if(fp==NULL)

{

perror("Cannot open jpeg\_file:");

goto exit;

}

//将jpeg解压对象和jpeg文件流关联起来，意味这jpeg解压对象将从文件流中获取数据

jpeg\_stdio\_src(&dinfo,fp);

//获取头部信息

jpeg\_read\_header(&dinfo,TRUE);

image\_height=dinfo.image\_height;

image\_width=dinfo.image\_width;

printf("w=%d,h=%d,num=%d\n",image\_width,image\_height,dinfo.num\_components);

//打开framebuffer

fd=open("/dev/fb0",O\_RDWR);

ioctl(fd,FBIOGET\_VSCREENINFO,&screeninfo);

screenWidth=screeninfo.xres;//获取屏幕的宽

screenHeight=screeninfo.yres;//获取屏幕的高

bytesperpixel=screeninfo.bits\_per\_pixel/8;//获取每个像素颜色所占的字节数

pFrameBuffer=(unsigned int (\*)[800])mmap(NULL,800\*480\*4,PROT\_READ|PROT\_WRITE,MAP\_SHARED,fd,0);

//开始解压

jpeg\_start\_decompress(&dinfo);

//申请一个保存jpeg一行颜色数据的缓冲区

pLineColor=(unsigned char\*)malloc(image\_width\*dinfo.num\_components);

//按行读取jpeg图片的数据并处理

while(dinfo.output\_scanline<dinfo.output\_height &&

dinfo.output\_scanline<480)

{

int col;

unsigned char \*pTmpColor=pLineColor;

unsigned char red,green,blue;

//读取一行

jpeg\_read\_scanlines(&dinfo,&pLineColor,1);

//处理一行

for(col=0;col<800&&col<dinfo.image\_width;col++)

{

red=\*pTmpColor;

green=\*(pTmpColor+1);

blue=\*(pTmpColor+2);

pTmpColor+=3;

// \*(pFrameBuffer+(dinfo.output\_scanline-1)\*800+col)=(red<<16)|(green<<8)|blue;

pFrameBuffer[dinfo.output\_scanline-1][col]=(red<<16)|(green<<8)|blue;

}

}

jpeg\_finish\_decompress(&dinfo);

munmap(pFrameBuffer,800\*480\*4);

free(pLineColor);

fclose(fp);

close(fd);

exit:

jpeg\_destroy\_decompress(&dinfo);

return 0;

}

进程控制

1. 进程的概念

进行中的程序，是程序一次运行。

程序是“死”的，位于文件系统中；

进程是“活的”，位于内存中；

“死”的程序因为加载到内存而变成“活”进程。一个进程一般存在下面三种状态：

1. 运行态：该进程占有cpu运行；
2. 就绪态：进程处在“万事俱备，只欠东风”的状态；
3. 阻塞态：运行中的进程因为等待某个事情而阻塞；

当程序被加载时进入就绪状态，处于就绪状态的进程因为调度而得到与运行，运行中的进程因为时间片到而转入就绪状态，运行中的进程因等待某个事件（如scanf）而进入阻塞态，阻塞中的进程因为等待事件到达而被唤醒进入就绪状态

一般情况下我们都是动过鼠标双击某一个可执行文件或者通过命令行方式让一个程序转变成一个进程，但有时候我们需要通过程序来产生一个进程。之所以需要程序自己产生一个进程，很多时候在完成一大的任务时，如果这个大任务中的多个小任务可以同时进行，这个时候可产生一个 或多个子进程来完成各个小任务以加快任务的处理进度。

1. 与进程控制相关的API
2. fork

功能描述：产生子进程

原型：

#include <unistd.h>

pid\_t fork(void);

参数描述：无

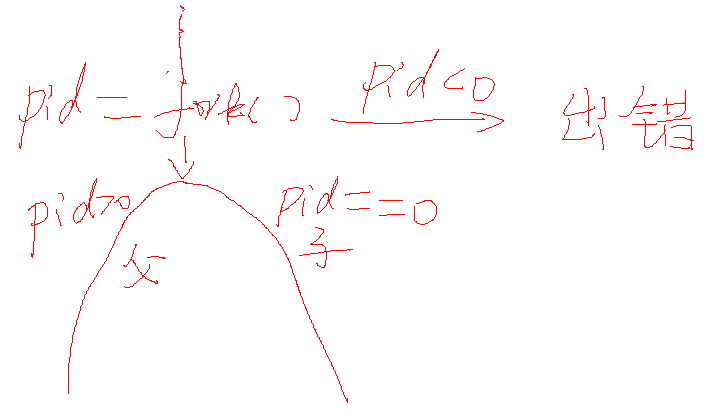
返回值：

成功则返回一个非负的整数：

（1）在子进程中的返回值为0；

（2）父进程中大于0，为子进程的进程号；

出错则小于0



通过fork()产生的子进程和其“父亲”非常的像是：代码、数据完全一样，但是占据的空间是不相同的，子进程有其独立的地址空间！！

进程可以通过getpid()来获取其进程好，getppid()获取父进程号。

课堂小任务：

父进程产生子进程，父子进程分别打印出自己的进程号后退出。

参考代码（注意区分父子进程所看到的fork()的返回值是不一样的）：

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

int main()

{

pid\_t pid;

printf("I am parent process and is going to create child process!\n");

pid=fork();

if(pid<0)

{

perror("Failed to create child process!");

return -1;

}

else if(pid==0)

{

printf("I am child process and my pid is %d\n",getpid());

}

else

{

printf("I am parent(pid=%d) and succeed in creating child process(pid=%d)!\n",getpid(),pid);

}

return 0;

