目录

[1-1 ※写一个"标准"宏MIN ，这个宏输入两个参数并返回较小的一个 4](#_Toc500068135)

[1-2 ※用预处理指令#define 声明一个常数，用以表明1年中有多少秒 5](#_Toc500068136)

[1-3 ※把上面的define.c 写成一个Makefile文件并编译出可执行文件 6](#_Toc500068137)

[1-4 ※交换两个数的值 6](#_Toc500068138)

[1-5 ※int main（int argc char \*argv[]）的用法 7](#_Toc500068139)

[1-6 ※fork（）的用法 9](#_Toc500068140)

[1-7 ※exit（）用法 12](#_Toc500068141)

[1-8 ※sleep（）用法 12](#_Toc500068142)

[1-9 ※用fprintf()/fscanf() 13](#_Toc500068143)

[1-10 ※用extern 存储类 15](#_Toc500068144)

[1-11 ※大端小端判断 16](#_Toc500068145)

[1-12 ※Linux命令 set 18](#_Toc500068146)

[1-13 ※Linux命令 man 1,2,3.. 18](#_Toc500068147)

[1-15 ※C语言字符操作 19](#_Toc500068148)

[1-16※struct的初始化 21](#_Toc500068149)

[1-17※关键字volatile有什么含意?并给出三个不同的例子 22](#_Toc500068150)

[1-18※unsigned int a=1; int b= -20 (a+b)>0 22](#_Toc500068151)

[1-19※Linux I/O操作 open read write open（系统） 23](#_Toc500068152)

[1-20※Linux socket 本地通信 26](#_Toc500068153)

[1-21※同步和异步使用 32](#_Toc500068154)

[1-22※select（）轮询函数 33](#_Toc500068155)

[1-23※network status check 34](#_Toc500068156)

[1-24※共享内存 35](#_Toc500068157)

[1-25※新建Ubuntu 用户 36](#_Toc500068158)

[1-26※Git 操作问题，怎么根据xml 下载代码 36](#_Toc500068159)

[1-27※定时执行脚本的例子 37](#_Toc500068160)

[1-28※配置gerrit git 服务器 38](#_Toc500068161)

[Gerrit与客户代码关联 42](#_Toc500068162)

[1-29※定时器 44](#_Toc500068163)

[1-30※void \*p 45](#_Toc500068164)

[1-31※makefile 自动化变量 45](#_Toc500068165)

[1-32※sizeof()字节长度 46](#_Toc500068166)

[1-33※C++ 引用与引用作为函数的参数 48](#_Toc500068167)

[1-34※音视频解码格式 49](#_Toc500068168)

[1-35※gcc thread.c -o thread -lpthread 50](#_Toc500068169)

[1-35※线程互斥锁 SoC：System on Chip 50](#_Toc500068170)

[1-36※算法 52](#_Toc500068171)

[1-37※算法 52](#_Toc500068172)

# 1-1 ※写一个"标准"宏MIN ，这个宏输入两个参数并返回较小的一个

#include<stdio.h>

**#define MIN(x,y) x<y? x:y //后面的比较：输出小的输 x,Y的值顺序不要反**

int main()

{

static int a=5;

static int b=9;

int c;

scanf("plate 输入数字：%d%d\n",&a,&b);

printf("a=%d\n b=%d\n",a,b);

**c=MIN(a,b); //使用**

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

}

# 1-2 ※用预处理指令#define 声明一个常数，用以表明1年中有多少秒

#include<stdio.h>

**#define COUNT\_YEAR\_MIN ((unsigned long)(60 \* 60 \* 24 \* 365))//注长度**

int main()

{

int c;

**c=COUNT\_YEAR\_MIN; //使用**

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

}

# 1-3 ※把上面的define.c 写成一个Makefile文件并编译出可执行文件

edit: define.o

**cc -o define define.o @ 必须tab键开头 @**

define.o:define.c

cc -c define.c

clean:

rm **define** define.o

**带变量的makefile**

**objects = define.o**

edit:$(objects)

cc -o define $(objects)

define.o: define.c

cc -c define.c

clean:

rm define $(objects)

# 1-4 ※交换两个数的值

#include<stdio.h>

**void swap(int \*a,int \*b)**

{

int c;

c=\*a;

\*a=\*b;

\*b=c;

}

int main()

{

int d=10;

int f=55;

swap(&d,&f);

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

}

# 1-5 ※int main（int argc char \*argv[]）的用法

#include <stdio.h>

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

{

int i;

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

{

printf("%s\n",argv[i]);

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

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

}

system("pause");

return 0;

}

**解释：**

argc：表示执行可运行程序后的参数

argv[]: 此可执行文件的存储路径、程序变量

**如：**执行./main 无参数

argc: 为1

argv[i]：为工程的路径和名称如：C:\Qt\Qt5.3.2\Tools\QtCreator\bin\build-untitled2-Desktop\_Qt\_5\_3\_MinGW\_32bit-Debug\debug\debug\untitled2.exe

运行结果：

C:\Qt\Qt5.3.2\Tools\QtCreator\bin\build-untitled2-Desktop\_Qt\_5\_3\_MinGW\_32bit-Debug\debug\debug\untitled2.exe

i=0

argc=1

**如：**执行./main sss.c(什么都行)

argc: 为2

argv[i]：如下

运行结果：

C:\Qt\Qt5.3.2\Tools\QtCreator\bin\build-untitled2-Desktop\_Qt\_5\_3\_MinGW\_32bit-Debug\debug\debug\untitled2.exe

i=0

argc=2

ssss.c

i=1

argc=2

# 1-6 ※fork（）的用法

#include<sys/types.h>

#include<unistd.h>

#include<stdio.h>

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

{

pid\_t pid;

pid = fork();

if (pid < 0 ) {

printf("error!");

} else if( pid ==0 ) {

printf("This is the child process!\n");

} else{

printf("This is the parent process!\n child process id = %d\n", pid);

}

return 0;

}

疑问：为什么每次执行结果不一样

无固定顺序是因为子程序与父程序并发导致的，因为在用fork（）函数创建的3个进程同时在cpu中运行，因为3个进程的本质是一样的，所以就会抢占资源，因此形成一个队列，因为每次的排序不同，故此输出的结果也会不同。

**●fork的用法二**

#include<sys/types.h>

#include<unistd.h>

#include<stdio.h>

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

{

pid\_t pid;

pid = fork();

if(pid < 0)

printf("error\n");

else if(pid == 0){ //等于0是子进程

printf("The child pid = %d\n",getpid());

}

else{

printf("The parent pid = %d\n",getpid());

}

}

Fork 返回值：**0 成功创建子进程**

>0 成功创建父进程，返回进程PID

-1 失败返回父进程

**运行结果：**

**The parent pid = 16938**

**The child pid = 16939**

# 1-7 ※exit（）用法

主要用到子函数中

exit（0）：正常运行程序并退出程序；

exit（1）：非正常运行导致退出程序；

* + main函数调用return
  + 调用exit
  + 调用\_exit
  + 调用abort
  + 被一个信号终止

区别 abort与exit

exit会做一些释放工作**：释放所有的静态的全局的对象，缓存，关掉所有的I/O通道**，然后终止程序。如果有函数通过atexit来注册，还会调用注册的函数。不过，如果atexit函数扔出异常的话，就会直接调用terminate。  
abort：立刻terminate程序，**没有任何清理工作**。

# 1-8 ※sleep（）用法

Sleep（1）：1s 秒

Usleep( 1000) 1000 毫秒

# 1-9 ※用fprintf()/fscanf()

#include <stdio.h>

#include <stdlib.h>

#include <errno.h>

#define FILE\_PATH "./demo.txt" // 文件路径

int main(){

int i=20;

char name[20]="yang"; // 姓名

int age=20; // 年龄

int studNo=45454; // 学号

float score=222; // 平均分

FILE \*fp; // 文件指针

// 判断文件是否能够正确创建/打开

if( (fp=fopen(FILE\_PATH,"wt+")) == NULL ){

perror(FILE\_PATH);

exit(1);

}

// 从控制台输入学生信息并写入文件

printf("(please enter以空格分隔)：\n");

// scanf("%s %d %d %f", name, &age, &studNo, &score);

while(1)

{

static age=555;

studNo--;

score--;

fprintf(fp,"%s\t%d\t%d\t%f\n", name, age, studNo, score);

// 刷新缓冲区，将缓冲区的内容写入文件

fflush(fp);

// 重置文件内部位置指针，让位置指针指向文件开头

rewind(fp);

// 从文件中读取学生信息

//printf("read file:\n");

while(fscanf(fp, "%s\t%d\t%d\t%f", name, &age, &studNo, &score) != EOF){

printf("%s %d %d %f\n", name, age, studNo, score);

}

if(i<0) break;

i--;

