# GUN编译工具GCC

## 1.编译

GCC可以编译C,C++,Object C,Java等语言，可以在多种平台上编译出可执行文件

## 2.GCC编译过程

a.预处理

b.编译

c.汇编

d.链接

GCC的编译过程可以控制，优化

gcc -v ：查看gcc版本

## 3.相关文件后缀名

|  |  |
| --- | --- |
| 扩展名 | 说明 |
| .a | 静态对象库 |
| .c | 需要预处理的c语言源代码 |
| .h | c语言源代码头文件 |
| .i | 不需要预处理的c语言源代码 |
| .o | 目标文件 |
| .s | 汇编语言代码 |
| .so | 共享对象库 |

## 4.编译单源程序

### 4.1 语法

gcc [选项参数] c文件

### 4.2 通用输出文件名

-o 指定输出文件

例：gcc -o main ch01.c

-pedantic 检测不符合ANSI/ISO C语言标准的源代码，使用扩展语法

的地方将产生警告信息

-Wall 产生尽可能多的警告信息

-Werror 要求编译器将错误当做错误进行处理

-x 指定编译代码类型，c,c++,assmebler,none。None参数将

将根据扩展名自动确认

例：gcc -x c -Wall -o main ch01.c

生成调试信息与优化

-g 生成调试信息

-O 优化

## 5 编译多源文件

语法: gcc [选项] c源代码1 c源代码2 c源代码3

## 6.预处理指令

|  |  |
| --- | --- |
| 预编译指示符号 | 说明 |
| #define | 定义宏 |
| #elif | else if多选分支 |
| #else | 与#if,#ifndef,#ifdef结合使用 |
| #if | 判定 |
| #endif | 结束判定 |
| #ifdef | 判定宏是否定义 |
| #ifndef | 判定宏是否定义 |
| #include | 将指定的文件插入#include的位置 |
| #include\_next | 与include一样，但是从当前目录之后的目录查找 |
| #line | 指定行号 |
| #pragma | 提供额外的信息的标准方法，可用来指定平台 |
| #undef | 删除宏 |
| #warning | 创建一个警告 |
| ## | 连接操作符号，用于宏内连接两个字符串 |

#warning,#error

1. #include <stdio.h>
2. #if (VERSION<3)//如果VERSION<3
3. #error "版本太低" //产生错误
4. #elif(VERSION>3)//如果VERSION>3
5. #warning "版本太高" //产生警告
6. #endif
7. int main()
8. {
9. printf("版本：%d\n",VERSION);
10. return 0;
11. }
13. /\*
14. 使用gcc wraning\_error.c -o main -DVERSION=2编译，即传入VERSION的值为2候会产
15. 生错误，并提示版本太低并终止编译
16. 使用gcc wraning\_error.c -o main -DVERSION=4编译，即传入VERSION的值为4候会产
17. 生警告，并提示版本太高
19. 使用gcc wraning\_error.c -o main -DVERSION=3编译，输出版本:3

22. \*/

#line

1. #include "stdio.h"
2. int main()
3. {
4. printf("%d\n",\_\_LINE\_\_);
5. #line 100
7. printf("%d\n",\_\_LINE\_\_);
8. return 0;
9. }
10. /\*\*
11. 预编译后，被替换为如下的形式，\_\_LINE\_\_表示在源代码中的行数，因为#line指定当前行为100所以下面的行为101
12. int main()
13. {
14. printf("%d\n",5);
15. # 100 "line.c"
16. printf("%d\n",101);
17. return 0;
18. }
19. \*/

#include,#include\_next

系统头文件使用 #include <…>

用户定义的头文件使用#include"…"

规则：系统头文件会在I参数指定的目录中寻找

用户头文件会在当前目录文件中寻找

编译C++优先查找/usr/include/g++

#include<sys/time.h>会在所有标准目录的子目录sys中查找time.h

#include的文件名含扩展“，\*.?无意义”，除非文件名中包含\*

#paragma

所有GCC的pragma都定义两个词GCC+其他

#pragma GCC dependency 文件名 提示符号

测试文件的时间戳，当定义文件比当前文件新的时候产生警告

#pragma GCC poison 每次使用指定的名字就产生警告

1. #include <stdio.h>
2. #pragma GCC dependency "dep.c"
3. #pragma GCC poison abc
4. int main()
5. {
6. int abc = 0;
7. printf("%d\n",abc);
8. return 0;
9. }
10. /\*
11. 当创建两次dep.c文件后，编译后提示：
12. warning: current file is older than dep.c
13. #pragma GCC dependency "dep.c"
14. 这个也就是比较两个文件的最后保存时间
15. 如果后面文件的最后修改时间比当前文件的晚，就会产生一个警告。当前文件依赖于 dependency后面文件，dependency后面文件的最后修改时间比当前文件的晚，就会产生一个警告。
16. 另外 #pragma GCC posion 单词
17. 把后面的单词声明为毒药，不能被使用，否则产生错误,如这个程序将报错
18. .c:7:7: error: attempt to use poisoned "abc"
19. int abc = 0;
20. #pragma pack n 指定对齐补齐字节数
21. \*/

## 7.预定宏

|  |  |
| --- | --- |
| 宏 | 说明 |
| \_BASE\_FILE\_ | 源代码的完整路径 |
| \_cplusplus | C++有效，程序不符合标准为1，否则是标准的年月 |
| \_DATE | 日期 |
| \_FILE\_ | 源代码的文件名 |
| \_func\_ | 当前函数名 |
| \_FUNCTION\_ | 同上 |
| \_INCLUDE\_LEVEL\_ | 包含层数，基本的为0 |
| \_LINE\_ | 行数 |
| \_TIME\_ | 时间 |

prin.h

1 #include <stdio.h>

2 void print(void){

3 printf("\_\_BASE\_FILE\_\_:%s\n",\_\_BASE\_FILE\_\_);

4 printf("\_\_FILE\_\_:%s\n",\_\_FILE\_\_);

5 printf("\_\_LINE\_\_:%d\n",\_\_LINE\_\_);

6 printf("\_\_FUNCTION\_\_:%s\n",\_\_FUNCTION\_\_);

7 printf("\_\_func\_\_:%s\n",\_\_func\_\_);

8

9 printf("\_\_DATE\_\_:%s\n",\_\_DATE\_\_);

10 printf("\_\_TIME\_\_:%s\n",\_\_TIME\_\_);

11 printf("\_\_INCLUDE\_LEVEL\_\_:%d\n",\_\_INCLUDE\_LEVEL\_\_);

12 #ifdef \_\_cplusplus

13

14 printf("\_\_cplusplus:%d\n",\_\_cplusplus);

15 #endif

16

17

18

19 }

1#include"Prin.h"

2

3 int main()

4 {

5 print();

6

7 return 0;

8

9 }

10 /\*

11 输出结果

12 \_\_BASE\_FILE\_\_:Prin.c

13 \_\_FILE\_\_:Prin.h

14 \_\_LINE\_\_:5

15 \_\_FUNCTION\_\_:print

16 \_\_func\_\_:print

17 \_\_DATE\_\_:Mar 13 2018

18 \_\_TIME\_\_:16:33:17

19 \_\_INCLUDE\_LEVEL\_\_:1

## 8.编译环境变量

C\_INCLUDE\_PATH: 查找头文件的目录

CPATH 查找头文件，相当于-I选项

CPLUS\_INCLUDE\_PATH 查找头文件的目录，C++

LD\_LIBRARY\_PATH 编译没有影响，主要是运行，指定目录便于定位

共享库

LIBRARY\_PATH 查找连接文件，相当于-I选项

## 9.生成汇编

编译成汇编

gcc -S ch01.c ch01\_1.c

编译汇编

gcc ch01.s ch01\_1.s -o main

# 静态库

静态库在编译连接时，就把库里的代码提取出来放入可执行文件中，程序在执行时不再依赖库。

## 1.编译静态库

gcc -c -static ch01\_1.c

其中-static可选，可阻止gcc使用共享库

不使用共享库会导致可执行文件变大，但会减少运行时间开销

## 2.ar指令

ar -r libmy.a ch01\_1.o

语法：ar [选项] 归档文件名 目标文件列表

指令ar的常用选项

|  |  |
| --- | --- |
| 选项 | 说明 |
| -d | 从归档文件删除指定目标文件列表 |
| -q | 将指定目标文件快速附加到归档文件末尾 |
| -r | 将指定目标文件插入文档，如果存在则更新 |
| -t | 显示目标文件列表 |
| -x | 把归档文件展开为目标文件 |

