linux内核的移植性非常好, 目前的内核也支持非常多的体系结构(有20多个).

但是刚开始时, linux也只支持 intel i386 架构, 从 v1.2版开始支持 Digital Alpha, Intel x86, MIPS和SPARC(虽然支持的还不是很完善).

从 v2.0版本开始加入了对 Motorala 68K和PowerPC的官方支持, v2.2版本开始新增了 ARMS, IBM S390和UltraSPARC的支持.

v2.4版本支持的体系结构数达到了15个, v2.6版本支持的体系结构数目提高到了21个.

目前的我使用的系统是 Fedora20, 支持的体系结构有31个之多.(源码树中 arch目录下有支持的体系结构, 每种体系结构一个文件夹)

考虑到内核支持如此之多的架构, 在内核开发的时候就需要考虑编码的可移植性.

提高可移植性最重要的就是要搞明白不同体系结构之间究竟是什么对移植代码的影响比较大.

**主要内容：**

* 字长
* 数据类型
* 数据对齐
* 字节顺序
* 时间
* 页长度
* 处理器顺序
* SMP, 内核抢占, 高端内存
* 总结

### ****1. 字长****

这里的字是指处理器能够一次完成处理的数据. 字长即使处理器能够一次完成处理的数据的最大长度.

目前的处理器主要有32位和64为2种, 注意这里的32位和64位并不是指操作系统的版本, 而是指处理器的能力.

一般来说, 32位的处理器只能安装32位的操作系统, 而64位的处理器可以安装32位的操作系统, 也可以安装64位的操作系统.

对于一种体系结构来说, 处理器通用寄存器(general-purpose registers, GPR)的大小和它的字长是相同的.

C语言定义的long类型总是对等于机器的字长, 而int型有时会比字长小.

1. 32位的体系结构中, int型和long型都是32位的
2. 64位的体系结构中, int型是32位的, long型是64位的.

内核编码中涉及到字长的部分时, 牢记以下准则:

1. ANSI C标准规定, 一个char的长度一定是一个字节(8位)
2. linux当前所支持的体系结构中, int型都是32位的
3. linux当前所支持的体系结构中, short型都是16位的
4. linux当前所支持的体系结构中, 指针和long型的长度不定, 在32位和64位中变化
5. 不能假设 sizeof(int) == sizeof(long)
6. 类似的, 不能假定 指针的长度和int型相同.

此外, 操作系统有个简单的助记符来描述此系统中数据类型的大小.

* LLP64 :: 64位的Windows, long类型和指针都是64位
* LP64 :: 64位的Linux, long类型和指针都是64位
* ILP32 :: 32位的Linux, int类型, long类型和指针都是32位
* ILP64 :: int类型, long类型和指针都是64位(非Linux)

### ****2. 数据类型****

编写可移植性代码时, 内核中的数据类型有以下3点需要注意:

#### **2.1 不透明类型**

linux内核中定义了很多不透明类型, 它们是在C语言标准类型上的一个封装, 比如 pid\_t, uid\_t, gid\_t 等等.

例如, pid\_t的定义可以在源码中找到:

typedef \_\_kernel\_pid\_t pid\_t; /\* include/linux/types.h \*/

typedef int \_\_kernel\_pid\_t; /\* arch/asm/include/asm/posix\_types.h \*/

使用这些不透明类型时, 以下原则需要注意:

1. 不要假设该类型的长度(那怕通过源码看到了它的C语言类型), 这些类型在不同体系结构中可能长度会变, 内核开发者也有可能修改它们
2. 不要将这些不透明类型转换为C标准类型来使用
3. 编程时保证不透明类型实际存储空间或者格式发生变化时代码不受影响

#### **2.2 长度确定的类型**

除了不透明类型, linux内核中还定义了一系列长度明确的数据类型, 参见 include/asm-generic/int-l64.h 或者 include/asm-generic/int-ll64.h

typedef signed char s8;

typedef unsigned char u8;

typedef signed short s16;

typedef unsigned short u16;

typedef signed int s32;

typedef unsigned int u32;

typedef signed long s64;

typedef unsigned long u64;