}

fclose(fp);

return EXIT\_SUCCESS;

}

# 1-10 ※用extern 存储类

extern 修饰符通常用于当有**两个或多个文件共享相同的全局变量或函数**的时候。

**如 extern\_1.c**

#include <stdio.h>

**extern void swap(int \*a,int \*b)**

{

int c;

c=\*a;

\*a=\*b;

\*b=c;

}

**如extern\_2.c**

#include <stdio.h>

int main()

{

int i=0,h=99999;

printf("before i= %d\t h= %d\n",i,h);

**swap(&i,&h);//能使用其他文件的函数**

printf("after i= %d\t h= %d\n",i,h);

return 0;

}

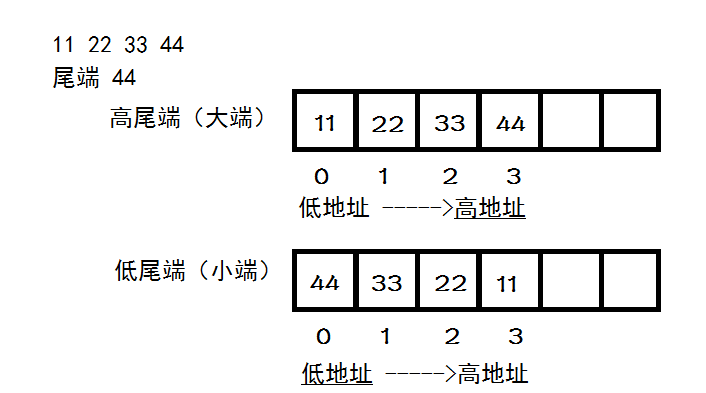
输出：before i= 0 h= 99999

after i= 99999 h= 0

# 1-11 ※大端小端判断

大端模式：是指数据的高字节保存在内存的低地址中，而数据的低字节保存在内存的高地址端。

小端模式，是指数据的高字节保存在内存的高地址中，低位字节保存在在内存的低地址端。

intel芯片是小端(修改分区表时要注意)，**单片机一般为大端**

编程检测：

#include <stdio.h>

int check()

{

union UN //union是共享内存的

{

char c;

int i;

}un;

un.i = 1;

return un.c;

}

int main(void)

{

if(check()==1)

printf("little\n");

else

printf("big\n");

return 0;

}

# 1-12 ※Linux命令 set

作用主要是**显示系统中已经存在的shell变量**，以及**设置shell变量的新变量值。**使用set更改shell特性时，符号"+"和"-"的作用分别是打开和关闭指定的模式  
来自: <http://man.linuxde.net/set>

**set 用来显示本地变量**  
env 用来显示环境变量  
export 用来显示和设置环境变量s

set 显示当前shell的变量，包括当前用户的变量  
env 显示当前用户的变量  
export 显示当前导出成用户变量的shell变量

每个shell有自己特有的变量（set）显示的变量

**为当前终端设置代理：**

**vim ~/.bashrc**

export http\_proxy=http://10.1.71.246:2468

# 1-13 ※Linux命令 man 1,2,3..

1、standard command （标准命令）

**2、System calls （系统调用）  
3、Library functions （库函数）**方法是把一些常用到的函数编完放到一个文件里，供不同的人进行调用。调用的时候把它所在的文件名用#include<>加到里面就可以了。一般是放到lib文件里的。  
4、Special devices （设备说明）  
5、File formats （文件格式）

# 1-15 ※C语言字符操作

|  |  |
| --- | --- |
| 1 | **strcpy(s1, s2);** 复制字符串 s2 到字符串 s1。 |
| 2 | **strcat(s1, s2);** **连接字符串 s2 到字符串 s1 的末尾**。 |
| 3 | **strlen(s1);** 返回字符串 s1 的长度。 |
| 4 | **strcmp(s1, s2);** 如果 s1 和 s2 是相同的，则返回 0；如果 s1<s2 则返回小于 0；如果 s1>s2 则返回大于 0。 |
| 5 | **strchr(s1, ch);** **返回一个指针，指向字符串 s1 中字符 ch 的第一次出现的位置**。 |
|  | **strstr(s1, s2);** 该函数是查找子字符串，返回值为**子字符串第一次出现的位置**及其后面的字符串，如果没查找到则返回为NULL  strtok 函数  char str[] = "2016/08/29 20:06:55";  char\* result = strtok(str, "/");  while (result)  {  printf("result = %s\n", result);//结果：result = 2016  // result = 08  // result = 29 20:06:55  result = strtok(NULL, "/");  }  字符在第一次调用时strtok()必需给予参数str字符串，往后的调用则将参数str设置成NULL，每次调用成功则返回指向被分割出片段的指针。  **atoi 函数 str 转换成int型**  **atof 函数 float**  **atol 函数 long**  **字符串 装换的函数，**  **rand 与 srand函数**  **随机函数**  int t = (int)time(NULL);  srand(t);  for (int i = 0; i < 3; i++)  {  printf("randNum = %d\n", rand());  //输出：randNum = 14824  // randNum = 16692  // randNum = 15958  }  **Strcpy 和memcpy区别**  1、复制的内容不同。**strcpy只能复制字符串**，而memcpy可以**复制任意数据结构**的内容，例如字符数组、整型、结构体、类等。  2、复制的方法不同。**strcpy不需要指定长度**，它遇到被复制字符的**串结束符"\0"才结束**，**所以容易溢出**。memcpy则是根据其第3个参数决定复制的长度。  3、用途不同。通常在复制**字符串时用strcpy**，而需要复制其他类型数据时则一般用memcpy |

# 1-16※struct的初始化

#include <stdio.h>

#include <string.h>

struct stud //定义一个结构体

{

int age;

char \*name;

int score;

};

void out(struct stud \*s) //**定义一个指向结构体的指针**

{

printf("age=%d\n",s->age);//取出初始化结构体的值

printf("name=%s\n",s->name);/\*用指向该结构的指针,访问结构的成员，必须使用 -> 运算符\*/

printf("score=%d\n",s->score);

}

int main()

{

struct stud std= //初始化结构体stud 类型std

{ //**声明 std，类型为 stud**

.age = 18,

.name = "yang",

.score = 99,

};

out(&std); //**通过传 std 的地址来输出 std 信息**

}

# 1-17※关键字volatile有什么含意?并给出三个不同的例子

1) 并行设备的硬件寄存器（如：状态寄存器）

2) 一个中断服务子程序中会访问到的非自动变量(Non-automatic variables)

3) 多线程应用中被几个任务共享的变量

# 1-18※unsigned int a=1; int b= -20 (a+b)>0

#include <stdio.h>

int main(void)

{

unsigned int a=2;

int b=-20;

if((a+b)>10000000)**//如果无符号的变量和有符号的变量相运算时会是一个非常大的数 [除了-运算]**

是当表达式中存在有符号类型和无符号类型时所有的操作数都自动转换为**无符号类型**。因此-20变成了一个非常大的正整数，所以该表达式计算出的结果大于6。

{

printf("a+b=%u\n",a+b);

printf("a-b=%u\n",a-b);

printf("a\*b=%u\n",a\*b);

printf("b/a=%u\n",b/a);

}else

{

printf("a+b<0\n");

}

return 0;

}

a+b=4294967278

a-b=22

a\*b=4294967256

b/a=2147483638

%d 有符号10进制整数 %u 无符号10进制整数

# 1-19※Linux I/O操作 open read write open（系统）

flags：（falgs参数可通过“|”组合构成，只读、只写、读写 这三种方式是互斥的，不可同时使用）  
        O\_RDONLY（只读方式打开），  
        O\_WRONLY（只写方式打开），  
        O\_RDWR（读/写方式打开），  
        O\_CREAT（如果文件不存在，就创建新的文件），  
        O\_EXCL（如果使用O\_CREAT时，文件存在，则可返回错误信息），  
        O\_TRUNC（如果文件已存在，且以只读或只写方式打开，则先删除文件中的原有数据），  
        O\_APPEND（以添加方式打开文件，在打开文件的同时，文件指针指向文件末尾）

 为了防止对文件的意外操作，往往要以合适的方式打开文件（只读，只写），每个文件只负责一个特定的用  
    途，有利于提高这些文件的重复利用

#include <unistd.h>

#include <sys/stat.h>

#include <sys/types.h>

#include <fcntl.h>

#include <stdio.h>

#define NAME "file.log"

int main(void){

int fd,size,w;//fd：文件描述符，用于判断打开|新建文件是否成功，size：读取到的字节数

char s[] = "hellossssssssss\n----->\n52661314ll";//需要写入的字符串

char **buffer**[80];//存储读出数据的缓冲区

fd = **open**(NAME, **O\_WRONLY | O\_CREAT**);//以只写方式打开文件，若不存在则创建文件

if (**fd == -1**){

printf("Open or create file failed.\n");

return -1;