## 2.使用静态库

gcc -o main ch01.c libmy.a

如果libmy.a在LIBRARY\_PATH的指定目录中，还可以按如下方式编译

gcc ch01.c -o main -lmy

## 3.总结

如何使用静态库

1）写源程序，编译生成.o文件

2）和静态库文件链接

gcc xxx.o libxxx.a

gcc xxx.o -lxxx (-l 后是静态库的库名，不文件名)

3）根据库名寻找文件

根据环境变量LIBRARY\_PATH去匹配库文件

如math文件就匹配libmath.a

如果没有环境就是LIBRARY\_PATH,也可以指定查找库文件的

路径 gcc xxx.o -o -lxxx -L 库文件所在目录

Calc.h

1 #ifndef \_CALC\_H\_

2 #define \_CALC\_H\_

3

4 int add(int a,int b);

5 int sub(int a,int b);

6

7 #endif

8

Calc.c

1 #include "Calc.h"

2

3 int add(int a,int b){

4

5 return a+b;

6

7 }

8

9 int sub(int a,int b){

10 return a-b;

11 }

12

Show.h

1 #ifndef \_SHOW\_H\_

2 #define \_SHOW\_H\_

3

4 void show(int a,char op,int b, int c);

5

6 #endif

Show.c

1 #include "stdio.h"

2 #include "show.h"

3

4 void show(int a,char op,int b,int c)

5 {

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

7

8

9 }

main.c

1 #include "calc.h"

2 #include "show.h"

3

4 int main()

5 {

6 show(30,'+',20,add(30,20));

7 show(30,'-',20,sub(30,20));

8

9 return 0;

10

11 }

12

13 /\*

14 首先得到编译得到calc.o show.o两个文件

15 然后用ar -r libmath.a calc.o show.o将两个文件打包成静态库

16 最后gcc main.c libmath.a编译链接静态库

17

18 调用静态库的方法还可以是：

19 gcc main.c -lmath -L.(-L.指定优先寻找当前文件夹)

20 或gcc main.c -lmath (静态库必须包含在可寻找的目录下)

21 \*/

# 共享库

动态库在编译连接时只记录了要用的名字在哪个库文件中，运行时才从库文件中找到这个名字(函数或者变量)边访问它。

## 1.创建和编译共享库

### 编译共享库有两个过程：

1.编译成未知独立代码的目标文件，如-fpic

2.编译成共享库，选项-shared

如：gcc -c -fpic ch01\_1.c

gcc -shared ch01\_1.o -o libmy.so

也可以按如下方式编译

gcc -fpic -shared ch01\_1.c -o libmy.so

## 2.定位共享库

共享库编译的时候与静态库一样依赖LIBRARAY\_PATH,运行的时候依赖

LD\_LIBRARY\_PATH

规则：

查找LD\_LIBRARY\_PATH,目录使用冒号分隔

查找/etc/ld.so.cache中找到的列表，工具ldconfig维护

查找目录 /lib

查找目录 /usr/lib

## 3.使用共享库

gcc ch01.c libmy.so -o main

## 4.共享库的四个函数

## #include <dlfcn.h>

void \*dlopen(const char\* filename,int flag)//打开动态库

filename:如只给共享库文件名，则通过LD\_LIBRARY\_PATH环境变量

搜索共享库，如给共享库路径，则按照路径加载，不使用环境

变量。

flag: RTLD\_LAZY 延迟加载，什么时候使用共享库，什么时候载入

RTLD\_NOW 立即加载

成功返回共享库句柄，失败返回NULL

char\* dlerror(void);//获取错误信息

返回错误信息字符串，没有错误信息返回NULL

void\* dlsym(void\* handle,const char\* symbol);//查找动态库中的函数并返回调用地址

handle：共享库句柄

const char\* symbol;//函数名

成功返回函数地址，失败返回NULL

int dlclose(void\* handle);//关闭动态库

成功返回0，不成功返回非零

# 内存管理

## 1.内存管理的形式

用户层

STL: 内存分配器

C++: new ,delete(构造，析构)

C: malloc/calloc/realloc/free

POSIX: brk/sbrk

Linux: mmap/munmap;

系统层

内核 kmalloc/vmalloc

驱动 get\_free\_page

硬件实现

## 2.进程映像

2.1 程序是保存在磁盘上的可执行文件

如，a.out ls gcc qq.exe

2.2 运行程序程序时，需要把磁盘上的可执行文件，加载到内存中，形成

进程

2.3 一个程序文件可以同时存在多个进程

2.4 进程在内存空间中的布局就是进程映像，从低地址到高地址依次是

代码区：可执行指令，字面值常量，具有常属性且初始化的全局

和静态变量

数据区：不具有常属性且初始化的全局和静态变量

BSS区：未初始化的全局和静态变量，进程一旦加载这个区域就

清零

堆区 ：动态内存分配

栈区 ：非静态局部变量

命令行参数和环境变量区：

在堆区和栈区之间会留有一段空隙，一方面为堆和栈的增长

预留空间，同时共享库和共享内存也会占用这个区域

局部变量在栈区，malloc在堆区，函数代码在代码区，全局变量和静态变量（赋值的在数据区，没有赋值的在BSS区），100,3.14这些数值常量在代码区存放，字面串字面量它在代码区的高地址部分单独开一个块空间存放，以地址的形式引用。const修饰的都是常属性。

1 #include "stdio.h"

2 #include "stdlib.h"

3 #include "unistd.h"

4

5 const int const\_global = 10;//常全句变量

6 int init\_global = 10;

7 int uninit\_global;//未初始化全局变量

8

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

10 {

11 //常静态变量

12 const static int const\_static = 10;

13 static int init\_static = 10;//初始化静态变量

14 static int uninit\_static;//未初始化静态变量

15

16 const int const\_local = 10;

17 int prev\_local ;//前局部变量

18 int next\_local;//后局部变量

19

20 //前堆变量

21 int\* prev\_heap = malloc(sizeof(int));

22 //后堆变量

23 int\* next\_heap = malloc(sizeof(int));

24

25 const char\* literal = "literal";//字面值常量

26 extern char\*\* environ;//环境变量

27

28 printf("-----命令行参数与环境变量-----------\n");

29 printf(" 环境变量: %p\n",environ);

30 printf(" 命令行参数: %p\n",argv);

31

32

33 printf("-------------栈--------------------\n");

34 printf(" 常局部变量: %p\n",&const\_local);

35 printf(" 前局部变量: %p\n",&prev\_local);

36 printf(" 后局部变量: %p\n",&next\_local);

37

38

39 printf("-------------堆-------------------\n");

40

41 printf(" 前堆变量: %p\n",&prev\_heap);

42 printf(" 后堆变量: %p\n",&next\_heap);

43

44 printf("-------------BSS-------------------\n");

45 printf(" 未初始化全局变量: %p\n",&uninit\_global);

46 printf(" 未初始化静态变量: %p\n",&uninit\_static);

47

48

49 printf("--------------数据区------------------\n");

50 printf(" 初始化静态变量: %p\n",&init\_static);

