

akaedu.org-1

xwp

March 24, 2014

Contents

1	linux c 基础复习	3
1.1	基本命令	3
1.2	vim 操作	3
1.3	c 基础	3
1.3.1	基本类型	3
1.3.2	循环控制语句	3
1.3.3	数组	3
1.3.4	指针	3
1.3.5	结构体	3
2	Ubuntu 系统安装软件方法	3
2.1	Ubuntu 服务器软件源安装	3
2.2	deb 包安装	4
2.3	源代码安装	4
2.4	ubuntu 使用小技巧	4
3	git 使用教程	5
4	计算机组成原理	5
4.1	寄存器	5
4.2	CPU	5
4.3	内存	5
4.4	总线	5
4.5	MMU	5

5	X86 汇编	5
5.1	学习汇编的目的	5
5.2	第一个汇编程序	5
5.2.1	编译链接过程	6
5.3	X86 寄存器	7
5.4	汇编语法	7
5.5	寻址方式	7
5.6	E L F 文件格式	7
5.6.1	绝对地址和相对地址辨析	8
5.6.2	Section 和 Segment 辨析	8
5.7	查看反汇编调试程序	9
6	C 和汇编的关系	9
6.1	重点指令讲解	9
6.1.1	栈的类型	9
6.1.2	call	9
6.1.3	ret	9
6.1.4	leave	10
6.1.5	总结	10
6.2	变量存储布局	10
6.2.1	存储区域	10
6.2.2	存储的生命周期	10
6.2.3	作用域	10
6.2.4	延长局部变量生命周期	11
6.2.5	命名空间	11
6.2.6	链接属性	11
6.2.7	其他	11
7	高级指针	11
7.1	指针数组	11
7.2	数组指针	12
7.2.1	数组指针与二维数组可等价转换	12
7.3	二重指针	12
7.3.1	二重指针和数组指针的关系	13
7.3.2	二重指针作为函数的参数	14
7.4	函数指针	14
7.4.1	基本定义	14
7.4.2	函数指针作为函数的参数	15
7.4.3	泛性函数设计	18
7.4.4	函数指针作为函数的返回值	20

7.4.5	函数指针作为函数的参数（回调函数）	20
7.4.6	函数指针数组	20
7.4.7	可变参函数 printf 实现	21
8	预处理课件	21
9	Makefile 课件	22
10	文件操作	23
10.1	刷新终端文件缓冲区	23
10.2	FILE 结构体定义	23
10.3	查看 errno 错误号的头文件	23
10.4	文件操作课件	24
10.5	文件操作练习	24

1 linux c 基础复习

1.1 基本命令

1.2 vim 操作

1.3 c 基础

1.3.1 基本类型

1.3.2 循环控制语句

1.3.3 数组

1.3.4 指针

1.3.5 结构体

2 Ubuntu 系统安装软件方法

2.1 Ubuntu 服务器软件源安装

- Ubuntu 软件中心 -> 编辑 -> 软件源
- 更新本地软件源列表 `sudo apt-get update`
- 搜索软件 `sudo apt-cache search thunderbird`

- 安装软件 `sudo apt-get install thunderbird`
- 卸载软件 `sudo apt-get autoremove thunderbird`

2.2 deb 包安装

- 网上下载相关软件 deb 包
- 安装 `sudo dpkg -i filename.deb`
- 卸载 `sudo dpkg -r filename`

2.3 源代码安装

- 网上下载相关软件源代码包（.tar.gz/.zip/.tar.bz2/.tar）
- 进入源码包，编译源代码包，并且安装
 - `./configure`
 - `make`
 - `sudo make install`
- 卸载源代码包安装的软件, 进入到源代码包里执行
 - `sudo make distclean`

2.4 ubuntu 使用小技巧

- 显示桌面, `Ctrl+win+d`
- 切换工作区, `Ctrl+Alt+up/down/left/right`

3 git 使用教程

4 计算机组成原理

4.1 寄存器

4.2 CPU

4.3 内存

4.4 总线

4.5 MMU

5 X86 汇编

5.1 学习汇编的目的

学习汇编不是为了掌握用汇编去写程序，而是为了更好的透视理解 C 语言行为，理解处理器的工作方式，了解程序背后底层的東西，相当于习武中的内功心法，以便你日后写出高效的代码和调试诡异的 B U G。