}

w=**write**(fd, s, sizeof(s));//向该文件写入字符串

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

close(fd);

fd = **open**(NAME, O\_RDONLY);//以只读方式打开文件

if (fd == -1){

printf("Open file failed.\n");

return -1;

}

size = **read**(fd, buffer, sizeof(buffer));//读取文件内容到buffer并返回读取的字符个数

printf("size==%d\n",size);/**/w=size 字符长度**

**close**(fd);

printf("%s", buffer);

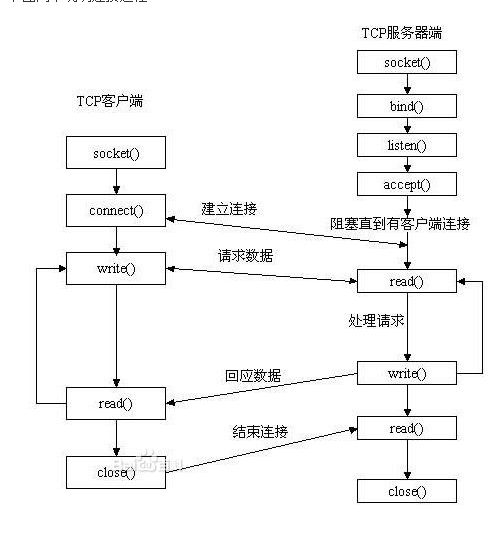
return 0;

}

# 1-20※Linux socket 本地通信

* **理论篇**

两个进程有唯一的PID，在本地进程通讯中我们可以使用PID来唯一标示一个本地进程，但PID只在本地唯一，网络中的两个进程PID冲突几率很大。IP层的ip地址可以唯一标示主机，而TCP层协议和端口号可以唯一标示主机的一个进程，这样我们可以利用ip地址＋协议＋端口号唯一标示网络中的一个进程。socket起源于Unix，而Unix/[Linux](http://lib.csdn.net/base/linux" \t "Linux知识库)基本哲学之一就是“一切皆文件”，都可以用“打开open –> 读写write/read –> 关闭close”模式来操作。Socket文件。(http://blog.csdn.net/sight\_/article/details/8138802)



* **Socket()函数**

**int socket(int domain, int type, int protocol)**

eg:int socket(AF\_INET,SCOK\_STREAM,0) 【常用】

 -**domain**：即协议域，又称为协议族（family）。常用的协议族有，**AF\_INET（PF\_INET没有区别）**、AF\_INET6、AF\_LOCAL（或称AF\_UNIX，Unix域socket）、AF\_ROUTE等等。协议族决定了socket的地址类型，在通信中必须采用对应的地址，如AF\_INET决定了要用ipv4地址（32位的）与端口号（16位的）的组合、**AF\_UNIX决定了要用一个绝对路径名作为地址**。

  **-type：**指定socket类型。常用的socket类型有，SOCK\_STREAM、SOCK\_DGRAM、SOCK\_RAW、SOCK\_PACKET、SOCK\_SEQPACKET等等（socket的类型有哪些？）。

 -**protocol**：就是指定协议。常用的协议有，IPPROTO\_TCP、IPPTOTO\_UDP、IPPROTO\_SCTP、IPPROTO\_TIPC等，它们分别对应TCP传输协议、UDP传输协议、STCP传输协议、TIPC传输协议（这个协议我将会单独开篇讨论！）。

注意：并不是上面的type和protocol可以随意组合的，如SOCK\_STREAM不可以跟IPPROTO\_UDP组合。**当protocol为0时（常用），会自动选择type类型对应的默认协议**。

当我们调用**socket**创建一个socket时，返回的socket描述字它存在于协议族（address family，AF\_XXX）空间中，但没有一个具体的地址。如果想要给它赋值一个地址，就必须调用**bind()函数（绑定一个具体的地址）**，否则就当调用connect()、listen()时系统会自动随机分配一个端口。

返回值：

success :返回一个新的socket fail: 返回 - 1

* **bind()函数**

int bind(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen);

**- sockfd**：即socket描述字，它是通过fd=socket()函数

**- addr**：一个const struct sockaddr \*指针，**指向要绑定给sockfd的协议地址**。这个地址结构根据地址创建socket时的地址协议族的不同而不同.

**Unix域对应的是**：

**#define UNIX\_PATH\_MAX 108**

struct sockaddr\_un {

sa\_family\_t sun\_family; /\* AF\_UNIX \*/

char sun\_path[UNIX\_PATH\_MAX]; /\* pathname \*/

};

**ipv4对应的是：**

struct sockaddr\_in {

sa\_family\_t sin\_family; /\* address family: AF\_INET \*/

in\_port\_t sin\_port; /\* port in network byte order \*/

struct in\_addr sin\_addr; /\* internet address \*/

};

/\* Internet address. \*/

struct in\_addr {

uint32\_t s\_addr; /\* address in network byte order \*/

};

ipv6对应的是：

struct sockaddr\_in6 {

sa\_family\_t sin6\_family; /\* AF\_INET6 \*/

in\_port\_t sin6\_port; /\* port number \*/

uint32\_t sin6\_flowinfo; /\* IPv6 flow information \*/

struct in6\_addr sin6\_addr; /\* IPv6 address \*/

uint32\_t sin6\_scope\_id; /\* Scope ID (new in 2.4) \*/

};

**主要事项：网络字节序与主机字节序**

**主机字节序**就是我们平常说的大端和小端模式：不同的CPU有不同的字节序类型，这些字节序是指整数在内存中保存的顺序，这个叫做主机序。引用标准的Big-Endian和Little-Endian的定义如下：

　　a) Little-Endian就是低位字节排放在内存的低地址端，高位字节排放在内存的高地址端。

　　b) Big-Endian就是高位字节排放在内存的低地址端，低位字节排放在内存的高地址端。

**网络字节序**：4个字节的32 bit值以下面的次序传输：首先是0～7bit，其次8～15bit，然后16～23bit，最后是24~31bit。这种传输次序称作大端字节序。**由于TCP/IP首部中所有的二进制整数在网络中传输时都要求以这种次序，因此它又称作网络字节序。**字节序，顾名思义字节的顺序，就是大于一个字节类型的数据在内存中的存放顺序，一个字节的数据没有顺序的问题了。

所以：在将一个地址绑定到socket的时候，**请先将主机字节序转换成为网络字节序，而不要假定主机字节序跟网络字节序一样使用的是Big-Endian。**由于这个问题曾引发过血案！公司项目代码中由于存在这个问题，导致了很多莫名其妙的问题，所以请谨记对**主机字节序不要做任何假定，务必将其转化为网络字节序再赋给socket。**

返回值：

success :返回一个新的socket fail: 返回 - 1

* **addrlen**：对应的是地址的长度。
* **listen（）/ connect（）函数**

int listen(int sockfd, int backlog);

int connect(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen);

listen函数的第一个参数即为要监听的socket描述字，第二个参数为相应socket可以排队的最大连接个数。**socket()函数创建的socket默认是一个主动类型的，listen函数将socket变为被动类型的，等待客户的连接请求**。

connect函数的第一个参数即为客户端的socket描述字，第二参数为服务器的socket地址，**第三个参数为socket地址的长度。客户端通过调用connect函数来建立与TCP服务器的连接。**

返回值：

success :返回一个新的socket fail: 返回 - 1

* **accept（）函数（while（1）{ accept（）阻塞在这里}）**

int accept(int sockfd, struct sockaddr \*addr, socklen\_t \*addrlen);

TCP服务器端依次调用socket()、bind()、listen()之后，**就会监听指定的socket地址了**。TCP客户端依次调用socket()、connect()之后就想TCP服务器发送了一个连接请求。TCP服务器监听到这个请求之后，**就会调用accept()函数取接收请求，这样连接就建立好了。之后就可以开始网络I/O操作了，即类同于普通文件的读写I/O操作**。

accept函数的第一个参数为服务器的socket描述字，第二个参数为指向struct sockaddr \*的指针，**用于返回客户端的协议地址**，第三个参数为协议地址的长度。如果accpet成功，那么其返回值是由内核自动生成的一个全新的描述字，**代表与返回客户的TCP连接**。

注意：accept的第一个参数为服务器的socket描述字，是服务器开始调用socket()函数生成的，称为监听socket描述字；**而accept函数返回的是已连接的socket描述字**。一个服务器通常仅只创建一个监听socket描述字，它在该服务器的生命周期内一直存在。内核为每个由服务器进程接受的客户连接创建了一个已连接socket描述字，当服务器完成了对某个客户的服务，相应的已连接socket描述字就被关闭。