51 printf(" 初始化全局变量: %p\n",&init\_global);

52

53

54 printf("--------------代码------------------\n");

55 printf(" 常静态变量: %p\n",&const\_static);

56 printf(" 字面值常量: %p\n",&literal);

57 printf(" 常全局变量: %p\n",&const\_global);

58 printf(" 函数: %p\n",main);

59

60 printf("PID = %d\n",getpid());

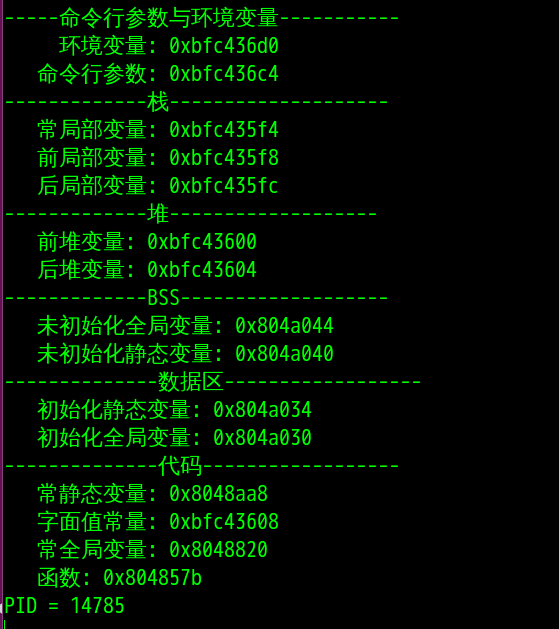
61 getchar();

62

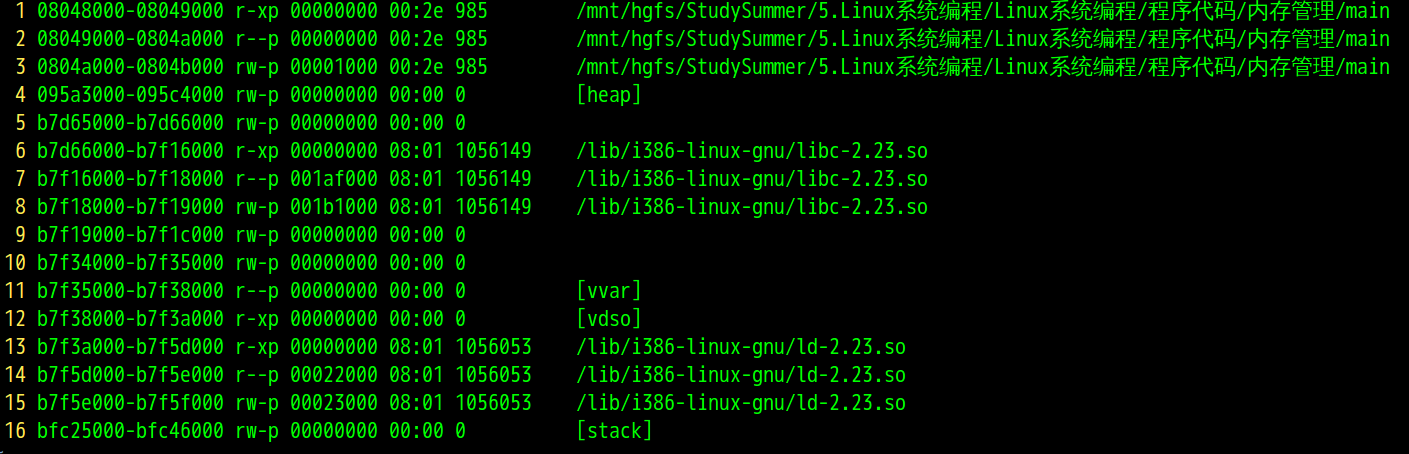
63 return 0;

64 }

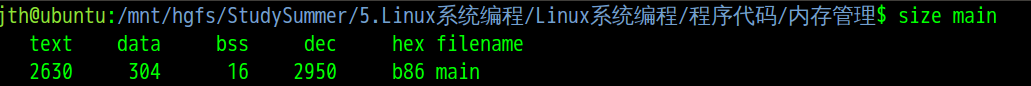
输出：



进程的堆栈信息：



查看对应的数据分区的大小



size命令

显示一个目标文件或者链接库文件中的目标文件的各个段的大小

其中 text表示正文段大小，data表示包含静态变量和已经初始化（可执行文件包含了初始化的值）的全局变量的数据段大小，bss由可执行文件中不含其初始化值的全局变量组成。

# 虚拟内存

## 1.每个进程都有各自独立的4G字节的虚拟地址空间

2.用户程序中使用的虚拟地址空间中的地址，永远无法直接访问实际的物理内存地址

3.虚拟内存到物理内存的映射由操作系统动态维护

4.虚拟内存一方面保护了操作系统的安全，另一方面允许应用程序使用比实际物理内存更大的地址空间

5.4G的进程空间分为两部分，0~3G为用户空间，3G~4G为内核空间

6.用户空间中代码不能直接访问内核空间中的代码和数据，但是可以通过系统调用进入内核状态，间接与内核交互

7.对内存的越权访问，或访问未建立映射的虚拟内存，将会导致段错误

int \*p;

\*p = 100;

这样就会出现段错误，因为未和内存映射，不知道这个p是指向何处。

int a;//建立映射关系

int \*p = &a;

\*p = 100;

只有映射后才能实际操作这个地址。

进程中需要使用的地址首先映射到内存中，如果一段时间后这些内存暂时不再使用，cpu检测到，然后换页，把这块东西绑定到你的硬盘上去。每个进程都有一个映射表，记录映射的关系。

8.用户空间对应进程，进程一旦切换，用户空间马上就变化。内核空间由操作系统内核使用，不会随进程切换而变化。内核空间由内核根据独立且唯一的页表

init\_mm.pgd进行映射，而用户空间的页表则是每个进程都有一份

9.每个进程的内存空间完全独立，因此在不同进程之间交换虚拟地址毫无意义。

映射以页为单位，1页是4096个字节

10.标准库的内存分配函数

（malloc/calloc/realloc）需要用一套数据结构维护动态分配的内存，因此会分配比实际要求的内存多12字节的内存，用户存储某些控制信息，该信息一旦被破坏，将导致后续操作（如free）也出现异常

例：

1 #include "stdio.h"

2 #include "stdlib.h"

3

4 int main()

5 {

6 int\* p1 = malloc(sizeof(int));

7 int\* p2 = malloc(sizeof(int));

8

9 printf("p1 = %p,p2 = %p\n",p1,p2);

10 free(p2);

11

12 return 0;

13 }

1 #include "stdio.h"

2 #include "stdlib.h"

3 int main(void)

4 {

5 int\* p1 = malloc(sizeof(int));

6 int\* p2= malloc(sizeof(int));

7

8 printf("%p ,%p\n",p1,p2);

9 free(p2);

10 p1[0]=0;//p1[3] = 0;//因为p1[3]是未分配的内存，即未映射> ，所以也不能正常使用

11 free(p1);

12

13 return 0;

14

15 }



这是堆空间，即21000（十六进制）字节=135168字节=4096\*33字节=33页



其中的p1[3]访问到控制信息，所有会报错

11.虚拟内存到物理内存的映射以页为单位(4096字节).通过malloc函数首次分配内存，至少映射33页。通过free函数，释放到全部动态分配的内存，最初从33页依然保存，直到进程退出。

# 内存管理APIS

#include<unistd.h>

void\* sbrk(intptr\_t increment//内存增量(字节)

)