5.2 第一个汇编程序

```
#PURPOSE: Simple program that exits and returns a
#
status code back to the Linux kernel
#
#INPUT:
none
#
#OUTPUT: returns a status code. This can be viewed
#
by typing
#
#
echo $?
#
#
after running the program
#
#VARIABLES:
```

```

#
%eax holds the system call number
#
%ebx holds the return status
#
.section .data
.section .text
.globl _start
_start:
movl $1, %eax # this is the linux kernel command
#number (system call) for exiting
#a program
movl $4, %ebx # this is the status number we will
               # return to the operating system.
               # Change this around and it will
               # return different things to
               # echo $?
int $0x80 # this wakes up the

```

5.2.1 编译链接过程

.c -> .i -> .s -> .o -> a.out(ELF)

文件类型	文件属性	工具链	目标文件类型
.c	源代码	cpp 预处理器	.i
.i	预处理后的文件	gcc 编译器	.s
.s	编译后的文件	as 汇编器	.o
.o	汇编后的文件	ld 链接器	a.out
a.out	链接后的可执行文件	loader 加载器	进程

5.3 X86 寄存器

寄存器	功能
eax	通用
ebx	通用
ecx	通用
edx	通用
esi	通用
edi	通用
ebp	帧指针
esp	栈指针
eip	程序计数器
eflag	程序状态
浮点寄存器	浮点数运算

5.4 汇编语法

去我的 nfs 服务器上下载汇编电子教材

5.5 寻址方式

```
int a = 3, d = 5;
int b[5] = {1,2,3,4,5};
struct STU {
int id;
char name[20];
char sex;
}c;
int *p;
```

寻址方式	含义	对应 C 语法
直接寻址	movl ADDRESS, %eax	a = *p
变址寻址	movl data_items(,%edi,4), %eax	a = b[3]'
间接寻址	movl(%eax), %ebx	a = *p
基址寻址	movl 4(%eax), %ebx	a = *(p+1)
立即数寻址	movl \$12,%eax	a = 12
寄存器寻址	movl %eax, %ebx	a = d

5.6 ELF 文件格式

ELF 文件格式是一种开放的标准，unix/linux 可执行程序都采用 ELF 标准。

- 可重定位的目标文件 (Relocatable, 或者 Object File) 如.o 文件
- 可执行文件 (Executable)
- 共享库 (Shared Object, 或者 Shared Library)

5.6.1 绝对地址和相对地址辨析

- 类别绝对路径和相对路径
- 相对地址：内部偏移地址，两条指令间的相对距离
- 绝对地址：虚拟地址 4G 空间内的唯一地址

5.6.2 Section 和 Segment 辨析

- Section：站在链接器角度的链接单位，如：

段名	含义	解释
Section Headers	段符号表	保存了各个段的起始地址和大小
.text	代码段	存放程序执行的机器码指令
.ro.data	只读数据段	存放只读常量
.data	数据段	存放已初始化的全局变量
.Bss	0 数据段	存放未初始化全局变量
.shstrtab		保存着各 Section 的名字
.strtab		保存着程序中用到的符号的名字
.rel.text	重定位符号表	告诉链接器指令中的哪些地方需要做重定位
.symtab	符号表	保存了符号内部的相对地址

- Segment：站在加载器角度的加载单位，如：

PrProgram Headers:

Type	Offset	VirtAddr	PhysAddr	FileSize	MemSiz	Flg	Align
LOAD	0x000000	0x08048000	0x08048000	0x00009e	0x00009e	R E	0x1000
LOAD	0x0000a0	0x080490a0	0x080490a0	0x000038	0x000038	R W	0x1000

Section to Segment mapping:

Segment Sections...

segment 编号	segment 名
00	.text
01	.data

5.7 查看反汇编调试程序

- `gcc -g test.c -o test`
- `objdump -dSsx test > file`
- `vim file`
- 找到 `main` 函数的位置，然后开始分析