返回值：

success :返回一个新的socket fail: 返回 - 1

# 1-21※同步和异步使用

同步：（用于不耗时的操作或者多线程用同一个资源时只能有一个进程访问资源，其他线程必须等待。同步的机制: 临界区、互斥、信号量和事件）

A

B

C

异步：（有耗时的操作时）

A

B

C

或者不用返回

（正常来说是不用回的）

# 1-22※select（）轮询函数

select机制中提供了一个数据结构 struct fd\_set ，可以理解为一个集合，实际上是一个位图，每一个特定为来标志相应大小文件描述符，这个集合中存放的是文件描述符(file descriptor)，即文件句柄(也就是位图上的每一位都能与一个打开的文件句柄(文件描述符)建立联系，这个工作由程序员来完成)，这可以是我们所说的普通意义的文件，当然Unix下任何设备、管道、FIFO等都是文件形式，全部包括在内，所以毫无疑问一个socket就是一个文件，socket句柄就是一个文件描述符。fd\_set集合可以通过一些宏由人为来操作，程序员通过操作4类宏，来完成最fd\_set的操作：

(1)、FD\_ZERO(fd\_set \*)  清空一个文件描述符集合；

(2)、FD\_SET(int ,fd\_set \*)将一个文件描述符添加到一个指定的文件描述符集合中；

(3)、FD\_CLR(int ,fd\_set\*)       将一个给定的文件描述符从集合中删除；

(4)、FD\_ISSET(int ,fd\_set\* )检查集合中指定的文件描述符是否可以读写。

深入的理解select模型的关键点在于理解fd\_set，为了说明方便，我们取fd\_set长度为1个字节，fd\_set中的每一个bit可以对应一个文件描述符fd。则1字节长的fd\_set最大可以对应8个fd。

（1）执行fd\_set set；FD\_ZERO(&set)；则set用位表示为 0000,0000 。

（2）若fd = 5 ，则执行 FD\_SET(fd,&set)后，set变为 0001，0000 （第5位置为1）

（3）若再加入fd=2 ，fd=1，则set变为 0001.0011

（4）执行select(6,&set,0,0,0)阻塞等待

（5）若fd=1,fd=2上都发生可读事件，则select返回，此时set变为0000，0011。没有可读事件发生时 fd = 5 被清空。

2、struct timeval，一个大家常用的结构，用来代表时间值，有两个成员，一个是秒数，另一个是毫秒数。

1. **struct** timeval
2. {
3. **long** tv\_sec;    //second
4. **long** tv\_usec;   //microsecond
5. };

# 1-23※network status check

static int

ping\_ipv6\_test (char \*ipv6\_gateway\_str)

{

printf("++++++[yq]--ping\_ipv6\_test :: ipv6\_gateway\_str -- %s [%s] [%d]\n", ipv6\_gateway\_str, \_\_FUNCTION\_\_, \_\_LINE\_\_);

FILE \*fp;

size\_t len = 0;

char \*pos;

int ret,percent = 100;

char\* line = NULL; char cmd[80]={0};

sprintf(cmd, "ping -c 1 -q -n %s > /work/ping.txt", ipv6\_gateway\_str);

system(cmd);

fp = fopen("/work/ping.txt", "r");

if (!fp){

goto error1;

}

while (getline (&line, &len, fp) != -1) {

if (strncmp ("packets transmitted", line + 2, 19) != 0) /\* buggy \*/

{

continue;

}

pos = strchr (line, '%');

if (pos == 0) {

goto error1;

}

while (\*pos != ' ')

{ pos--;

}

sscanf (pos, " %d%% loss", &percent);

printf("++++++[yq]--ping\_ipv6->pos -- %s [%s] [%d]\n", pos, \_\_FUNCTION\_\_, \_\_LINE\_\_);

printf("++++++[yq]--ping\_ipv6->percent -- %d [%s] [%d]\n", percent, \_\_FUNCTION\_\_, \_\_LINE\_\_);

if (percent == 0) {

printf("++++++[yq]--ping\_ipv6\_result:: OKOK -- [%s] [%d]\n", \_\_FUNCTION\_\_, \_\_LINE\_\_);

ret = PING\_TEST\_IPV6\_OK;

}else {

printf("++++++[yq]--ping\_ipv6\_result:: NGNG -- [%s] [%d]\n", \_\_FUNCTION\_\_, \_\_LINE\_\_);

ret = PING\_TEST\_IPV6\_LOSS\_ERROR;

}

}

error1:

free(line);

ping\_err\_ipv6 = ret;

printf ("++++++++++[yq]--ping\_ipv6 ::ret =++++++++++++++++++++++++++ %d [%s] [%d]\n",ret,\_\_FUNCTION\_\_, \_\_LINE\_\_);

return ret;

}

# 1-24※共享内存

明显，由于进程可以直接读写内存，所以采用共享内存进行进程间通信的一个优点就是快速、效率高。像我们前面介绍的匿名管道和命名管道通信方式，需要在用户空间和内核空间之间进行数据拷贝，而共享内存通信方式只在内存中操作，要高效得多。

另外一方面，由于多个进程对同一段内存区域都具有读写权限，在共享内存通信中，进程间的同步问题就显得尤为重要。必须保证同一时刻只有一个进程对共享内存进行写操作，否则会导致数据被覆盖。而共享内存通信又没有提供同步机制，需要使用诸如信号量等手段进行同步控制，这又增加了其复杂性。

# 1-25※新建Ubuntu 用户

删除用户

sudo userdel

添加用户

sudo useradd yq -m -s /bin/bash

设置密码：

sudo passwd yq

# 1-26※Git 操作问题，怎么根据xml 下载代码

Cd .repo

备份 manifest.Xml

自己的xml改成 manifest.xml

替换成自己的xml

Repo sync ok

# 1-27※定时执行脚本的例子

**登陆服务器**

ssh [user@10.1.59.253](mailto:user@10.1.59.253)

使用以下：

crontab -l :查看定时任务

crontab -r ：删除定时任务

crontab -e :添加定时任务

**crontab -e**

no crontab for yang.q - using an empty one

Select an editor. To change later, run 'select-editor'.

1. /bin/ed

2. /bin/nano <---- easiest

3. /usr/bin/emacs24

4. /usr/bin/mcedit

5. /usr/bin/vim.basic

6. /usr/bin/vim.tiny

Choose 1-6 [2]: 5 :选择打开方式 选5

删除所有的注释的东西

添加：

\*/30 \* \* \* \* /home/user/repo\_jp.sh >> /home/user/log\_jp.log ：每整点，30分钟者执行一次，写入执行的 .sh脚本的log

**自动执行同步代码脚本**

#!/bin/sh

PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin

export PATH

cd /home/user/ && date >time\_jp.txt

cd /harddisk/tv-scsoft-repo/scsoft-165-master/ && /home/user/bin/repo sync

# 1-28※配置gerrit git 服务器

git据点服务器创建代码库目录

$cd

$mkdir repo

4.安装Gerrit

$java -jar gerrit.war init -d gerrit

注意：

1）以上命令运行后，需要接收输入的内容（\*\*之间的内容），用于配置gerrit.

2）以下提到的<ip>，需要替换为本机的IP地址。

\*\*\* Gerrit Code Review 2.9.4

\*\*\*

Create '/home/git/gerrit' [Y/n]?\*回车\*

\*\*\* Git Repositories

\*\*\*

Location of Git repositories [git]: \*/home/git/repo\*

\*\*\* SQL Database

\*\*\*

Database server type [h2]:\*回车\*

\*\*\* Index

\*\*\*

Type [LUCENE/?]:\*回车\*

\*\*\* User Authentication

\*\*\*

Authentication method [OPENID/?]: \*ldap\*

LDAP server [ldap://localhost]: \*dev.svvd.com\*

Use SSL [y/N]? \*n\*

LDAP username :

Account BaseDN [DC=svvd,DC=com]:\*回车\*

Group BaseDN [DC=svvd,DC=com]:\*回车\*

\*\*\* Review Labels

\*\*\*

Install Verified label [y/N]?\*回车\*

\*\*\* Email Delivery

\*\*\*

SMTP server hostname [localhost]: \*10.1.71.244\*

SMTP server port [(default)]: \*25\*

SMTP encryption [NONE/?]: \*none\*

SMTP username [git]:\*回车\*

git's password :\*回车\*

confirm password :\*回车\*

\*\*\* Container Process

\*\*\*

Run as [git]: \*gerrit\*

Java runtime [/usr/lib/jvm/java-7-openjdk-amd64/jre]:\*回车\*

Copy gerrit.war to /home/git/gerrit/bin/gerrit.war [Y/n]?\* y\*

Copying gerrit.war to /home/git/gerrit/bin/gerrit.war

\*\*\* SSH Daemon

\*\*\*

Listen on address [\*]:\*回车\*

Listen on port [29418]:\*回车\*

Gerrit Code Review is not shipped with Bouncy Castle Crypto SSL v149

If available, Gerrit can take advantage of features

in the library, but will also function without it.

