第四章

在上一部分我们完成了基本的字符打印函数，但它的功能相对简单，而且除了这个基本的字符输出函数之外，为了后续开发方便，我们还需要其他一些基本的函数。

主要有这些函数，我们之间看头文件好了：

/include/string.h

1. #ifndef INCLUDE\_STRING\_H\_
2. #define INCLUDE\_STRING\_H\_
3. #include "types.h"
4. void memcpy(uint8\_t \*dest, const uint8\_t \*src, uint32\_t len);
5. void memset(void \*dest, uint8\_t val, uint32\_t len);
6. void bzero(void \*dest, uint32\_t len);
7. int strcmp(const char \*str1, const char \*str2);
8. char \*strcpy(char \*dest, const char \*src);
9. char \*strcat(char \*dest, const char \*src);
10. int strlen(const char \*src);
11. #endif

具体实现的代码：

/libs/string.c

1. #include "string.h"
2. inline void memcpy(uint8\_t \*dest, const uint8\_t \*src, uint32\_t len)
3. {
4. for (; len != 0; len--) {
5. \*dest++ = \*src++;
6. }
7. }
8. inline void memset(void \*dest, uint8\_t val, uint32\_t len)
9. {
10. uint8\_t \*dst = (uint8\_t \*)dest;
11. for ( ; len != 0; len--) {
12. \*dst++ = val;
13. }
14. }
15. inline void bzero(void \*dest, uint32\_t len)
16. {
17. memset(dest, 0, len);
18. }
19. inline int strcmp(const char \*str1, const char \*str2)
20. {
21. while (\*str1 && \*str2 && (\*str1++ == \*str2++))
22. ;
23. if (\*str1 == '\0' && \*str2 == '\0') {
24. return 0;
25. }
26. if (\*str1 == '\0') {
27. return -1;
28. }
30. return 1;
31. }
32. inline char \*strcpy(char \*dest, const char \*src)
33. {
34. char \*tmp = dest;
35. while (\*src) {
36. \*dest++ = \*src++;
37. }
38. \*dest = '\0';
40. return tmp;
41. }
42. inline char \*strcat(char \*dest, const char \*src)
43. {
44. char \*cp = dest;
45. while (\*cp) {
46. cp++;
47. }
48. while ((\*cp++ = \*src++))
49. ;
50. return dest;
51. }
52. inline int strlen(const char \*src)
53. {
54. const char \*eos = src;
55. while (\*eos++)
56. ;
58. return (eos - src - 1);
59. }

这部分代码并不复杂，但基本都用指针操作，所以自己写的话需要小心不要写错。

接下来，我们实现一个比较复杂的函数，printf()，这个函数在C语言的学习中几乎是使用最多的一个函数了，它不仅方便使用而且功能强大，能够做各种类型的输出。但它的实现涉及到C语言中比较复杂的一个内容，就是可变参函数，也就是说它的参数是可变的。为什么呢，因为我们想一想，我们在用printf()的时候，里面要输出的内容数量是任意的。所以就和传统的固定参数的函数有所不同。

对于可变参函数，它的声明如下：void printk(char \*str, ...);可以看出来，它的参数有一个char \*str,和...。其中...就表示了未知数量的参数。但未知数量在实际函数运行中，也是需要确定的，而对于这个内容的确定，我们用宏函数来实现。先看看代码吧：

/include/vargs.h

1. #ifndef INCLUDE\_VARGS\_H\_
2. #define INCLUDE\_VARGS\_H\_
3. #include "types.h"
4. typedef int8\_t \* va\_list;
5. #define va\_start(ap,v) ( ap = (va\_list)\_ADDRESSOF(v) + \_INTSIZEOF(v) )
6. #define va\_arg(ap,t) ( \*(t \*)((ap += \_INTSIZEOF(t)) - \_INTSIZEOF(t)) )
7. #define va\_end(ap) ( ap = (va\_list)0 )
8. #define \_ADDRESSOF(v) ( &(v) )
9. #define \_INTSIZEOF(n) ( (sizeof(n) + sizeof(int) - 1) & ~(sizeof(int) - 1) )
10. #endif

这里定义的宏函数，就是后面用到的对不确定的参数的操作。\_ADDRESSOF(v)是取了v的地址，这个很容易看出来，不多说了。\_INTSIZEOF(n)是把n对齐到大于等于sizeof(int)的边界。这样说明之后，不知道能不能想到可变长参数的用法。