}

课堂任小任务：

分析下面的代码，说出父子进程中输出的global和local分别是多少?编译运行后输出结果又是多少呢？解释原因！

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

int global=12;

int main()

{

pid\_t pid;

int local=20;

printf("I am parent process and is going to create child process!\n");

pid=fork();

if(pid<0)

{

perror("Failed to create child process!");

return -1;

}

else if(pid==0)

{

sleep(1);

printf("I am child process and my pid is %d\n",getpid());

printf("global=%d,local=%d in child process\n",global,local);

}

else

{

global+=50;

local+=10;

printf("I am parent(pid=%d) and succeed in creating child process(pid=%d)!\n",getpid(),pid);

printf("global=%d,local=%d in parent process\n",global,local);

}

return 0;

}

1. wait

原型：

#include <sys/types.h>

#include <sys/wait.h>

pid\_t wait(int \*status);

功能描述：

如果调用这有子进程，调用者一直阻塞至一个子进程结束才返回；如果调用者没有子进程则立即返回；

参数说明：

status所指向的空间保存子进程返回的状态号！！如果为NULL,则表示调用者不关心子进程的返回状态；该状态可以通过：WEXITSTATUS(status)来获取

返回：

成功为结束的子进程的进程号；失败则小于0；

课堂小任务：

分析运行下面的代码：尽管子进程睡眠了2秒钟，但父进程因为调用了wait而一定等到子进程结束了他才结束，并打印出了子进程的状态。

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

int global=12;

int main()

{

pid\_t pid;

int local=20;

printf("I am parent process and is going to create child process!\n");

pid=fork();

if(pid<0)

{

perror("Failed to create child process!");

return -1;

}

else if(pid==0)

{

sleep(1);

printf("I am child process and my pid is %u,pid=%u,my parent pid=%u\n",getpid(),pid,getppid());

printf("global=%d,local=%d in child process\n",global,local);

sleep(2);

return 8;

}

else

{

int status;

int ret;

global+=50;

local+=10;

printf("I am parent(pid=%u) and succeed in creating child process(pid=%u),my parent pid=%u!\n",getpid(),pid,getppid());

printf("global=%d,local=%d in parent process\n",global,local);

ret=wait(&status);

if(ret>0)

{

printf("Pid=%u is over and exit status=%d\n",ret,WEXITSTATUS(status));

}

}

return 0;

}

1. exit和\_exit

#include <unistd.h>

void \_exit(int status);

#include <stdlib.h>

void exit(int status);

在进程的任何一个地方调用上面的两个函数中的一个都会导致进程立即结束，但二者有区别：

1. exit在进程退出前会刷新缓冲区：对于c标准的IO函数（fread fwrite printf fgets等 ）和真正的IO文件中间有一块内存区作为缓冲的，当缓冲区中的数据的数量满足一定的条件时才会发生整整的输入、输出，例如，printf就是基于行缓冲的，也就是说当缓冲区接收到一个换行符或者缓冲区满或者执行了fflush(stdout)时把缓冲区中的数据输出到标准的输出设备上。
2. \_exit不会刷新缓冲区

课堂小任务：

分析、运行下面四个程序的运行结果。

程序一：

#include <stdio.h>

#include <unsitd.h>

int main()

{

printf(“Hello,world!”);

exit(0);

return 0;

}

程序二：

#include <stdio.h>

#include <unsitd.h>

#include <stdlib.h>

int main()

{

printf(“Hello,world!”);

\_exit(0);

return 0;

}

程序三：

#include <stdio.h>

#include <unsitd.h>

#include <stdlib.h>

int main()

{

printf(“Hello,world!”);

return 0;

}

程序四：

#include <stdio.h>

#include <unsitd.h>

#include <stdlib.h>

int main()

{

printf(“Hello,world!”);

fflush(stdout);

\_exit(0);

return 0;

}

1. 加载外部程序

第一种方法：

#include <stdlib.h>

int system(const char \*command);

功能描述：

启动command之所指向的可执行文件，当所启动的程序执行完毕后system才返回；

参数说明：

command：指向可执行文件的文件名（可带路径）

返回值：

出错是返回-1;

成功则返回命令的执行状态；

第二种方法：

如果用system方法所启动的外部程序是不会导致调用这本身的代码和数据丢失；

第二种方法所使用的exec函数族会导致调用者本身的代码和数据丢失，exec函数族多用在子进程不要父进程继承过来的代码和数据，需要“脱胎换骨”——用新的程序的代码和数据

的时候。

#include <unistd.h>

extern char \*\*environ;

**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 \*path, char \*const argv[]);**

int execvp(const char \*file, char \*const argv[]);

int execvpe(const char \*file, char \*const argv[],

char \*const envp[]);

课堂小任务：

分析运行下面程序，并解释原因

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

int main()

{

int pid;

pid=fork();

if(pid==0)

{

execl("/bin/ls","ls","-l",NULL);//用列表的形式传输命令的参数，最后的一个参数必须为NULL

printf("Child is over\n");

}

else if(pid<0)

{

perror("Failed to create child!");

}

printf("hahahahhahaha\n");

return 0;

}

用execv改编上面的程序

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

int main()

{

int pid;

char \*argv[]={“ls”,”-l”,NULL}

pid=fork();

if(pid==0)

{

execv("/bin/ls",argv);//用列表的形式传输命令的参数，最后的一个参数必须为NULL

printf("Child is over\n");

}

else if(pid<0)

{

perror("Failed to create child!");

}

printf("hahahahhahaha\n");

return 0;

}

本周的课后作业：

1. 继续晚上学生信息管理系统和数码相框；
2. 把目前为止所练习的所有课堂小任务重新做一遍以对所学系统编程知识的梳理。

下周内容：

进程通信