Download and install it now [Y/n]? n

Generating SSH host key ... rsa(simple)... done

\*\*\* HTTP Daemon

\*\*\*

Behind reverse proxy [y/N]?\*回车\*

Use SSL (https://) [y/N]? \*n\*

Listen on address [\*]:\*<ip>\*

Listen on port [8080]: \*8081\*

Canonical URL [http://<ip>:8081/]: \*<ip>:8081/gerrit/\*

\*\*\* Plugins

\*\*\*

Install plugin commit-message-length-validator version v2.9.4 [y/N]?\*回车\*

Install plugin download-commands version v2.9.4 [y/N]?\*回车\*

Install plugin replication version v2.9.4 [y/N]?\*回车\*

Install plugin reviewnotes version v2.9.4 [y/N]?\*回车\*

Install plugin singleusergroup version v2.9.4 [y/N]?\*回车\*

Initialized /home/git/gerrit

Executing /home/git/gerrit/bin/gerrit.sh start

Starting Gerrit Code Review: OK

5.修改gerrit配置文件

1)附件gerrit.config，拷贝到～/gerrit/etc/目录下。

$cp gerrit.config ～/gerrit/etc/gerrit.config

2)修改配置文件。

$vim ～/gerrit/etc/gerrit.config

\* basePath = <path>

中的<path>需要配置为git代码库的路径。

\* canonicalWebUrl = http://<ip>:8081/gerrit/

和

listenUrl = proxy-http://<ip>:8081/gerrit/

中，<ip>修改为本机ip地址。

6.重新启动Gerrit

$cd ~/gerrit/bin

$./gerrrit.sh restart

7.网页浏览Gerrit，<ip>为据点服务器的ip地址。

http://<ip>:8081/gerrit/

gerrit.config 文件内容：

[gerrit]

**basePath = /home/git/repo**

**:写repo的里的 All-Projects.git**

**:找到All-Projects.git文件夹的路径例如：/home/git/repo/ All-Projects.git**

canonicalWebUrl = http://10.1.59.253:8081/gerrit/

[database]

type = h2

database = db/ReviewDB

[index]

type = LUCENE

[auth]

type = LDAP

[ldap]

server = ldap://dev.svvd.com

sslVerify = false

username = redmine@dev.svvd.com

password = svvd\_0901

accountBase = CN=Users,DC=dev,DC=svvd,DC=com

accountPattern = (&(objectClass=user)(sAMAccountName=${username}))

[sendemail]

enable = true

smtpServer = 10.1.71.244

smtpServerPort = 25

# smtpUser =

# smtpPass =

smtpEncryption = none

sslVerify = false

from = SVVD Projects Code Review <gerrit@dev.svvd.com>

[container]

user = gerrit

javaHome = /usr/lib/jvm/java-7-openjdk-amd64/jre

[sshd]

listenAddress = \*:29418

[httpd]

listenUrl = proxy-http://10.1.59.253:8081/gerrit/

[cache]

directory = cache

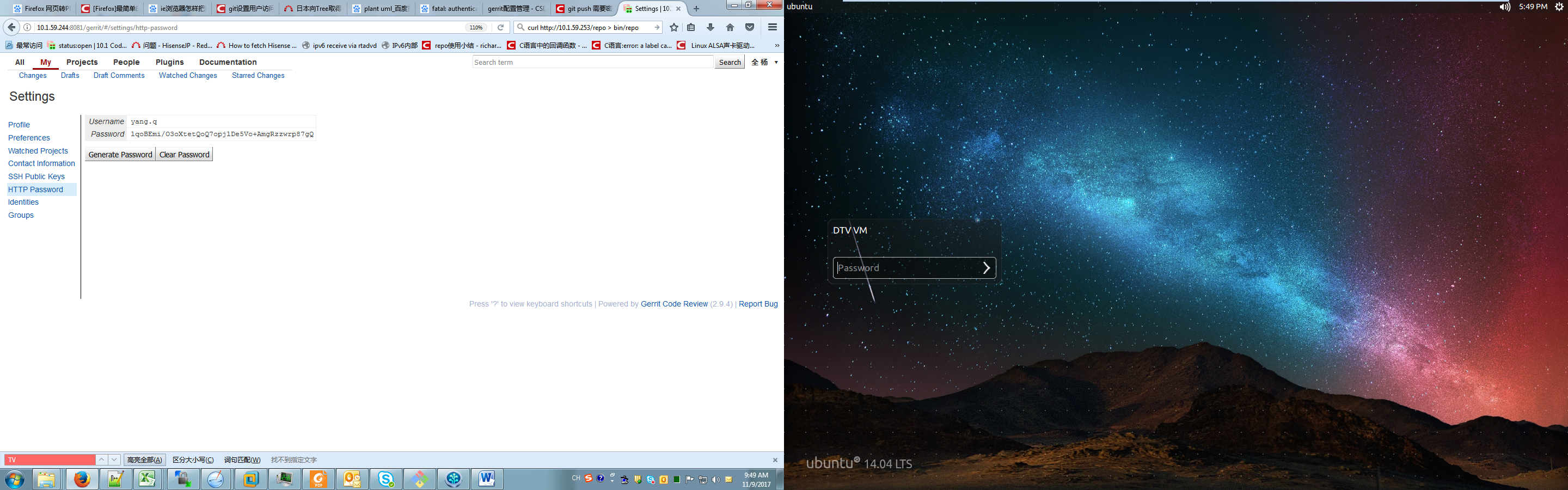
~

1. **新建远程分支和push代码向gerrit、**
   1. **Git push origin dairen 方式1**
   2. **直接在Git服务器Git branch dairen**
   3. **通过HTTP方式提交代码**

**要求输入username：**

**Password：**

**是gerrit的如下图的账号和密码**



# Gerrit与客户代码关联

1.拷贝https证书到gerrit服务器。

$cd ~

$mkdir gerrit/auth

将附件的内容解压，以下文件Copy到gerrit服务器的/home/git/gerrit/auth路径。

\* liulei.neu@neusoft.com.key

\* liulei.neu@neusoft.com.crt

\* private\_ca.crt

2.配置gerrit服务器的https证书。

$cd ~

$touch .gitconfig

$vim .gitconfig

输入以下内容：

[user]

email = <提交者的Mail>

name = <提交者>

[color]

ui = auto

[http]

sslKey = /home/git/gerrit/auth/liulei.neu@neusoft.com.key

sslCert = /home/git/gerrit/auth/liulei.neu@neusoft.com.crt

sslCaInfo = /home/git/gerrit/auth/private\_ca.crt

sslVerify = false

3.初始化repo工具

$mkdir ~/bin

$curl http://10.1.59.253/repo > ~/bin/repo

$chmod a+x ~/bin/repo

$export PATH=~/bin/:$PATH

4.设置代理服务器。

$sudo vim /etc/environment

追加以下内容：

export https\_proxy=10.1.71.246:2468

保存退出。

$source /etc/environment

5.客户代码同步

$mkdir ~/dtv13/

$cd ~/dtv13/

以下命令，因为Tree不一样，所以获取的命令也有差异：

\*hisense eu

repo init --mirror -u https://liulei.neu:liulei@sc02.socionext.com/repos/01dtvdist1/git/scsoft-165.manifests.git -m proj\_dtv13\_eu.xml

\*tks eu/tks asia

repo init --mirror -u https://liulei.neu:liulei@sc02.socionext.com/repos/01dtvdist1/git/scsoft-165.manifests.git -m proj\_dtv13.xml

\*hisense jp/sony tvp

repo init --mirror -u https://liulei.neu:liulei@sc02.socionext.com/repos/01dtvdist1/git/scsoft-165.manifests.git -m proj\_dtv13.xml

\*ap社

repo init --mirror -u https://liulei.neu:liulei@sc02.socionext.com/repos/01dtvdist1/git/scsoft-164.manifests.git -m proj\_dtv16.xml

$repo sync

**6.gerrit与代码库关联**

**$cd ~/repo**

**$ln -s ~/dtv13/ dtv 把代码路径关联到repo/dtv/ 下**

**$cd ~/gerrit/bin**

**$./gerrit.sh restart**

**打开gerrit的网页，即可看到据点服务器的代码库。**

**repo init -u http://10.1.59.244:8081/gerrit/dtv/scsoft-165.manifests -m dairen\_proj\_dtv13.xml -b dairen**

# 1-29※定时器

pause函数:

pause函数使调用进程挂起, 直到捕捉到一个信号.

pause()会令目前的进程暂停(进入睡眠状态), 直到被信号(signal)所中断.

它的原型如下:

#include <unistd.h>  
int pause();  
返回-1, 并将errno设置为EINTR.