可变长参数是这样实现的：给出了第一个参数的地址，然后按照我们输入的参数的类型去寻找对于大小的地址，来读出我们输入的参数值。

\_INTSIZEOF(n)就是一个能找出n的对于int类型对齐的长度的函数。实现想法是这样的：

如果要把a对齐到最近的大于等于b的边界，应该这样做：

1. 首先找出a的长度相当于b的长度的倍数，也就是向上舍入。
2. 乘以b的长度。

第一步实现：q = (a + b - 1) / b 这里的/是C语言中的/，也就是向下舍入的除号。

第二步实现：aligned\_a = (a + b - 1) / b \* b。这样就可以了。

我们把这个简单的思想用到我们的\_INTSIZEOF(n)函数中，应该这样：

#define \_INTSIZEOF(n)\

( (sizeof(n) + sizeof(int) - 1) / sizeof(int) \* sizeof(int) )

这样就足够好了。然而由于我们的对齐都是2的幂，所以可以用一个掩码(mask)来优化一下，就有这样：

aligned\_a = (a + b - 1) & ~(b - 1)

所以就能明白这个函数的作用了。如果还有疑问，可以参考后面的参考资料，有更详细的解释。

宏函数va\_start(ap,v) 可以使ap指向n后面的可变参数表中的第一个参数。

va\_arg(ap,t) 返回一个t类型的值，并且把ap更新，指向下一个参数

va\_end(ap) 把ap指空，防止后续误调用产生错误。

好了，有了这几个宏函数了，我们看一下printf的实现吧，这里我把它命名为了printk()，因为它相比起printf()还是有些缺陷与不足的。

/kernel/print.c

1. #include "print.h"
2. #include "console.h"
3. #include "types.h"
4. #include "vargs.h"
5. //进制转换，在一个长度为50的栈里转换，所以这里有问题隐患，不是吗。
6. char \*convert(unsigned int num, int base)
7. {
8. static char Representation[]= "0123456789ABCDEF";
9. static char buffer[50];
10. char \*ptr;
11. ptr = &buffer[49];
12. \*ptr = '\0';
13. do
14. {
15. \*--ptr = Representation[num%base];
16. num /= base;
17. }while(num != 0);
18. return(ptr);
19. }
20. //printk的实现
21. void printk(char \*str, ...)
22. {
23. va\_list arg; //声明一个va\_list类型的变量，在宏定义里有说明
24. va\_start(arg, str); //把printk()第一个参数取出到str中
25. char \*temp; //用一个字符指针来作为我们的字符输出
26. int i;
27. for(temp=str;\*temp;temp++) //对我们的字符进行遍历
28. {
29. if(\*temp!='%')
30. putchar(\*temp); //如果字符是正常字符，则输出
31. else{ //当字符不是正常字符，而是用%标记的特定输出
32. temp++; //加一，取到%的下一个字符
33. switch(\*temp){ //对%标记做讨论
34. case 'd':
35. i=va\_arg(arg,int); //将取到的参数按照int型赋值给i
36. if(i<0) //对i转换为10进制，讨论正负，输出
37. {
38. putchar('-');
39. console\_write(convert(-i,10));
40. }
41. else
42. console\_write(convert(i,10));
43. break;
44. case 'c':
45. i=va\_arg(arg,int); //将取到的参数按照int型赋值给i
46. putchar(i); //对i对于的ASCII码输出
47. break;
48. case 'o':
49. i=va\_arg(arg,unsigned int); //将取到的参数按照unsigned int型赋值给i
50. console\_write(convert(i,8)); //把i转换为8进制输出
51. break;
52. case 's':
53. i=(int)va\_arg(arg, char\*); //将取到的参数按照char\*型赋值给i
54. console\_write((char\*)i);
55. break;
56. case 'x':
57. i=va\_arg(arg,unsigned int); //将取到的参数按照unsigned int型赋值给i
58. console\_write(convert(i,16)); break;
59. case 'b':
60. i=va\_arg(arg,int); //将取到的参数按照int型赋值给i
61. console\_write(convert(i,2));
62. break;
63. }
64. }
65. }
66. va\_end(arg); //结束我们的arg，防止后序误调用引发错误
67. }