返回上次调用sbrk/brk函数后的末尾指针，失败返回-1

increment取值：0 获取末尾指针

>0 增加内存空间

<0 释放内存空间

它是控制内部维护的一个指针，这个指针指向当前堆内存最后一个字节的下一个位置。

sbrk函数根据增量参数调整该指针的位置，同时返回该指针被调整前的位置，若发现内存页耗尽或空闲，则自动追加或解除页映射。负值是释放内存。

它重新指定了数据段的结束位置

#include<unistd.h>

int brk(

void\* end\_data\_segment//内存块尾地址

)

成功返回0，失败返回-1

它内部维护一个指针，指向当前堆内存最后一个字节的下一个位置。brk函数根据指针参数设置该指针的位置，若发现页耗尽或空闲。则自动追加或解除页映射。

sbrk/brk底层维护一个指针位置，以页(4k)为单位映射和解映射内存，并根据参数调整所维护的指针位置。一般使用sbrk分配内存，brk释放内存。

练习：使用brk/sbrk实现一个堆栈数据结构，存放整数。利用该堆栈实现一个任意进制打印整数的函数。

1 #include "stdio.h"

2 #include "unistd.h"

3

4 void\* g\_begin = NULL;

5 //初始化

6 void init()

7 {

8 g\_begin = sbrk(0);

9 }

10

11 void deinit()

12 {

13 brk(g\_begin);

14 }

15 //入栈

16 void push(int data)

17 {

18 \*(int\*)sbrk(sizeof(int))= data;

19 }

20 //出栈

21 void pop()

22 {

23 sbrk(-sizeof(int));

24 }

25 //栈顶

26 int top()

27 {

28 return \*((int\*)sbrk(0) - 1);

29 }

30 //判断是否为空

31 int empty()

32 {

33 return sbrk(0) == g\_begin;

34

35 }

36 //任意进制打印

37 void printb(unsigned int data,int base)

38 {

39 init();

40

41 do{

42 push( data % base);

43 } while(data/=base);

44

45 int i =0;

46 for(i=0;!empty()&&i<5;pop(),i++)

47

48 if(top()<10)

49 {

50 printf("%d",top());

51 }

52 else

53 {

54 printf("%c",top()-10+'A');

55 }

56 printf("\n");

57 deinit();

58 }

实现一个简化的malloc/free

1 #include "stdio.h"

2 #include "stdbool.h"

3 #include "unistd.h"

4 //内存控制模块

5 typedef struct mem\_control\_block

6 {

7 bool free;//自由标志

8 struct mem\_control\_block\* prev;//前块指针

9 size\_t size;//本块大小

10 }MCB;

11

12 //栈顶指针

13 MCB\* g\_top = NULL;

14

15 void\* mymalloc(size\_t size)

16 {

17 MCB\* mcb;

18 for(mcb = g\_top;mcb;mcb = mcb->prev)

19

20 if(mcb->free&&mcb->size>=size)

21 {

22 break;

23 }

24 if(!mcb)

25 {

26 mcb = sbrk(sizeof(MCB)+size);

27 if(mcb == (void\*)-1)

28 {

29 return NULL;

30 }

31 mcb->prev = g\_top;

32 mcb->size = size;

33 g\_top = mcb;

34 }

35

36 mcb->free=false;

37 return mcb+1;

38 }

39 //释放内存

40 void myfree(void\* ptr)

41 {

42 if(!ptr)

43 {

44 return ;

45 }

46 MCB\* mcb = (MCB\*)ptr-1;

47 mcb->free = true;

48 for(mcb=g\_top;mcb->prev; mcb=mcb->prev)

49 {

50 if(!mcb->free)

51 break;

52 if(mcb->free)

53 {

54 g\_top = mcb->prev;

55 brk(mcb);

56 }

57 else if(mcb!=g\_top)

58 {

59 g\_top = mcb;

60 brk((void\*)mcb+sizeof(MCB)+mcb->size);

61 }

62

63 }

64 }

#include <sys/mman.h>

void mmap(

void\* start,//映射区首地址

size\_t length,//映射区长度(字节)

int port,//映射权限

int flags,//映射标志

int fd,//文件描述符

off\_t offset//文件偏移量

)

成功将返回映射区首地址，失败返回MAP\_FAILED（-1）

port取值：

PROT\_EXEC 映射区可执行

PROT\_READ 映射区可读

PROT\_WRITE 映射区可写

PROT\_NONE 映射区不可访问

flag取值：

MAP\_FIXED 如在start上无法创建映射，返回失败

MAP\_SHARED 对映射区的写操作直接反映到文件中

MAP\_PRIVATE 对映射区的写操作只反映到内存中，不进文件

MAP\_ANONYMOUS 内存映射，将虚拟地址，映射到物理地址

MAP\_DENYWRITE 拒绝其他文件对它的写入

MAP\_LOCKED 锁定映射区域

int munmap(

void\* start,//映射区首地址

size\_t length//映射区长度(按页取整)

)

mmap和munmap()在内存分配/回收时提供了更多的选择，是一个可管理的内存分配方式，可以用第一个参数设定分配的首地址，也可以用第三个参数设定权限。默认情况下mmap()映射文件

1 #include "stdio.h"

2 #include "stdlib.h"

3 #include "unistd.h"

4 #include "sys/mman.h"

5 int main()

6 {

7 char\* psz = mmap(sbrk(0),8192,

8 PROT\_READ|PROT\_WRITE,

9 MAP\_PRIVATE|MAP\_ANONYMOUS,0,0);

10 if(psz == MAP\_FAILED)

11 {

12 perror("mmap");

13 return -1;

14 }

15

16 printf("psz = %p\n",psz);

17 sprintf(psz,"Hello , Memory!");

18 printf("%s\n",psz);

19 printf("%d\n",getpid());

20 getchar();

21

22 if(munmap(psz,4096)==-1)

23 {

24 perror("munmap(1)");

25 return -1;

26 }

27 getchar();

28 if(munmap(psz+4096,4096)==-1)

29 {

30 perror("munmap(2)");

31 return -1;

32 }

33 getchar();

34 char\* p = malloc(1024);

35 free(p+512);

36 return 0;

37

38

39 }

# 文件

在Linux中，几乎一切都是文件。目录，内存，各种硬件设备都可以看做文件。

标准C语言中，用FILE\*（文件指针）代表一个打开的文件，UNIX C用文件描述符代表一个打开的文件，文件描述符其实就是一个非负整数，文件描述符自身不存储任何文件信息，信息都存在文件表中，文件描述符对应文件表。

对应Linux，一个进程最大打开256个文件，描述符从0开始计算。0,1,2

系统已经占用，程序员不能够使用，它们分别代表标准输入，标准输出和标准错误。程序员的文件描述符从3开始。

## 1文件读写函数

open(): 打开一个文件，返回文件描述符

read()/write(): 读/写一个文件

close() 关闭文件

int open(char\* filename,int flags,…)

参数：filename是打开文件的路径(包括文件名)

flag: 标识，主要有以下宏定义

权限标识：O\_RDONLY

O\_WRONLY

O\_RDWR

必须选择其中一个

附加标识：O\_APPEND 用追加的方式打开(从文件尾开始写，读文件可以不必)

创建标识： O\_CREAT(存在就打开，不存在就新建)

O\_TRUNC(文件存在时就清空所有数据)

O\_EXCL(不存在就新建不存在返回-1代表出错)

… 是可变常参数，代表0-n个任意的参数，只有

在新建文件时，才使用，传入新文件的权限。

第三个参数是文件在硬盘上的权限(某些权限可能被系统屏蔽)

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

int write(int fd,void\* buf,size\_t length)

参数：fd文件描述符，即open的返回值

buf 读写的首地址，任何类型都可以

size是buf的大小

length 是真实写入的长度

返回值：

正数：真实读/写入的字节数

0 ：读到文件的末尾，（什么都没有写入）

-1 ：出错

read（）返回0常用于循环读文件的退出

实现文件的拷贝

#include "stdio.h"