6 C 和汇编的关系

6.1 重点指令讲解

指令	示例	作用
<code>push</code>		
<code>pop</code>		
<code>call</code>		
<code>leave</code>		
<code>ret</code>		

6.1.1 栈的类型

栈的类型	工作方式
满递减	
满递增	
空递减	
空递增	

6.1.2 `call`

`call 0x80a0000<foo>`

- 把 `call` 的下一条指令的地址压到栈上保存
- 修改 `eip`，跳到 `foo` 函数的地址上去执行

6.1.3 `ret`

`call` 指令的逆向操作

- 把当前 `esp` 里保存的返回地址值给 `eip`
- 由于修改了 `eip`，所以跳向返回地址

6.1.4 leave

是函数开头的 `push %ebp` 和 `mov %esp,%ebp` 的逆操作

- 把 `ebp` 的值赋给 `esp`
- 现在 `esp` 所指向的栈顶保存着 `foo` 函数栈帧的 `ebp`，把这个值恢复给 `ebp`，同时 `esp` 增加 4

6.1.5 总结

- x86 平台函数调用，利用栈来传递参数
- 每个函数都有自己的帧指针，称为函数的栈帧，通过函数的栈帧向上可以取到调用函数的参数，向下可以取到函数定义的局部变量
- 函数调用时从右到左去压栈

6.2 变量存储布局

6.2.1 存储区域

存储区域	存储元素
栈	局部变量
data 段	已初始化全局变量， <code>static</code> 修饰并初始化的变量
bss 段	未初始化全局变量， <code>static</code> 修饰未初始化的变量
rodata	只读数据， <code>const</code> 修饰的变量
text	存放对局部变量初始化的数据

6.2.2 存储的生命周期

存储区域	生命周期
栈	函数调用结束
data 段	整个程序结束前，都有效
bss 段	整个程序结束前，都有效
rodata	整个程序结束前，都有效
text	整个程序结束前，都有效

6.2.3 作用域

定义方式	作用域
在语句块{ 定义的变量， <code>for () {int a;}</code>	本语句块内
在函数内定义的变量， <code>fun{int a;}</code>	本函数体内
在文件里，其他函数外定义的 <code>static int a;</code>	本文件内
在文件里，其他函数外定义的 <code>int a;</code>	所有文件

6.2.4 延长局部变量生命周期

```
void foo(void)
{
    static int a;
    printf("%d\n", a);
    a = 3;
    printf("%d\n", a);
}

int main(void)
{
    foo();
    foo();
}
```

6.2.5 命名空间

6.2.6 链接属性

符号	作用域
globel/外部链接	全局
local/内部链接	局部
no link	不能用于链接

6.2.7 其他

7 高级指针

7.1 指针数组

```
struct STU {
    int id;
    char name[20];
    char sex;
};
```

定义	申请字节数	含义	可以赋值吗?
char *a ¹	40	数组有 10 个 char * 指针	a ² = 'x'; *a ² = 'x'
int *a ¹	40	数组有 10 个 int * 指针	a ² = 123
struct STU *a ¹	40	数组有 10 个 struct STU* 指针	a ² = {1, "hi", 'm'}

可不可以对上述元素进行赋值? a² = 'x';

7.2 数组指针

```
struct STU {  
    int id;  
    char name[20];  
    char sex;  
};
```

定义	申请字节数	sizeof(*p)	p+1	含义
char (*p) ¹	4	10	10	
int (*p) ¹	4	40	40	
struct STU (*p) ¹	4	280	280	

辨析以下 p+1, 分别加过多少字节? (注: 加 1 表示加过 1 个元素)

```
char p[10];    p+1  
char *p[10];   p+1  
char (*p)[10]; p+1
```

7.2.1 数组指针与二维数组可等价转换

```
int foo(char s[][10], int n)  
{  
    int i;  
    for (i = 0; i < n; i++)  
        printf("%s\n", s[i]);  
}  
  
int main(void)  
{  
    char str[3][10] = {"hello", "world", "akaedu"};  
    foo(str);  
    return 0;  
}
```