这个函数很简单, 由字面意思就可以理解出来"暂停". pause只有在执行了一个信号处理程序并从其返回时, pause才返回

# 1-30※void \*p

#include <stdio.h>

int main()

{

void\* p;

int a;

char \*s=”sss”;

p=s;

printf(“void \*p==%d ”,\*(\*int)p,p );

return 0;

}

# 1-31※makefile 自动化变量

例如：inux里的gcc编程的makefile里$@什么的表示什么意思？

1) $@ ——目标文件的名称；

(2) $^ ——所有的依赖文件，以空格分开，不包含重复的依赖文件；

(3) $< ——第一个依赖文件的名称。

**目标**   
**$@**   
表示规则中的**目标文件集**。在模式规则中，如果有多个目标，那么，”$@”就是匹配与目标中模式定义的集合。

**$%**   
仅当目标是**函数库**文件时，”%”就是”bar.o”\*\*,”%@”就是”foo.a”。如果目标不是函数库文件，那么其值为空。

**$<**   
**依赖目标**中的第一个目标名字。如果依赖目标是以模式（即”%”）定义的，那么”%<”将是符合模式的一系列的文件集。注意，其是一个个取出来的。

**依赖目标**   
**$?**   
所有**比目标新**的依赖目标的集合。以空格分隔。

**$^**   
所有的依赖目标的集合。以空格分隔。如果在依赖目标中有多个重复的，那么这个变量会取出重复的依赖目标，只保留一份。

**$+**   
这个变量很像”$^”，也是所有依赖目标的几何。只是它不去除重复的依赖目标。

# 1-32※sizeof()字节长度

**32位编译器**

1. **char** ：1个**字节**
2. **char**\*（即指针变量）: 4个**字节**（**32**位的寻址空间是2^**32**, 即**32**个bit，也就是4个**字节**。同理**64位**编译器）
3. short int : 2个**字节**
4. int：  4个**字节**
5. unsigned int : 4个**字节**
6. float:  4个**字节**
7. double:   8个**字节**
8. **long:   4个字节**
9. long long:  8个**字节**
10. **unsigned long:  4个字节**

**64位**编译器

1. **char** ：1个**字节**
2. **char\*(即指针变量): 8个字节**
3. short int : 2个**字节**
4. int：  4个**字节**
5. unsigned int : 4个**字节**
6. float:  4个**字节**
7. double:   8个**字节**
8. **long:   8个字节**
9. long long:  8个**字节**
10. **unsigned long:  8个字节**

#include <stdio.h>

typedef struct

{

int i;

char a;

int d;

}S; //sizeof(S)= 12

union UN

{

int z;

char g;

int v[10]

}un; //sizeof(v) union 共享内存的 最大的占用最大的字节

void foo(int u[100])

{//形参 sizeof（int/char）=4

printf("sizeof(char u[100])=[%d]\n",sizeof(u));

}

int main()

{

int \*p;

int l;

char v;

char \*q ="ssssiiiiii";

char t[10];

int e[2];

char o;

printf("sizeof(int \*p)==[%d]\n",sizeof(p)); //4

printf("sizeof(char \*q ="ssssiiiiii";)==[%d]\n",sizeof(q)); //4

printf("sizeof(char t[10])==[%d]\n",sizeof(t)); //10

printf("sizeof(int e[2])==[%d]\n",sizeof(e)); //8

printf("sizeof(char o)==[%d]\n",sizeof(o)); //1

printf("sizeof(S)==[%d]\n",sizeof(S)); //12

printf("sizeof(UN)==[%d]\n",sizeof(un)); //40

foo(e); //4

}

# 1-33※C++ 引用与引用作为函数的参数

对一个数据建立一个“引用”，他的作用是为一个变量起一个别名。这是C++对C语言的一个重要补充.

http://blog.csdn.net/chaipp0607/article/details/60151813

**引用作为函数参数**

#include <iostream>

using namespace std;

int main()

{

**void swap(int & ,int&);**

int i =3,j = 5;

swap(i,j);

cout<<"i="<<i<<endl<<"j="<<j<<endl;

getchar();

return 0;

}

void swap(int &a,int &b) //初始化引用

{

int temp;

temp = a;

a =b;

b= temp;

}

运行结果：   
i = 5   
j = 3

第13行，形参是声明的引用，注意这个引用并没有初始化，这就是上面提到的特例。而在第7行调用函数的过程中，实现了引用的初始化，这是传入的实参就是变量，而不是数值，所以**做到了真正意义上的“变量传递**”。

# 1-34※音视频解码格式

video：

视频编码的主要作用是将视频像素数据（RGB，YUV等）压缩成为视频码流，从而降低视频的数据量。如果视频不经过压缩编码的话，体积通常是非常大的，一部电影可能就要上百G的空间。视频编码是视音频技术中最重要的技术之一。视频码流的数据量占了视音频总数据量的绝大部分。高效率的视频编码在同等的码率下，可以获得更高的视频质量。

封装格式有多种：MP4，FLV，F4V等

**视频编码标准和音频编码标准是H.264和AAC**这两种标准分别是当今实际应用中编码效率最高的视频标准和音频标准。视频播放器方面，无一例外的都使用了Flash播放器。

主要视频编码一览

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 推出机构 | 推出时间 | 目前使用领域 |
| HEVC(H.265) | MPEG/ITU-T | 2013 | 研发中 |
| **H.264（主流）** | MPEG/ITU-T | 2003 | 各个领域 |
| MPEG4 | MPEG | 2001 | 不温不火 |
| MPEG2 | MPEG | 1994 | 数字电视 |
| VP9 | Google | 2013 | 研发中 |
| VP8 | Google | 2008 | 不普及 |
| VC-1 | Microsoft Inc. | 2006 | 微软平台 |

audio：

音频采样数据（PCM等）压缩成为音频码流

主要音频编码一览

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 推出机构 | 推出时间 | 目前使用领域 |
| AAC | MPEG | 1997 | 各个领域（新） |
| AC-3 | Dolby Inc. | 1992 | 电影 |
| MP3 | MPEG | 1993 | 各个领域（旧） |
| WMA | Microsoft Inc. | 1999 | 微软平台 |

# 1-35※gcc thread.c -o thread -lpthread

Error：

thread.c:(.text+0xac): undefined reference to `pthread\_create'

thread.c:(.text+0x111): undefined reference to `pthread\_join'

collect2: error: ld returned 1 exit status

编译时应加上：

gcc thread.c -o thread **-lpthread**

# 1-35※线程互斥锁 SoC：System on Chip

互斥锁的操作流程如下：

1）在访问共享资源后临界区域前，对互斥锁进行加锁。

2）在访问完成后释放互斥锁导上的锁。

3）对互斥锁进行加锁后，任何其他试图再次对互斥锁加锁的线程将会被阻塞，直到锁被释放。

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <string.h>

int g;

pthread\_mutex\_t mutex;

void print(int y)

{

pthread\_mutex\_lock(&mutex);//set mutex

g=0;

g= g+y;

printf("yang-print[%d]\n",g);

pthread\_mutex\_unlock(&mutex); //unlocked

}

void \*thrd(void \*arg)

{

int thrd\_num =(int) arg;

int delay\_time;

int count=0;

int j=0;

j++;

// printf("Thread[%d]is starting\n",thrd\_num);

for(count=0;count < 2;count++)

{

delay\_time++;

printf("Thead {%d} warking....\n",thrd\_num);

print(thrd\_num);

sleep(2\*delay\_time);

}

}

int main()

{

**pthread\_mutex\_init(&mutex, NULL); //初始化互斥锁**

int i,ret;

pthread\_t thread[5];

void \*thread\_ret;

printf("main\n");

for(i=0;i < 5;i++)

{

if(ret = pthread\_create(&thread[i],NULL,thrd,(void\*)i) != 0)

{

perror("pthread\_create");

exit(1);

}

// printf("create [%d]thread is ok\n",i );

}

for(i=5;i<0;i--)

{

if(ret = pthread\_join(thread[i],&thread\_ret) != 0)

{

perror("pthread\_join");

exit(1);

}

printf("create [%d]thread is finished\n",i );

}

**pthread\_mutex\_destroy(&mutex); //销毁互斥锁**

return 0;

}

# 1-36※算法

递归

递归就是一个函数在它的**函数体内调用它自身**。执行**递归函数将反复调用其自身**，每调用一次就进入新的一层。

**递归函数必须有结束条件**。

递归分为回推和递推两个阶段，当一直回推，直到遇到墙后返回，这个墙就是结束条件。

所以递归要有两个要素，回推墙与递推关系

计算n的阶乘

例题

猴子第一天摘下N个桃子，当时就吃了一半，还不过瘾，就又多吃了一个。第二天又将剩下的桃子吃掉一半，又多吃了一个。以后每天都吃前一天剩下的一半零一个。到第10天在想吃的时候就剩一个桃子了,问第一天共摘下来多少个桃子？并反向打印每天所剩桃子数。