#include "stdlib.h"

#include "unistd.h"

#include <fcntl.h>

int main()

{

int fdr = open("a.txt",O\_RDWR);

if(fdr == -1)

perror("open a"),exit(-1);

int fdw = open("b.txt",O\_WRONLY|O\_CREAT|O\_TRUNC,0666);

if(fdw==-1)

perror("open a"),exit(-1);

char buf[100] = {};

while(1)

{

int res = read(fdr,buf,sizeof(buf));

if(res == -1)

{perror("read");break;}

if(res==0)

break;

write(fdw,buf,res);

}

return 0;

}

员工管理模块：

新增员工，查询新员工

输入员工信息，把信息存入文件中，员工信息包括：

员工ID，员工姓名，工资

从文件中读取员工的信息，并打印出来。

Staff.h

#include "stdio.h"

#include "stdlib.h"

#include "unistd.h"

#include "fcntl.h"

typedef struct tag\_staff{

int id;

char name[10];

unsigned int salay;

}staff;

int add\_staff(char\* filename);//添加员工信息

int read\_staffFile(char\* filename);//打印员工信息

staff.c

#include "staff.h"

int add\_staff(char\* filename)

{

staff buf;

printf("请输入新员工的信息\n");

printf("姓名:");

scanf("%s",buf.name);

printf("工资:");

scanf("%d",&buf.salay);

printf("编号:");

scanf("%d",&buf.id);

printf("\n");

int fd = open(filename,O\_RDWR|O\_APPEND);

int ret = -1;

if(fd == -1)

{

return ret;

}

ret = write(fd,&buf,sizeof(buf));

close(fd);

return ret;

}

int read\_staffFile(char\* filename)

{

int fd = open(filename,O\_RDONLY);

int ret = -1;

if(fd == -1)

{

return ret;

}

staff buf;

do

{

ret = read(fd,&buf,sizeof(staff));

if(ret >0)

{

printf("姓名:%s\t\t工资:%d\t\t编号:%d\n",buf.name,buf.salay,buf.id);

}

}while(ret>0);

close(fd);

return ret;

}

int main()

{

char filename[] = "staff.txt";

int a = add\_staff(filename);

add\_staff("staff.txt");

//add\_staff("staff.txt");

//add\_staff("staff.txt");

//add\_staff("staff.txt");

//add\_staff("staff.txt");

int b = read\_staffFile(filename);

return 0;

}

## Printf函数系列

int printf(const char\* format,…)

int fprintf(FILE \*stream,const char\*format,…)

根据指定的格式（format）向输出流stream写数据

int sprintf(char\* str,Const char \*format,…)

主要功能是把格式化的数据写入某个字符串

如 sprintf(buf,“%d,%s,%lf”,id,name,sal);

就向字符串buf中写入了作为整型的id，和作为字符串性的name

## 字符串的处理

可以定义的字符串有以下类型：

‘abc’ 字面值，本身不是变量

char buf[length] 字符数组

char\* st 字符指针

数组可以看做常指针(不能改地址，只能初始化)。

## 标C读写文件

#include <stdio.h>

#include <stdlib.h>

#include <fcntl.h>

#include <unistd.h>

int main()

{

//标准c文件函数

FILE\* file = fopen("a.txt","w");//获取文件指针

if(file == NULL)

perror("open"),exit(-1);

int i;

for(i=0;i<10000;i++)

{

fwrite(&i,4,1,file);

}

fclose(file);

//UC 文件函数

int arr[10000];//数组做缓冲

int fd = open("a.txt",O\_WRONLY|O\_CREAT|O\_TRUNC,0666);

if(fd == -1)

perror("open"),exit(-1);

int i ;

for(i=0;i<10000;i++)

{

arr[i%10000]=i;

if(i%10000 == 9999)

write(fd,arr,sizeof(arr));

}

close(fd);

return 0;

}

所有的标C函数都用在用户层定义了输入/输出缓冲区，作用就是累积到一定量后再进入内核读/写一次，UC函数都没有定义缓冲区，但可以由程序员自定义缓冲区

1. lseek()

lseek()可以随意移动偏移量

int lseek(int fd,int offset,int whence)

其中fd是文件描述符，offset是偏移量，whence是偏移的开始位置。

这个函数将返回当前位置到文件头的偏移量，失败返回-1

Whence+offset可以确定位置，whence包括：

SEEK\_SET 从头开始

SEEK\_CUR 从当前位置开始

SEEK\_END 从结尾开始

1. fcntl()

fcntl()可以复制文件描述符，

可以设置/获取描述符的状态

可以设置文件锁

函数原型：int fcntl(int fd,int cmd,…)

参数：fd 文件描述符

cmd 命令，设置fcntl完成的功能

F\_DUPFD 复制文件描述符，传入第三个参数做新

# 文件锁

当多个进程同时写一个文件时，有可能发生数据混乱，这个问题可以用文件锁的方式解决。

文件锁就是当文件读的时候，加锁使其它进程能读不能写，当文件写的时候，加锁使其它进程既不能读，也不能写。

文件锁照样可以用fcntl实现

fcntl(fd,cmd,…)

当cmd为F\_SETLK/F\_SETLKW时，可以对文件上锁。当使用文件锁时，第三个锁就是结构体flock

struct flock {

...

short l\_type; /\* Type of lock锁的类型: F\_RDLCK(读锁),

F\_WRLCK（写锁）, F\_UNLCK（释放锁） \*/

short l\_whence; /\* How to interpret l\_start:锁定的起始点的参考位置

SEEK\_SET, SEEK\_CUR, SEEK\_END \*/

off\_t l\_start; /\* Starting offset for lock ，针对起始点的偏移量\*/

off\_t l\_len; /\* Number of bytes to lock 锁定的区间长度\*/

pid\_t l\_pid; /\* PID of process blocking our lock

(set by F\_GETLK and F\_OFD\_GETLK) \*/

...

};