7.3 二重指针

```
struct STU {  
    int id;  
    char name[20];  
    char sex;  
};
```

定义	含义	申请字节数	辨析
char **p			
int **p			
struct STU **p			

```

    int main(void)
{
    char a = 'x';
    char *q = &a;
    char **p = &q;
    *q = 'd';
    **p = 'e';
    printf("%c\n", a);
}

//void foo(char *argv[], int n)
void foo(char **argv, int n)
{
    int i;
    for (i = 0; i < n; i++)
        printf("%s\n", argv[i]);
}

    int main(void)
{
    char *buf[3] = {"hello", "world", "aka"};
    foo(buf, 3);
}

```

7.3.1 二重指针和数组指针的关系

命令行传参

```

    int main(int argc, char *argv[])
{
    int i;
    printf("argc = %d\n", argc);
    for (i = 0; i < argc; i++)
        printf("argv[%d]=%s\n", i, argv[i]);
}

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

```

```

{
if (argc < 3) {
printf("./a.out num str\n");
return -1;
}
int n = atoi(argv[1]);
while (n--)
printf("%s\n", argv[2]);
return 0;
}

```

7.3.2 二重指针作为函数的参数

7.4 函数指针

7.4.1 基本定义

函数指针就是用来保存函数地址的变量。

```

void foo(void)
{
    printf("helloworld\n");
}
int p_str(char *str, int n)
{
    if (n < 0)
        return -1;
    while (n--)
        printf("%s\n", str);
    return 0;
}
char *maxlen_str(char *str[], int n) //char *(*p)(char **, int);
{
    int maxlen = 0, i;
    char *maxstr = NULL;
    if (str == NULL)
        return NULL;
    for (i = 0; i < n; i++) {
        if (str[i] == NULL)
            continue;
        if (maxlen < strlen(str[i])) {

```

```

        maxlen = strlen(str[i]);
        maxstr = str[i];
    }
    return maxstr;
}
int main(void)
{
    void (*p)(void);
    int (*q)(char *, int);
    char *(*m)(char **, int) = maxlen_str;
    char *s[5] = {"hello", NULL, "world", "aa", "abcdefg"};
    printf("%s\n", m(s, 5));
    //q = p_str;
    //q("hello", 3);

    //p = foo;
    // p = foo();
    // *p();
    // p();
    //foo();
    return 0;
}

```

7.4.2 函数指针作为函数的参数

```

int p_str(char *str, int n)
{
    if (n < 0)
        return -1;
    while (n--)
        printf("%s\n", str);
    return 0;
}
int p_str_cat(char *str, int n)
{
    if (n < 0)
        return -1;
    while (n--)
        printf("akakedu %s\n", str);
}

```

```

        return 0;
    }
    void show_str(char *str, int n, int (*fun)(char *, int))
    {
        fun(str, n);
    }
    int main(void)
    {
        //show_str("abc", 5, p_str);
        show_str("abc", 5, p_str_cat);
    }

```

练习：动物走秀练习

```

#include <stdio.h>
#include <string.h>
struct animal {
    char name[20];
    void (*say_what)(void);
    int level;
};

typedef struct animal ANIMAL;

void cow_say(void)
{
    printf("men men\n");
}
void dog_say(void)
{
    printf("wang wang\n");
}
void man_say(void)
{
    printf("OH YEAH\n");
}
void init_animal(char *name, void (*say)(void), int level, ANIMAL *p)
{
    strcpy(p->name, name);
    p->say_what = say;
}