参考着注释，相信理解起来也并不是那么困难。为什么可以这样实现呢，是因为根据X86的调用约定，在函数调用的时候，都是先把参数压到栈里，然后调用函数的。所以我们只要知道了栈的地址，就能按照地址一个一个来取出来我们压入的参数了。

X86调用约定

在x86芯片架构上的调用约定(calling conventions)。 调用约定描述了被调用代码的接口：

1. 原子(标量)参数，或复杂参数独立部分的分配顺序;
2. 参数是如何被传递的(放置在堆栈上，或是寄存器中，亦或两者混合);
3. 被调用者应保存调用者的哪个寄存器;
4. 调用函数时如何为任务准备堆栈，以及任务完成如何恢复;

cdecl

cdecl(C declaration，即C声明)是源起[C语言](https://zh.wikipedia.org/wiki/C%E8%AF%AD%E8%A8%80)的一种调用约定，也是C语言的事实上的标准。在**x86架构**上，其内容包括：

1. 函数实参在线程栈上按照从右至左的顺序依次压栈。
2. 函数结果保存在寄存器EAX/AX/AL中
3. 浮点型结果存放在寄存器ST0中
4. 编译后的函数名前缀以一个[下划线字符](https://zh.wikipedia.org/wiki/%E4%B8%8B%E5%88%92%E7%BA%BF)
5. 调用者负责从线程栈中弹出实参（即清栈）
6. 8比特或者16比特长的整形实参提升为32比特长。
7. 受到函数调用影响的寄存器（volatile registers）：EAX, ECX, EDX, ST0 - ST7, ES, GS
8. 不受函数调用影响的寄存器： EBX, EBP, ESP, EDI, ESI, CS, DS
9. RET指令从函数被调用者返回到调用者（实质上是读取寄存器EBP所指的线程栈之处保存的函数返回地址并加载到IP寄存器）

举例来说，如果有这样一个函数func(1, 2, 3, 4)，那么在汇编中，它的执行就是

1. push 4
2. push 3
3. push 2
4. push 1
5. call func
6. sub esp, 16

到这里，终于实现了我们非常有用的prink了，它的存在，会给我们后面的工作带来极大的便利。

最后列一下我们的头文件：

/include/print.h

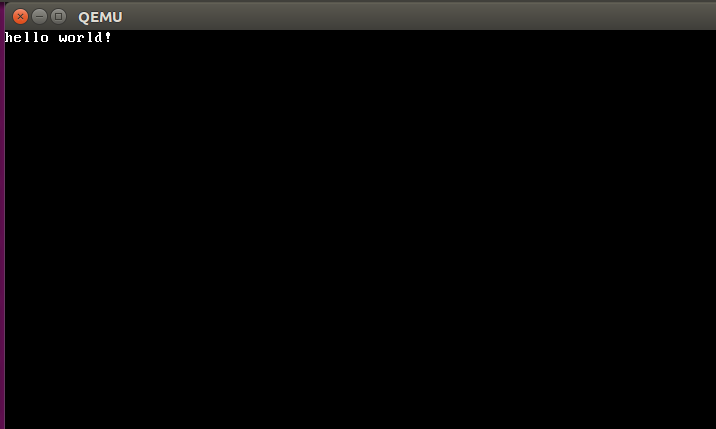
1. #ifndef INCLUDE\_DEBUG\_H
2. #define INCLUDE\_DEBUG\_H
3. #include "types.h"
4. //进制转换
5. char \*convert(uint32\_t num, int32\_t base);
6. // 内核的打印函数
7. void printk(char \*str, ...);
8. #endif

我们修改一下entry.c的内容，看下输出：

/entry.c

1. #include"console.h"
2. #include"print.h"
3. int kern\_entry()
4. {
5. console\_clear();
6. printk("hello world!\n");
7. return 0;
8. }

输出：



有了printk()的实现，我们后续的程序设计将会如虎添翼。但人无完人，相信在实践过程中，或多或少都会遇到一些难以解决的问题。如果是在IDE中，我们可以用一些调试器实现代码的调试，而在这里，我们能否用gdb这类调试软件呢。答案当然是可以的。