#include <stdio.h>

//定义一个函数，输入第n天，返回该天剩下的桃子数

int getPeachNumber(n)

{

int num; //定义所剩桃子数

if(n==10)

{

num=1; //递归结束条件，即回推墙

return num;

}

else

{

num = (getPeachNumber(n+1) + 1) \* 2; //递推关系

printf("第%d天所剩桃子%d个\n", n, num); //天数，所剩桃子个数

}

return num;

}

int main()

{

int num = getPeachNumber(1);

printf("猴子第一天摘了:%d个桃子。\n", num);

return 0;

}

# 1-37※栈堆区别

1、栈区（stack）— 由**编译器自动分配释放** ，**存放函数的参数值，局部变量的值**等。其操作方式类似于数据结构中的栈，如果还不清楚，那么就把它想成数组，它的内存分配是连续分配的，即，所分配的内存是在一块连续的内存区域内．当我们声明变量时，那么编译器会自动接着当前栈区的结尾来分配内存。

是用户存放程序临时创建的**局部变量**，也就是说我们函数括弧“{}”中定义的变量（但不包括static声明的变量，**static意味着在数据段中存放变量**）。除此以外，在函数被调用时，其参数也会被压入发起调用的进程栈中，并且待到调用结束后，函数的返回值也会被存放回栈中。由于栈的**先进先出(**FIFO)特点，所以栈特别方便用来保存/恢复调用现场。  
2、堆区（heap） — **一般由程序员分配释放**， 若程序员不释放，程序结束时可能由OS回收 。注意它与数据结构中的堆是两回事，分配方式类似于链表，在内存中的分布不是连续的，它们是不同区域的内存块通过指针链接起来的．一旦某一节点从链中断开，我们要人为的把所断开的节点从内存中释放。   
3、全局区（静态区）（static）—，**全局变量和静态变量**的存储是放在一块的，初始化的全局变量和静态变量在一块区域， 未初始化的全局变量和未初始化的静态变量在相邻的另一块区域。 - 程序结束后有系统释放   
4、文字常量区 —常量字符串就是放在这里的。 程序结束后由系统释放   
5、程序代码区—存放函数体的二进制代码。

int a = 0; 全局初始化区

char \*p1; 全局未初始化区

main()

{

int b;// 栈

char s[] = "abc"; //栈

char \*p2; //栈

char \*p3 = "123456"; 123456/0";//在常量区，p3在栈上。

static int c =0； //全局（静态）初始化区

p1 = (char \*)malloc(10);

p2 = (char \*)malloc(20);

//分配得来得10和20字节的区域就在堆区。

strcpy(p1, "123456"); //123456/0放在常量区，编译器可能会将它与p3所指向的"123456"优化成一个地方。

}

**bss段（bss segment）通常是指用来存放程序中未初始化的全局变量的一块内存区域。**

**数据段（data segment）通常是指用来存放程序中已初始化的全局变量的一块内存区域, bss段/数据段属于静态内存分配。**

# 1-38※linux静态库和动态库

linux下文件的类型是不依赖于其后缀名的，但一般来讲：

**.o,是目标文件,相当于windows中的.obj文件**

**.so 为共享库,是shared object,用于动态连接的,和dll差不多**

**.a为静态库,是好多个.o合在一起,用于静态连接**

**.la为libtool自动生成的一些共享库，vi编辑查看，主要记录了一些配置信息**。可以用如下命令查看\*.la文件的格式 $file \*.la

\*.la:ASCII English text

Linux下有静态链接库和动态链接库两种，**静态链接库的后缀名是.a，动态链接库的后缀名是.so**，

**静态链接库是指程序在编译链接的过程中就把依赖的库插入到程序中，这样程序的运行就不需要这些库的支持，但缺点是文件变大，下面以一个例子介绍如何制作静态链接库：**

**add.c**

**int my\_add(int num1, int num2)**

**{**

**return num1 + num2;**

**}**

**mul.c**

**int my\_mul(int num1, int num2)**

**{**

**return num1 \* num2;**

**}**

**computer.h**

**int my\_add(int num1, int num2);**

**int my\_mul(int num1, int num2);**

**main.c**

**#include <stdio.h>**

**#include "compute.h"**

**int main()**

**{**

**int a=3, b=5;**

**printf("%d + %d = %d\n", a, b, my\_add(a, b));**

**printf("%d \* %d = %d\n", a, b, my\_mul(a, b));**

**return 0;**

**}**

**add.c**和**mul.c**编译成静态链接库：

 gcc -o add.a -c add.c -static

 gcc -o mul.a -c mul.c -static

 **ar -rc** libcompute.a add.a mul.a

**上面首先把add.c和mul.c编译成两个独立的.a文件，然后把他们合并成一个静态链接库libcompute.a（当然你也可以把它们做成两个静态链接库，连接main的时候指定这两个就可以了）。static表示编译成静态链接库。ar是一个打包命令**

gcc -o main main.c -lcompute -L.

**上面的-lcomputer表示要链接的静态链接库是libcomputer.a，-L.表示在当前目录下寻找。**

**动态库：**

**将这几个文件编译成一个动态库：libtest.so**

$ gcc test\_a.c test\_b.c test\_c.c **-fPIC -shared -o** libtest.so

**将test.c与动态库libtest.so链接生成执行文件test**

**$ gcc test.c -L. -ltest -o test**

最主要的是GCC命令行的一个选项:

-**shared该选项指定生成动态连接库**（让连接器生成T类型的导出符号表，有时候也生成弱连接W类型的导出符号），不用该标志外部程序无法连接。相当于一个可执行文件

l -fPIC：表示编译为位置独立的代码，不用此选项的话编译后的代码是位置相关的所以动态载入时是通过代码拷贝的方式来满足不同进程的需要，而不能达到真正代码段共享的目的。

l  **-L.：表示要连接的库在当前目录中**

l -ltest：编译器查找动态连接库时有隐含的命名规则，即在给出的名字前面加上lib，后面加上.so来确定库的名称

l LD\_LIBRARY\_PATH：这个环境变量指示动态连接器可以装载动态库的路径。

l 当然如果有root权限的话，可以修改/etc/ld.so.conf文件，然后调用/sbin/ldconfig来达到同样的目的，不过如果没有root权限，那么只能采用输出LD\_LIBRARY\_PATH的方法了。

if(a)等价于if(a!=0)

# 1-38※回调函数

概念：函数指针

回调函数是指 使用者自己定义一个函数，实现这个函数的程序内容，然后把这个函数（入口地址）作为参数传入别人（或系统）的函数中，由别人（或系统）的函数在运行时来调用的函数。函数是你实现的，但由别人（或系统）的函数在运行时通过参数传递的方式调用，这就是所谓的回调函数。简单来说，就是由**别人的函数运行期间来回调你实现的函数**。

**C**语言中一般用typedef来为回调函数定义别名（参数名）。 别名通过宏定义typedef来实现，不是简单的宏替换。可以用作同时声明指针型的多个对象。

#include <stdio.h>

void print(char\* s)

{

printf(s);

}

int main()

{

void (\*p)(char \*s);

p=print;

p("hello callback\n");

return 0;

}

#include<stdio.h>

//方法指针的格式为：int (\*ptr)(char \*p) 即：返回值(指针名)(参数列表)  
typedef int (\*CallBackFun)(char \*p);    //为回调函数命名，类型命名为 CallBackFun，参数为char \*p

//方法 Afun，格式符合 CallBackFun 的格式，因此可以看作是一个 CallBackFun     
int Afun(char \*p)  
{  
    printf("Afun 回调打印出字符%s!\n", p);     
    return 0;  
}

// 方法 Cfun，格式符合 CallBackFun 的格式，因此可以看作是一个 CallBackFun  
int Cfun(char \*p)  
{     
    printf("Cfun 回调打印:%s, Nice to meet you!\n", p);     
    return 0;  
}

// 执行回调函数，方式一：通过命名方式，pCallBack可以看做是CallBackFun的别名  
int call(CallBackFun pCallBack, char \*p)  
{     
    printf("call 直接打印出字符%s!\n", p);     
    pCallBack(p);     
    return 0;  
}

// 执行回调函数，方式二：直接通过方法指针      
int call2(char \*p, int (\*ptr)())  //或者是int call2(char \*p, int (\*ptr)(char \*)) 同时ptr可以任意取名  
{  
    printf("==============\n", p);      
    (\*ptr)(p);  
}

int main()  
{     
    char \*p = "hello";  
    call(Afun, p);     
    call(Cfun, p);  
    call2(p, Afun);     
    call2(p, Cfun);  
    return 0;  
}