```



```

        p->level = level;
    }
int level_cmp(ANIMAL a, ANIMAL b)
{
    if (a.level < b.level)
        return 1;
    else if (a.level == b.level)
        return 0;
    else
        return -1;
}
int name_cmp(ANIMAL a, ANIMAL b)
{
    return strcmp(a.name, b.name);
}
void swap(ANIMAL *a, ANIMAL *b)
{
    ANIMAL tmp;
    tmp = *a;
    *a = *b;
    *b = tmp;
}
void show_animal(ANIMAL *a, int n)
{
    int i;
    for (i = 0; i < n; i++) {
        printf("%s\tsay\t", a[i].name);
        a[i].say_what();
    }
}
void sort_animal(ANIMAL *a, int n, int (*cmp)(ANIMAL, ANIMAL))
{
    int i, j;
    for (i = 0; i < n; i++)
        for (j = 0; j < n-i-1; j++)
            if (cmp(a[j], a[j+1]) > 0)
                swap(&a[j], &a[j+1]);
}
int main(void)
{

```

```

    ANIMAL a[10];

    init_animal("cow", cow_say, 2, &a[0]);
    init_animal("dog", dog_say, 1, &a[1]);
    init_animal("man", man_say, 3, &a[2]);
    sort_animal(a, 3, level_cmp);
    show_animal(a, 3);
    printf("*****\n");
    sort_animal(a, 3, name_cmp);
    show_animal(a, 3);
    return 0;
}

```

7.4.3 泛性函数设计

设计一个冒泡排序，可以排任意数据类型的数组，当用户调用此冒泡排序时，只需要提供一个比较函数即可。泛性函数设计时，通常借助 void 或 void *。

```

#include <stdio.h>
#include <string.h>

void bsort(void *a, int nmemb, int nsize, int compar(void *, void *))
{
    int i, j;
    char tmp[nsize];
    for (i = 0; i < nmemb; i++)
        for (j = 0; j < nmemb-i-1; j++)
            if (compar((void *)((char *)a+j*nsize),
                        (void *)((char *)a+(j+1)*nsize)) > 0) {
                memcpy((void *)tmp, (void *)((char *)a+j*nsize), nsize);
                memcpy((void *)((char *)a+j*nsize), (void *)((char *)a+(j+1)*nsize), nsize);
                memcpy((void *)((char *)a+(j+1)*nsize), (void *)tmp, nsize);
            }
}

int int_cmp(void *a, void *b)
{
    int m = *((int *)a);
    int n = *((int *)b);
    if (m > n)

```

```

        return 1;
    else if (m == n)
        return 0;
    else
        return -1;
}
void show_int(int a[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        printf("%d\n", a[i]);
}
void show_str(char *b[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        printf("%s\n", b[i]);
}
int str_cmp(void *a, void *b)
{
    //char *m = (char *)*((int *)a);
    //char *n = (char *)*((int *)b);
    char *m = *((char **)a);
    char *n = *((char **)b);
    return strcmp(m, n);
}
int main(void)
{
    int a[5] = {33,1,5,9,2};

    char *b[5] = {"abc", "world", "aka", "hello", "good"};
    bsort((void*)a, 5, sizeof(a[0]), int_cmp);
    show_int(a, 5);
    bsort((void*)b, 5, sizeof(b[0]), str_cmp);
    show_str(b, 5);
    return 0;
}

```

7.4.4 函数指针作为函数的返回值

7.4.5 函数指针作为函数的参数（回调函数）

7.4.6 函数指针数组

```
#include <stdio.h>
#include <string.h>
void fun_cat(char *s1, char *s2)
{
    printf("%s\n", strcat(s1, s2));
}
void fun_cmp(char *s1, char *s2)
{
    printf("%d\n", strcmp(s1, s2));
}
void fun_cpy(char *s1, char *s2)
{
    printf("%s\n", strcpy(s1, s2));
}
/*
typedef void (*FUN)(char *, char *);
FUN p[3];
*/
/*
```

练习：定义以下函数的函数指针数组？

```
char *foo(int *n, char **m)
```

解答：typedef char *(*FUN)(int *, char **)
FUN p[3];

练习：如果 foo(int n) 的函数返回值是一个函数指针
char (*)(char *, int) 类型,foo 函数如何定义？

解答：typedef char (*FUN)(char *, int)
FUN foo(int n)

练习：上题 foo 函数的函数指针数组如何定义？

解答：FUN (*p[3])(int);

```
*/
int main(int argc, char *argv[])
{
    void (*p[3])(char *, char *) = {fun_cat, fun_cmp, fun_cpy};
    char s1[100] = "hello";
    char s2[100] = "world";
```