进程结束会自动释放文件锁，但最好自己释放。

# 程序和进程

程序是由代码生成的保存在硬盘中的可执行的文件，进程是在内存中被CPU调用和处理的一段内存区域的泛称即（程序的执行）。硬盘中的文件加载到内存中就成了进程。

## 1父进程和子进程

如果由a进程启动了b进程，那么a进程就是b进程的父进程，b就是a进程的子进程。

## 2 进程状态

S 休眠状态

s 有子进程

R 正在运行

Z 僵尸进程（已经结束，但资源没有回收）

T 暂停或被追踪

## 3 进程ID

每个进程都用进程ID(PID)做唯一的标识，进程PID是系统管理。

函数getpid()可以获取进程的PID。如果进程结束，PID可以重复使用，但要延迟重用。PID唯一标识一个进程

## 4 获取进程

getpid() 取当前进程的PID

getppid() 取父进程的PID

getuid() 取用户的ID

## 5 创建子进程

fork() 通过复制父进程创建子进程

vfork() + execl() 不复制，创建全新子进程

### 5.1 fork

进程PID用pid\_t类型，是一个非负整数，pid\_t fork()，返回子进程的PID，出错返回-1。

fork()是通过复制父进程的内存空间创建的子进程，除了代码区父子进程共享，其他内存区域都要复制。

fork()函数在复制父进程的内存空间时，如果遇到文件描述符，复制描述符但不复制文件表。

fork()在复制父进程的内存空间时，也会复制输出/输入缓冲区

fork()函数调用一次，返回两次，父进程返回一次，子进程也会返回一次。父进程返回子进程的PID，子进程返回0

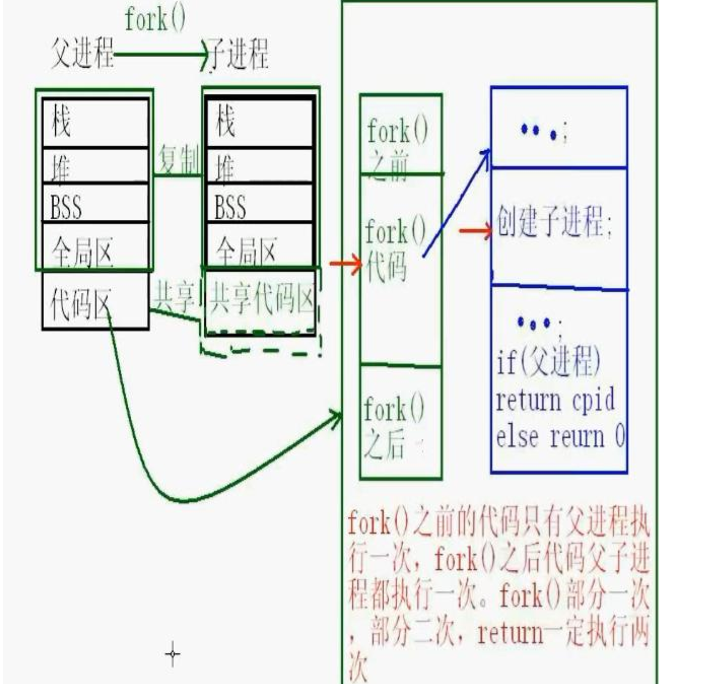
父进程启动子进程后，父子进程同时运行，如果子进程先结束，会给父进程发送信号，父进程负责回收子进程的资源。

父进程启动子进程后，父子进程同时运行，如果子进程没有给父进程发信号就结束，或者父进程没有及时处理信号，此时子进程就变成僵尸进程。

fork()之前的代码父进程执行一次，fork()之后的代码父子进程分别执行一次，fork()将返回两次。

刷新输出缓冲区的条件

1. 遇到换行 \n
2. 缓冲区满了
3. 程序结束了
4. fflush()函数人工刷新



### 5.2 结束进程

正常结束：主函数执行return

执行exit()

\_Exit()和\_exit()

非正常结束:信号中断进程（ctrl+c,kill -9）

最后一个线程被取消

5.3 wait()/waitpid()

可以让父进程等待子进程结束，病取得子进程的退出状态和退出码

pid\_t wait(int \* status)

wait()函数可以让父进程等待任意一个子进程的结束，并返回结束子进程的PID。把结束子进程的退出状态和退出码存入status，如果果没有子进程结束，会阻塞父进程，直到有子进程结束为止。包括僵尸进程.

宏函数WIFEXITED(status)判断是否正常结束，而WEXITSTATUS（status）可以获得退出码

waitpid() 可以设置等待的方式和等待的子进程。

函数原型：

pid\_t waitpid(pid\_t pid,int\* status,int options)

参数：pid 可以指定等待哪个子进程

-1 等待任意子进程，和wait()一样

>0 等待子进程的ID = pid

0 等待本组子进程

<-1 等待进程组为|pid|的所有子进程

status 用法和wait一样

options 可以设置非阻塞的等待（不等待）

为0 没有子进程结束继续等待

为WNOHANG 没有子进程结束不等待，直接返回0

返回值：结束子进程的ID

-1 代表出错

0 只有在options为WNOHANG时返回，代表没有子进程结束，也没有出错

### 5.4 vfork()

fork()父子进程使用相同的代码区，如果需要父子进程代码区不同，可以用vfork()+execl()

vfork()创建新的子进程，execl()负责提供子进程的代码和数据

execl()函数是用新的程序替换原有的程序。

vfork()从语法上和fork()完全一样，区别在于vfork()不复制任何父进程的资源，vfork()会抢父进程的资源，导致父进程阻塞，父进程解除阻塞的条件：

1 子进程结束时，归还父进程的资源

2 子进程调用exec系列函数(execl等)，也会归还父进程的资源

vfork()确保子进程先运行(父进程没有资源)，调用execl()之后，父子进程同时运行。

vfork()创建的子进程必须用exit()退出。

execl()可以用一个新程序替换旧程序，但不创建任何新的进程，如果新的程序正常启动，旧程序不再继续运行，如果新的程序启动失败，旧程序继续运行。

execl(程序所在路径，命令，选项，命令参数，NULL)，启动失败返回-1

# 信号

信号是Unix/Linux系统中，软件中断的最常用的方式。

信号本质是一个非负整数，Unix和Linux在信号上有区别，Unix是48个，Linux是64个，但中间不保证连续

每个信号都有一个宏名称， 编程时尽量使用宏名称而不是信号的值。不同的系统中，同一个宏名称对应的值可能不同，宏名称以SIG开头，比如：SIGINT就是信号2的宏名称

## 信号的处理方式

默认处理：系统对每个信号都有默认处理方式，默认处理大多数都是退出进程

忽略信号：不做任何处理，就像没有信号一样

自定义信号处理函数：信号的处理方式改为执行我们自定义的函数

注：信号9不能忽略，也不能自定义处理函数

当前用户只能给当前用户的进程发送信号，不能给其他用户的进程发送信号，root可以给所有进程发送信号

信号0没有特殊的意义，用于测试是否有发信号的权限。kill -0 3333测试对3333进程是否有发送信号的权限

## 2 信号处理函数

(void(\*f)(int)) signal(int signum,void(\*f)(int))

参数说明：signum:被设置处理方式的信号

第二个参数f是函数指针，支持三种值：

SIG\_IGN 忽略该信号

SIG\_DFL 代表信号到来执行默认处理方式

自定义的函数 代表信号到来执行自定义函数返回之前的

信号处理方式，如果出错返回SIG\_ERR

自定义信号处理方式的步骤：

1. 写一个处理函数，格式 void fa(int){}

2. 调用signal(int signum,fa)注册处理函数

如果父进程改变了信号的处理方式，由fork（）创建的子进程，与父进程的处理方式一致，vfork()+execl()创建的子进程，也与父进程保持一致。

## 3 alarm()函数

不是真正意义上的信号发送函数，而是过一段时间发送特定的信号

#include <unistd.h>

unsigned int alarm(unsigned int seconds);

seconds 延时发送闹钟信号的时间

当seconds为0时取消闹钟信号。

## 4 sleep(unsigned int seconds)

让程序休眠一段时间（秒）

usleep() 让程序休眠毫秒

# 信号集

多个信号可以存入信号集，类型sigset\_t,可以看做一个超大型整数。

数据结构包括：逻辑结构，物理结构和运算结构。逻辑结构就是逻辑上是怎样的

物理结构，内存如何组织(计算机底层的实现)

运算结构，对外提供什么函数,即实现的功能

运算结构包括：1.创建和销毁

2.增加和删除元素

3.修改和查询元素

4.其他函数，比如排序

## 1.信号集的函数

增加信号和删除信号()

查询信号

sigaddset() 增加一个信号

sigdelset() 删除一个信号

sigemptyset() 全部删除信号

sigfillset() 填满全部信号

sigismember() 查询有没有某个信号

## 2. 信号屏蔽

信号不确定发送的时间，因此可能在执行关键代码时，接收到信号，因此可能产生重大的错误，程序员无法阻止信号的到来，但可以屏蔽信号，就是信号可以到来但暂时不处理，等关键代码执行完毕再解除信号屏蔽后再处理。

信号屏蔽/解除函数 sigprocmask()

int sigprocmask(int how,sigset\_t\* set,sigset\_t\* old);

参数：how 就是信号屏蔽的方式

SIG\_BLOCK 旧的屏蔽+新的屏蔽

ABC+CDE --- ABCDE

SIG\_UNBLOCK 相当于旧的屏蔽-新的屏蔽

ABC-CDE AB

SIG\_SETMASK 无视旧的，直接替换成新的屏蔽

set 新的权限屏蔽字

old 传出参数，可以传出旧的权限屏蔽字，用于恢复之前的屏蔽

信号屏蔽之后，一定要解除屏蔽

信号9 屏蔽无效

sigpending()