结构解析

回调函数主要结构有三部分组成：**主函数、调用函数和被调函数（**如图1-1所示）。C语言中，被调函数通常以函数指针（指向对应函数的入口地址）的形式出现。

这里给出一个最简单的回调函数结构，并解析相关数据结构。

//定义回调函数

void PrintfText()

{

printf("Hello World!\n");

}

//定义实现回调函数的"调用函数"

void CallPrintfText(void (\*callfuct)())

{

callfuct();

}

//实现函数回调

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

{

CallPrintfText(PrintfText);

return 0;

}

**调用函数向其函数中传递 void (\*callfuct)(void) 这是一个 void callfuct(void) 函数的入口地址，即PC指针可以通过移动到该地址执行void callfuct(void) 函数，可以通过类比数组来理解。**

实现函数调用中，函数调用了“调用函数”，再在其中进一步调用被“调用函数”。相比于主函数直接调用“被调函数”，这种方法为使用者，而不是开发者提供了灵活的接口。另外，函数入口可以像变量一样设定同样为开发者提供了灵活性。

include <iostream>

#include <string>

using namespace std;

**typedef void (\*FP)(char\* s); //表示函数指针**

void f1(char\* s){cout<<s;}

void f2(char\* s){cout<<s;}

void f3(char\* s){cout<<s;}

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

{

int funcselector=0; //定义一个整数用于控制待执行的函数

void\* a[]={f1,f2,f3}; //定义了**指针数组**，这里a是一个普通指针

a[0]("Hello World!\n"); //编译错误，指针数组不能用下标的方式来调用函数

FP f[]={f1,f2,f3}; //定义一个函数指针的数组，这里的f是一个函数指针

/\* Handle of funselector \*/ //此处用于处理funselector，控制待执行的函数

f[funselector]("Hello World!\n"); //正确，函数指针的数组进行下标操作可以进行函数的间接调用

return 0;

}

kill -s 9 4803 client // kill -9 222  
http://blog.csdn.net/journey0804/article/details/69376822?locationNum=3&fps=1

# 1-39※sem\_wait sem\_post信号量操作进本函数

sem\_post函数的作用是给信号量的值加上一个“1”，它是一个“原子操作”－－－即同时对同一个信号量做加“1”操作的两个线程是不会冲突的；而同时对同一个文件进行读、加和写操作的两个程序就有可能会引起冲突。**信号量的值永远会正确地加一个“2”－－因为有两个线程试图改变它**。  
 sem\_wait函数也是一个原子操作，它的作用是从信号量的值减去一个“1”，但它永远会先等待该信号量为一个非零值才开始做减法。也就是说，如果你对一个值为2的信号量调用sem\_wait(),线程将会继续执行，介信号量的值将减到1。如果对一个值为0的信号量调用sem\_wait()，这个函数就会地等待直到有其它线程增加了这个值使它不再是0为止。如果有两个线程都在sem\_wait()中等待同一个信号量变成非零值，那么当它被第三个线程增加一个“1”时，等待线程中只有一个能够对信号量做减法并继续执行，**另一个还将处于等待状态。**  
 信号量这种“只用一个函数就能原子化地测试和设置”的能力下正是它的价值所在。还有另外一个信号量函数sem\_trywait，它是sem\_wait的非阻塞搭档。

1. #include <stdio.h>
2. #include <unistd.h>
3. #include <stdlib.h>
4. #include <string.h>
5. #include <pthread.h>
6. #include <semaphore.h>
8. sem\_t bin\_sem;
9. **void** \*thread\_function1(**void** \*arg)
10. {
11. printf("thread\_function1--------------sem\_wait\n");
12. sem\_wait(&bin\_sem);
13. printf("sem\_wait\n");
14. **while** (1)
15. {
16. }
17. }
19. **void** \*thread\_function2(**void** \*arg)
20. {
21. printf("thread\_function2--------------sem\_post\n");
22. sem\_post(&bin\_sem);
23. printf("sem\_post\n");
24. **while** (1)
25. {
26. }
27. }


31. **int** main()
32. {
33. **int** res;
34. pthread\_t a\_thread;
35. **void** \*thread\_result;
37. res = sem\_init(&bin\_sem, 0, 0);
38. **if** (res != 0)
39. {
40. perror("Semaphore initialization failed");
41. }
42. printf("sem\_init\n");
43. res = pthread\_create(&a\_thread, NULL, thread\_function1, NULL);
44. **if** (res != 0)
45. {
46. perror("Thread creation failure");
47. }
48. printf("thread\_function1\n");
49. sleep (5);
50. printf("sleep\n");
51. res = pthread\_create(&a\_thread, NULL, thread\_function2, NULL);
52. **if** (res != 0)
53. {
54. perror("Thread creation failure");
55. }
56. **while** (1)
57. {
58. }
59. }

62. sem\_init
63. thread\_function1
64. thread\_function1--------------sem\_wait
65. sleep
66. thread\_function2--------------sem\_post
67. sem\_wait
68. sem\_post
69. **//不加sem\_post 控制**
70. sem\_init
71. thread\_function1
72. thread\_function1--------------sem\_wait
73. sem\_wait
74. sleep
75. thread\_function2--------------sem\_post
76. sem\_post

# 1-39※解压rar

rar x project.rar

# 1-39※c++ 常见知识点

# **1.基类采用virtual虚析构函数:**

# 直接的讲，C++中基类采用virtual虚析构函数是为了防止内存泄漏。具体地说，如果派生类中申请了内存空间，并在其析构函数中对这些内存空间进行释放。

# 假设基类中采用的是非虚析构函数，当删除基类指针指向的派生类对象时就不会触发动态绑定，因而只会调用基类的析构函数，而不会**调用派生类的析构函数**。

# 那么在这种情况下，派生类中申请的空间就得不到释放从而产生内存泄漏。所以，为了防止这种情况的发生，C++中基类的析构函数应采用virtual虚析构函数。

# **2.virtual 继承问题：**

# 在多继承下，虚继承就是为了解决菱形继承中，B,C都继承了A，D继承了B,C，那么D关于 A的引用只有一次，而不是 普通继承的 对于A引用了两次

# 3. **virtual 应用场景**

# 当我们一个指针指向了子类，或者引用子类那么我们希望调用函数的时候是调用子类的函数，

# 但是实际情况却不是这样。这时候就需要virtual关键字了

# 4.**抽象类**

# 纯虚函数： 含有 它的类叫抽象类 ，不能被实例化 即new()

# 纯虚函数子类必须实现它

# 5.**虚函数底层实现**

# 实现原理：虚函数表+虚表指针

# 关键字：虚函数底层实现机制；虚函数表；虚表指针

# 编译器处理虚函数的方法是：为每个类对象添加一个隐藏成员，隐藏成员中保存了一个指向函数地址数组的指针，称为虚表指针（vptr），这种数组成为虚函数表（virtual function table, vtbl），即，每个类使用一个虚函数表，每个类对象用一个虚表指针。

# 举个例子：基类对象包含一个虚表指针，指向基类中所有虚函数的地址表。派生类对象也将包含一个虚表指针，指向派生类虚函数表。看下面两种情况：

# 如果派生类重写了基类的虚方法，该派生类虚函数表将保存重写的虚函数的地址，而不是基类的虚函数地址。

# 如果基类中的虚方法没有在派生类中重写，那么派生类将继承基类中的虚方法，而且派生类中虚函数表将保存基类中未被重写的虚函数的地址。注意，如果派生类中定义了新的虚方法，则该虚函数的地址也将被添加到派生类虚函数表中。

1.const :

const 与 #define 区别

1)const 有类型检测 后者没有

2)有些集成化的调试工具可以对 const 常量进行调试,但是不能对宏常量进行调试。

2.类中的const

1)企图在类声明中初始化 const 数据成员 错误

2)数据成员的初始化只能在类构造函数的初始化表中进行 A::A(int size) : SIZE(size) // 构造函数的初始化表

3)整个类中都恒定的常量呢?别指望 const 数据成员了,应该用类中 的枚举常量来实现：enum { SIZE1 = 100, SIZE2 = 200};

枚举常量不会占用对象的存储空间,它们在编译时被全部求值。枚举常量的缺点是: 它的隐含数据类型是整数,其最大值有限,且不能表示浮点数(如 PI=3.14159)。

return int(x + y); // 创建一个临时变量并返回它

应该：

int temp = x + y;

return temp;

引用：

(1)引用被创建的同时必须被初始化(指针则可以在任何时候被初始化)。

(2)不能有 NULL 引用,引用必须与合法的存储单元关联(指针则可以是 NULL)。

(3)一旦引用被初始化,就不能改变引用的关系(指针则可以随时改变所指的对象)

注意 当 数组 作为 函 数的 参数 进 行传 递时 , 该数 组自 动 退化 为同 类 型的 指针

# 1-40※c++ 内存思考