我们实现的方法是使用qemu联合gdb，来进行C语言源代码的调试。

首先是通讯问题，因为qemu和gdb运行的时候毕竟是两个进程，数据交换必然涉及到进程间通信机制。所幸它们都支持一个标准的调试协议，而且开启的方法都很简单。qemu使用以下命令启动即可：

qemu -S -s -fda floppy.img -boot a

这几个参数中 -fda floppy.img 和 -boot a 是指定启动的镜像，-s 这个参数指的  
是启动时开启1234端口等待gdb连接（这个参数从字面上看比较隐晦），-S 是指是启动时不自动开始运行，等待调试器的执行命令。以调试模式启动了虚拟机之后，再启动gdb。需要注意的是，此时的gdb没有内核程序的符号文件，没有办法进行代码级调试。解决的办法很简单，我们使用命令加载待调试内核对应的可执行文件即可。 启动了gdb之后，我们依次执行以下指令即可：

file jf\_kernel  
target remote :1234

这几个命令的意思分别是加载待调试文件的符号信息；连接本地的1234端口。然后我们就可以用break命令来设置断点，用continue来开始执行。

方便起见，我把这两个命令写到了一个脚本文件中：

/script/gdbinit

1. file jf\_kernel
2. target remote :1234

在我们的Makefile里，调试在debug里面，命令如下：

.PHONY:debug

debug:

qemu -S -s -fda floppy.img -boot a &

sleep 1

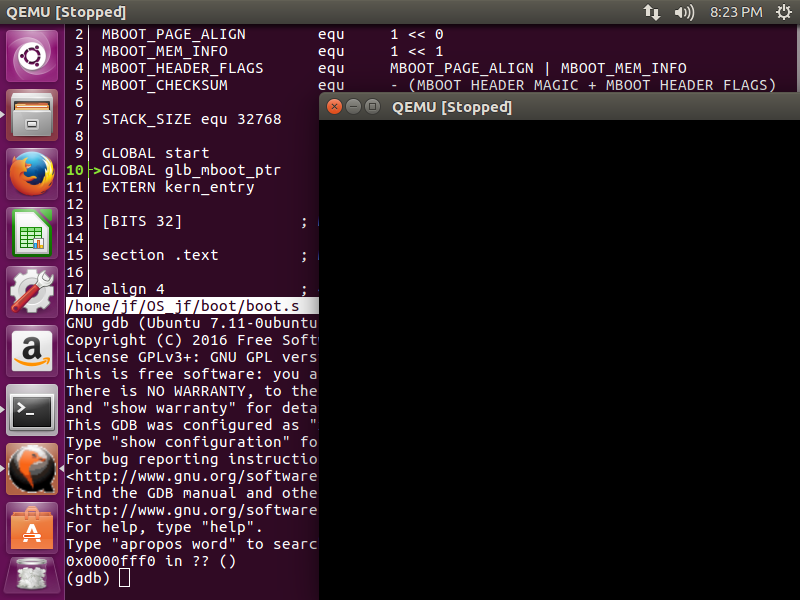
cgdb -x scripts/gdbinit

可以看到这里用的是cgdb，cgdb是GNU Debugger（GDB）的轻量级curses（基于终端）接口。 除了标准的gdb控制台之外，cgdb还提供了一个拆分屏幕视图，在执行时显示源代码。

如果用的不舒服，可以换成gdb就行。

在终端中，执行make后，执行make debug就可以开启调试

调试界面：



当前目录：

jf@ubuntu:~/OS\_jf$ tree

.

├── boot

│   └── boot.s

├── drivers

│   └── console.c

├── entry.c

├── floppy.img

├── include

│   ├── common.h

│   ├── console.h

│   ├── print.h

│   ├── string.h

│   ├── types.h

│   └── vargs.h

├── kernel

│   └── print.c

├── libs

│   ├── common.c

│   └── string.c

├── Makefile

└── scripts

├── gdbinit

└── kernel.ld

6 directories, 16 files

http://stackoverflow.com/questions/14175748/an-implement-of-sizeof-guaranteeing-bit-alignment

http://blog.chinaunix.net/uid-15014334-id-4691200.html

<http://blog.csdn.net/hackbuteer1/article/details/7558979>

https://zh.wikipedia.org/wiki/X86%E8%B0%83%E7%94%A8%E7%BA%A6%E5%AE%9A