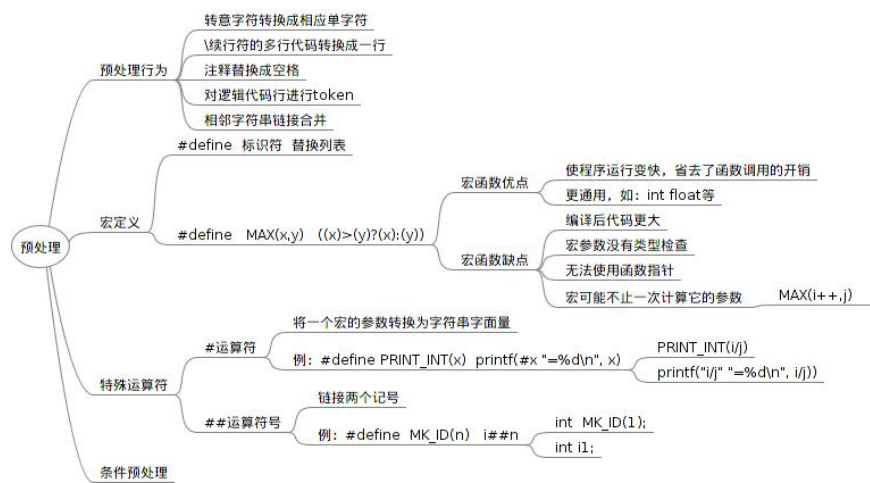
```

p[0](s1, s2);
p[1](s1, s2);
p[2](s1, s2);
return 0;
}

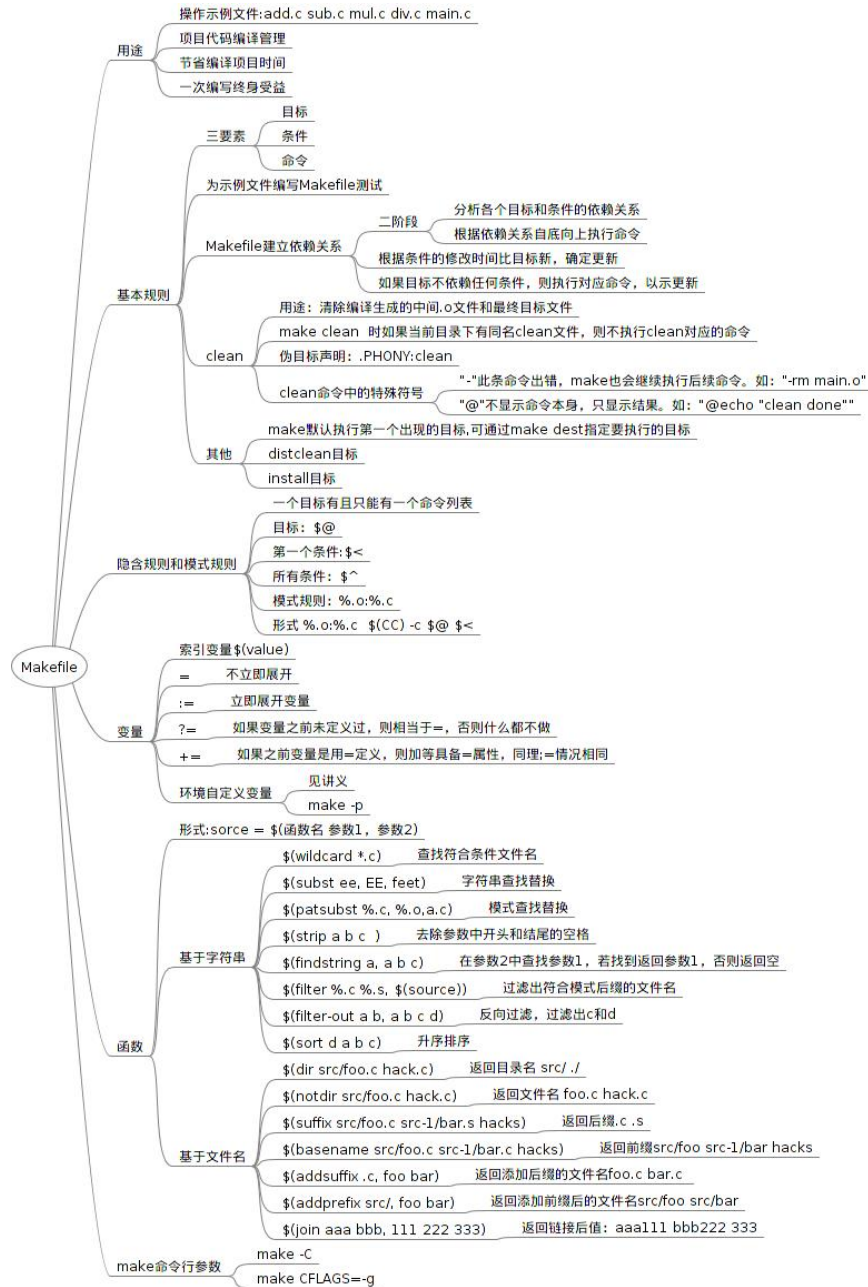
```