这个函数可以判断在信号屏蔽期间，有没有信号来过，功能就是把信号屏蔽期间来过的信号放入信号集。

#include <signal.h>

int sigpending(sigset\_t \*set);

## 3 计时器

每个进程在Linux中有三种计时器，真实计时器，虚拟计时器和实用计时器

其中真实计时器是产生SIGALRM信号来工作。计时器可以用settimer()进行设置。

int setitimer( int which, 选择哪种计时器

const struct itimerval \*new\_value,设置计时器的开始

struct itimerval \*old\_value时间间隔);

struct itimerval {

struct timeval it\_interval; 周期性的间隔时间/\* Interval for

periodic timer \*/

struct timeval it\_value; 直到下次的到期时间 /\* Time until n

ext expiration \*/

};

struct timeval {

time\_t tv\_sec;秒 /\* seconds \*/

suseconds\_t tv\_usec; 微秒 /\* microseconds \*/

};

# 进程间通信

Unix/Linux系统基于多进程，进程和进程之间经常做数据的交互，即进程间通信。

常见的IPC

1 文件

2 信号

3 管道

4 共享内存

5 消息队列

6 信号量集

7 网络编程

共享内存，消息队列，信号量集都遵循相同的规范，因此编码上有很多的共同点，并统称为XSI IPC。

XSI IPC的通用规范：

1 所有的IPC结构都有一个内部的ID做唯一标识

2 内部ID的获取需要借助外部的key,类型为key\_t

3 key的获取方式：

使用宏IPC\_PRIVATE做key，但这种方式外部无法获取，因此不用

使用ftok()提供一个key

在头文件中统一定义一个所有的key

4 用key获取内部id的函数都是xxxget()，比如shmget(),msgget()

5 每种IPC结构都提供了一个xxxctl()函数，这个函数的功能包括：

查询修改删除。

其中有一个cmd参数，值：

IPC\_STAT 查询

IPC\_SET 修改

IPC\_RMID 按ID删除IPC结构

6 所有IPC结构都是内核管理，不使用时需要手工删除

7 IPC结构的相关命令

ipcs查看当前的IPC结构

ipcrm 删除当前的IPC结构(用id删除)

选项 -a 所有IPC结构

-m 共享内存

-q 消息队列

-s 信号量集

网络编程以前用于IPC，现在更多的用于网络.

## 1 管道(pipe)

即用管道文件作为交互媒介的IPC，管道文件是一种特殊的文件，文件类型是p。

mkfifo命令/函数 管道文件名 就能够创建管道文件，touch命令和open都无法创建管道文件。

管道文件只是交互的媒介，不存储任何的数据，只有在有读进程和写进程的时候才能畅通，否则阻塞。

管道分为有名管道和无名管道，有名管道可以用于所有进程之间的交互，而无名管道只能用于fork()创建的父子进程之间的交互。

有名管道即程序员创建管道文件进行IPC，无名管道就是系统创建和维护管道文件进行IPC。

有名管道的用法：

1.用mkfifo命令/函数 创建管道文件

2.像读写普通文件一样操作管道文件

3.如果不再使用管道文件，可以删除

共享内存：

以一块共享的物理内存做媒介，通常情况下，两个进程无法之间映射相同的内存。共享的实现过程:

1.内核先拿出一块物理内存，内核负责管理。

2.允许所有进程对这块内存进行映射

3.两个不同的进程就可以映射到相同的物理内存上，从而实现信息的交互。

编程步骤：

1.获取key，获取方式ftok()或头文件定义

2.使用shmget()函数获取内部ID

3.使用shmat()挂接共享内存(映射)

4.可以像正常操作一样使用共享内存

5.使用shmdt()脱接共享内存(解除映射)

6.如果确定已经不再使用，可以使用shmctl()删除共享内存

key\_t ftok(char\* pathname,int projectid)

参数pathname是一个真实存在的可访问的路径，projectid是项目编号，低8位有效，返回key.

ftok()如果给定路径有效，不会出错。会按着路径和项目ID生成一个key，相同的路径+相同的项目ID生成相同的key。

int shmget(key\_t key,size\_t size,int flag)

参数：key 即上面的key，

size 共享内存的大小

flag 在获取时用0，在新建时用IPC\_CREAT|0666(权限)

void\* shmat(int shmid,0,0)

int shmdt(void\* addr)，addr是shmdt()的返回值，即虚拟的首地址

shmctl() 可以查询，修改，和删除共享内存。

查询时，会把共享内存的信息放入第三个参数

修改时，只有用户id,组id和权限可以修改

删除时，第三个参数给0即可

删除共享内存时，挂接数必须为0时才能真正删除，否则删除只是做一个删除标记，等挂接数为0时才真正删除

消息队列

共享内存虽然速度最快，但当多个进程同时写数据时，会发生互相覆盖，导致数据混乱。

消息队列就可以解决多个进程同时写数据的问题

消息队列就是存放消息的队列，队列是线性的数据结构，先入先出。一般情况下，队列有满有空。数据先封入消息中，然后把消息存入队列。

消息队列的编程步骤：

1. 得到外部的key,用函数ftok()

2. 用key创建/获取一个队列(消息队列)，函数msgget()

3. 使用msgsnd()/msgrcv()把数据/消息 存入队列或从队列中取出

4. 如果不再使用消息队列，可以msgctl()删除

msgget()和msgctl()与共享内存的函数相似

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

int msgsnd(int msqid, const void \*msgp, size\_t msgsz, int msgflg);

参数msgid就是消息队列的ID，msgp就是消息的首地址，其中消息分

有型消息和无类型消息，有类型消息：

struct msgbuf {

long mtype; /\* message type, must be > 0 \*/消息类型

char mtext[1]; /\* message data \*/消息数据

};

msgsz是数据区域的大小，不包括emtype

msgflg是选项，可以是0(代表阻塞，队列排满了等待)，也

可以是IPC\_NOWAIT代表非阻塞(队列满了返回错误)

成功返回0，失败返回-1

ssize\_t msgrcv(int msqid, void \*msgp, size\_t msgsz, long msgtyp,

int msgflg);

前三个参数和msgsnd()一样

参数msgtyp可以让接收者有选择的接收消息，值可能是：

0 接收任意类型的消息(第一个，先入先出)

>0 接收 类型为msgtype的特定消息

<0 接收类型小于msgtype绝对值的消息

信号量

设定一个初始计数，每来一个进程计数-1，每完成一个进程计数+1，计数到0不允许进程访问，直到大于0为止。

信号量的编程步骤

1.获得key

2.用semget() 获取信号量集的ID

3.用semctl()设置信号量的初始计数

4.用semop()进行加1或减1的操作

5.如果不再使用，用semctl()删除

其中semctl()设置初始值的代码

semctl(semid,index,SETVAL,count)

其中semid是信号量集的ID，index是信号量在信号量集中的小标，SETVAL是宏，count是该信号量的初始计数。

int semop(int semid,struct sembuf,semoparray[],size\_t nops)

参数semoparray是一个指针，它指向一个信号量操作数组，信号量操作由sembuf结构表示：

struct sembuf{

unsigned short sem\_num;//操作信号量的下标

short sem\_op;//对信号量的操作方式，-1和1

short sem\_flg;//0或IPC\_NOWAIT

}

# 网络编程

## 1 OSI七层模型

物理层，数据链路层，网络层，传输层，回话层，表现层，应用层

协议就是计算机信息交互时的规范，常见的协议：

http协议 超文本传输协议

ftp协议 文件传输协议

tcp协议 传输控制协议(传输层)

udp协议 用户数据报协议(传输层)

ip协议 网络层

协议簇即多个协议的集合，协议簇一般以核心协议命名

## 2 IP和端口

IP地址就是网络中计算机的唯一标识，本质是一个整数。IP地址早期都是32位，叫IPV4，后来推出IPV6

IP地址有两种描述方式:

点分十进制就是每8位做一个整数，分4段，中间用.隔开

十六进制就是把32位二进制直接写成8位十六进制

IP地址绑定网卡，每个网卡在出厂时都有一个唯一的物理地址(MAC地址)，IP地址是找到网卡的物理地址，从而找到计算机。

IP地址分为A/B/C/D四类，系统预留了127.0.0.1，做为本机的IP地址

IP地址可以让我们找到计算机，但不能找到对应的进程。在网络中，端口代表计算机内部的一个进程，网络地址加端口就可以网络通信。

ip地址+端口号的网络编程就是socket编程，端口号也是一个整数short，0到65535。其中：0-1023是作为系统的端口使用，不能使用，48000后也不要使用，不稳定。有些软件会强占一个端口，如Oracle数据库，1521端口

Http 80 ftp 21 telnet 23

## 3 字节顺序

整数是4个字节，有些计算机从低位字节到高位字节存储，有些机器从高位字节到低位字节存储。

本机的字节顺序无法确定，但网络的字节顺序是固定的。编程用网络字节顺序传输，到本地以后再由网络转换本地格式。

## 4. Unix/Linux网络编程的实现

### 4.1 服务端的编程步骤

1 创建一个socket，使用socket()

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

domain :用于选择协议簇

AF\_UNIX/AF\_LOCAL/AF\_FILE：本地通信IPC

AF\_INET: 网络通信

AF\_INET6: IPV6的网络通信

type 选择通信的类型(协议)

SOCK\_STREAM : 数据流(TCP协议)

SOCK\_DGRAM ： 数据报(UDP协议)

protocol 本来应该选择协议，但因为前两个参数已经决定，所以给0

成功会返回socket描述符，类似文件描述符。失败返回-1

2 准备通信地址(IPC是文件，网络是IP/端口)

系统提供了三种通信地址，即三个结构体

struct sockaddr 本身不保存数据，做函数的参数

struct sockaddr\_un 存本地通信的通信地址

struct sockaddr\_in 存网络通信通信地址

struct sockaddr\_in{

int sin\_family;//协议簇，与socket()一致

short sin\_port;//端口号

struct in\_addr sin\_addr;//IP地址

}

3 绑定socket描述符和通信地址

bind(int sockfd,struct sockaddr\* addr,int length)

length是通信地址sizeof

4 通信( read() write() )

5 关闭socket描述符（ close() ）

### 4.2 客户端的编程步骤

和服务器编程一样，除了第三步把bind换成connect()，但函数的参数不用改变。

bind()是服务器绑定的通信地址，开放端口。connect()是客户端链接服务器，通信地址要使用服务器的

在使用网络编程时，IP地址和端口号都需要做一些处理，IP地址需要做点分十进制和十六进制转换，使用函数inet\_addr()，端口号需要本机格式和网络格式之间的转换，使用函数htons()

## 5一对多的编程模型

5.1 TCP一对多

有两种socket描述符，其中一种负责等待客户端链接，当有客户端连接时，启动另外一个新的描述符负责信息交互。

TCP是一个基于连接的协议，全程保持客户端和服务器的连接，会重发一切的错误数据，因此TCP可以保证数据的完整和有效，缺点就是当客户端多的时候，效率非常低。

在服务端：

1.socket() 得到第一类的socket描述符

2.准备通信地址 struct sockaddr\_in

3.绑定bind()

4.监听客户端的连接，函数listen()

5.等待客户端的连接，函数accept(),返回新的描述符，用于信息交互（无客户端连接会阻塞）

6.用第五步返回描述符进行读写操作

7.close()关闭两个描述符

在客户端：

客户端的编程步骤和前面的一样，当多个客户端同时连接时，需要把多余的客户端存入队列，backlog就是队列的最大长度

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

参数sockfd就是socket描述符，第一步返回addr是一个结构体指针，存客户端的通信地址len是传入传出参数，先传入addr的长度，再传出接收到的客户端通信地址的真实长度

返回新的socket描述符，失败返回-1

### 5.2 UDP

TCP和UDP的区分主要在于socket()第二个参数，如果SOCK\_STREAM就是TCP，SOCK\_DGRAM就是UDP

UDP发送数据一般使用sendto(),接收数据可以使用read()/recvfrom()

区别就是read()函数不知道数据的来源，而recvfrom()可以获取数据的发送者信息。

# 线程

线程是轻量级的代码并行，不需要额外创建过多的内存空间，而是共享所在进程的内存空间，线程只需要额外建立一个独立的栈即可。

CPU不可分，大多数操作系统都是采用CPU时间片实现CPU的在多线程之间的轮换。CPU时间片是一段CPU的执行时间，时间极其短。

## 1.创建线程

Unix/Linux的线程相关函数都在pthread.h中，代码都在libpthread.so中，线程相关的函数/结构都以pthread\_开头。

创建线程：

int pthread\_create(

pthread\_t \*thread, //线程id

const pthread\_attr\_t \*attr,//线程的属性

void \*(\*start\_routine) (void \*), //线程执行的代码

void \*arg//传给start\_routine的参数

);

成功返回0，失败返回错误码，错误信息需要用strerror()做转换

每个线程启动以后，只能执行一个函数，主线程执行的是main()函数，其他线程执行自定义的一个函数，这个函数以并行的方式运行。

线程之间的代码乱序执行，每个线程的内部代码都是顺序执行。每个线程都会返回自己的错误码，而不是errno。

pthread\_join()函数可以让一个线程等待另外一个线程的结束，并取得线程的返回值。如在线程a中调用pthread\_join(b,0),线程a就等待b结束再继续执行。

## 2线程状态

分离状态

线程一旦结束，不用管其他线程，直接回收资源

join状态

如果线程用pthread\_join()，就处于join()状态，就是线程结束时暂

不回收资源，到pthread\_join()结束再回收资源

## 3 线程的退出

正常退出

在线程的函数中执行returen；

在线程中执行pthread\_exit(void\*)函数

非正常退出

自身出错

被其他线程终止/取消

exit()和pthread\_exit(void\*)的区别

exit()是结束进程，所有线程全部结束

pthread\_exit()是结束线程，其他线程继续运行

参数void\*和return 一样，都用于返回值

取消线程的函数：pthread\_cancel();

## 4 线程同步

多线程之间是共享进程的资源，所以可能出现共享数据的冲突，解决方案就是把并行访问改为串行访问，这种技术称为线程同步。

主要包括：互斥量，信号量，条件变量

### 4.1 互斥量

互斥量又叫互斥锁，是线程在设计时，官方的同步技术，编程步骤

1.声明互斥量

pthread\_mutex\_t lock;//

2.初始化互斥量

pthread\_mutex\_init(&lock);

3.加锁

pthread\_mutex\_lock(&lock);

4.执行共享数据的访问代码

5.解锁 pthread\_mutex\_unlock(&lock)

6.释放锁的资源

pthread\_mutex\_destroy(&lock)

### 4.2 信号量

信号量是一个线程计数辅助，头文件semaphore.h,信号量如果初始计数为1，效果等同于互斥量。

编程步骤：

1.声明信号量 sem\_t sem;

2.初始信号量的原始计数sem\_init()

sem\_init(&sem,0,count)

第一个参数就是信号量的地址

第二个参数必须是0,0代表线程的计数，非0代表进程的计数(Linux系统没有提供进程计数功能)

第三个参数是计数的初始值

3.计数减1， sem\_wait(&sem)

4.正常使用

5.计数加1 sem\_post(&sem)

6.释放信号量资源 sem\_destroy(&sem)