7.4.7 可变参函数 printf 实现

8 预处理课件



9 Makefile 课件



10 文件操作

10.1 刷新终端文件缓冲区

1. “
2. `fflush`（文件流指针）
3. 缓冲区满，默认 8192Bytes
4. 程序运行结束时，调用 `exit` 函数

10.2 `FILE` 结构体定义

查看 `stdio.h` 和 `libio.h`

```
/* Standard streams. */
extern struct _IO_FILE *stdin;      /* Standard input stream. */
extern struct _IO_FILE *stdout;     /* Standard output stream. */
extern struct _IO_FILE *stderr;     /* Standard error output stream. */
```

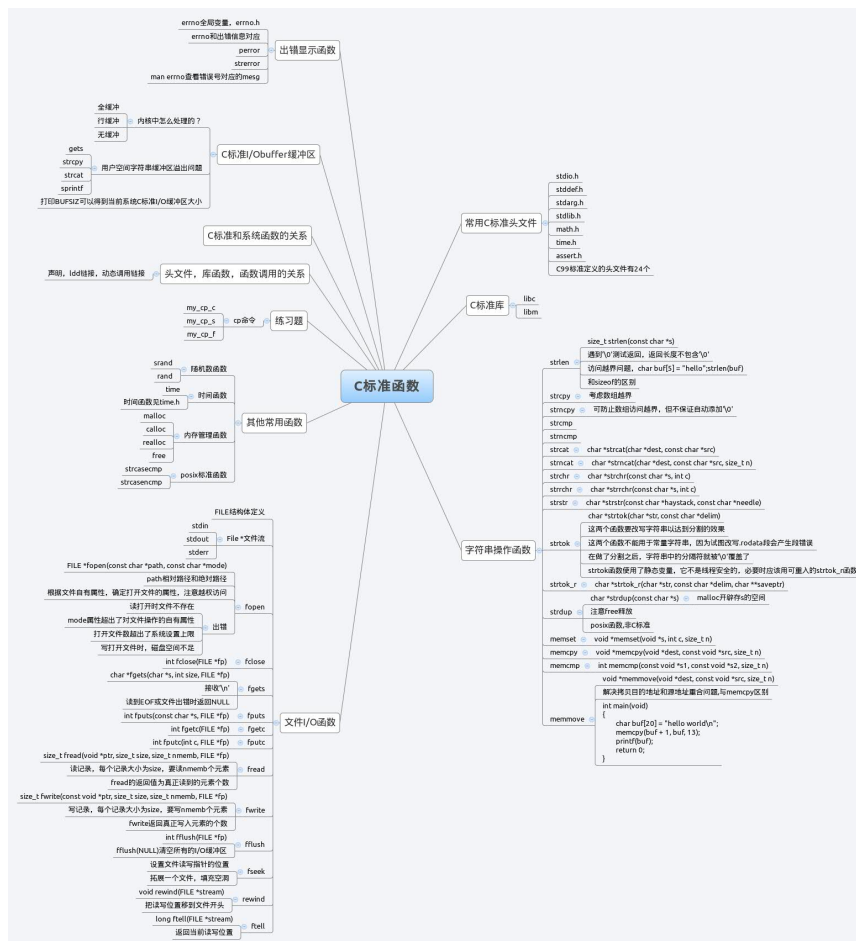
10.3 查看 `errno` 错误号的头文件

请大家查看此文件，加深印象

`/usr/include/asm-generic/errno-base.h`

`errno` 是全局变量，只记录最近一次函数调用时的出错原因，如果函数调用成功，不会修改 `errno` 的值

10.4 文件操作课件



10.5 文件操作练习

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
struct STU {
    int id;
    char name[20];
    char sex;
};
int read_file(FILE *fp, struct STU *a, int n)
```



```

{
    int i;
    char buf[100];
    for (i = 0; i < n; i++) {
        fgets(buf, sizeof(buf), fp);
        a[i].id = atoi(strtok(buf, "\n"));

        fgets(buf, sizeof(buf), fp);
        strcpy(a[i].name, strtok(buf, "\n"));

        fgets(buf, sizeof(buf), fp);
        a[i].sex = buf[0];
    }
    return i;
}

int cnt_stu(FILE *fp)
{
    int n = 0;
    char buf[100];
    while (fgets(buf, sizeof(buf), fp))
        n++;
    return n/3;
}

void show_stu(struct STU *a, int n)
{
    int i;
    for (i = 0; i < n; ++i)
        //printf("%d\t%s\t%c\n", a[i].id, a[i].name, a[i].sex);
        fprintf(stdout, "%d\t%s\t%c\n", a[i].id, a[i].name, a[i].sex);
}

void swap(struct STU *m, struct STU *n)
{
    struct STU tmp;
    tmp = *m;
    *m = *n;
    *n = tmp;
}

void sort_stu(struct STU *a, int n)

```

```

{
    int i, j;
    for (i = 0; i < n; ++i)
        for (j = 0; j < n-i-1; ++j)
            if (a[j].id > a[j+1].id)
                swap(&a[j], &a[j+1]);
}
int write_file_bin(FILE *fp, struct STU *a, int n)
{
    int i;
    //how many student num
    fwrite(&n, sizeof(int), 1, fp);
    for (i = 0; i < n; ++i)
        fwrite(&a[i], sizeof(struct STU), 1, fp);
    return i;
}
int read_file_bin(FILE *fp, struct STU *a, int n)
{
    int i;
    for (i = 0; i < n; ++i)
        fread(&a[i], sizeof(struct STU), 1, fp);
    return i;
}
int read_file_bin_head(FILE *fp)
{
    int n = 0;
    fread(&n, sizeof(int), 1, fp);
    return n;
}
int main(int argc, char *argv[])
{
    if (argc < 2) {
        printf("./a.out stufile stufile_bin\n"
            "or ./a.out stufile_bin\n");
        exit(-1);
    }
    int n;
    if (strcmp(argv[1], "stufile") == 0) {
        FILE *fp = fopen(argv[1], "r");
        if (fp == NULL) {

```

```

        perror(argv[1]);
        exit(-1);
    }
    FILE *fp_bin = fopen(argv[2], "w");
    if (fp_bin == NULL) {
        perror(argv[2]);
        exit(-1);
    }
    n = cnt_stu(fp);
#ifdef DEBUG
    printf("n = %d\n", n);
#endif
    struct STU a[n];
    rewind(fp);
    read_file(fp, a, n);
    sort_stu(a, n);
    show_stu(a, n);
    write_file_bin(fp_bin, a, n);
    fclose(fp);
    fclose(fp_bin);
}
else if (strcmp(argv[1], "stufile_bin") == 0) {
    FILE *fp = fopen(argv[1], "r");
    int n = read_file_bin_head(fp);
    struct STU a[n];
    read_file_bin(fp, a, n);
    show_stu(a, n);
    fclose(fp);
}

return 0;
}

```