上面这些类型只能在内核空间使用, 用户空间无法使用. 用户空间有对应的变量类型, 名称前多了2个下划线:

typedef \_\_signed\_\_ char \_\_s8;

typedef unsigned char \_\_u8;

typedef \_\_signed\_\_ short \_\_s16;

typedef unsigned short \_\_u16;

typedef \_\_signed\_\_ int \_\_s32;

typedef unsigned int \_\_u32;

typedef \_\_signed\_\_ long \_\_s64;

typedef unsigned long \_\_u64;

#### **2.3 char类型**

之所以把char类型单独拿出来说明, 是因为char类型在不同的体系结构中, 有时默认是带符号的, 有时是不带符号的.

比如, 最简单的例子:

/\*

\* 某些体系结构中, char类型默认是带符号的, 那么下面 i 的值就为 -1

\* 某些体系结构中, char类型默认是不带符号的, 那么下面 i 的值就为 255, 与预期可能有差别!!!

\*/

char i = -1;

避免上述问题的方法就是, 给char类型赋值时, 明确是否带符号, 如下:

signed char i = -1; /\* 明确 signed, i 的值在哪种体系结构中都是 -1 \*/

unsigned char i = 255; /\* 明确 unsigned, i 的值在哪种体系结构中都是 255 \*/

### ****3. 数据对齐****

数据对齐也是增强可移植性的一个重要方面(有的体系结构对数据对齐要求非常严格, 载入未对齐的数据可导致性能下降, 甚至错误).

数据对齐的意思就是: 数据的内存地址可以被 4 整除

1. 通过指针转换类型时, 不要转换长度不一样的类型, 比如下面的代码有可能出错

/\*

\* 下面的代码将一个变量从 char 类型转换为 unsigned long 类型,

\* char 类型只占 1个字节, 它的地址不一定能被4整除, 转换为 4个字节或者8个字节的 usigned long之后,

\* 导致 unsigned long 出现数据不对齐的现象.

\*/

char wolf[] = "Like a wolf";

char \*p = &wolf[1];

unsigned long p1 = \*(unsigned long\*) p;

2. 对于数组, 安装基本数据类型进行对齐就行.(数组元素的存放在内存中是连续的, 第一个对齐了, 后面的都自动对齐了)

3. 对于联合体, 长度最大的数据对齐就可以了

4. 对于结构体, 保证结构体中每个元素能够正确对齐即可

如果结构体中的元素没有对齐, 编译器会自动填充结构体, 保证它是对齐的. 比如下面的代码, 预计应该输出**12**, 实际却输出了**24**

我的代码运行环境: Fedora20 x86\_64

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* @file : struct\_align.c

\* @author : wangyubin

\* @date : 2014-01-09

\*

\* @brief :

\* history : init

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <stdio.h>

struct animal\_struct

{

char dog; /\* 1个字节 \*/

unsigned long cat; /\* 8个字节 \*/

unsigned short pig; /\* 2个字节 \*/

char fox; /\* 1个字节 \*/

};

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

{

/\* 在我的64bit 系统中是按8位对齐, 下面的代码输出 24 \*/

printf ("sizeof(animal\_struct)=%d\n", sizeof(struct animal\_struct));

return 0;

}

测试方法:

gcc -o test struct\_align.c

./test # 输出24

结构体应该被填充成如下形式:

struct animal\_struct

{

char dog; /\* 1个字节 \*/

/\* 此处填充了7个字节 \*/

unsigned long cat; /\* 8个字节 \*/

unsigned short pig; /\* 2个字节 \*/

char fox; /\* 1个字节 \*/

/\* 此处填充了5个字节 \*/

};

通过调整结构体中元素顺序, 可以减少填充的字节数, 比如上述结构体如果定义成如下顺序:

struct animal\_struct

{

unsigned long cat; /\* 8个字节 \*/

unsigned short pig; /\* 2个字节 \*/

char dog; /\* 1个字节 \*/

char fox; /\* 1个字节 \*/

};

那么为了保证8位对齐, 只需在后面补充 4位即可:

struct animal\_struct

{

unsigned long cat; /\* 8个字节 \*/

unsigned short pig; /\* 2个字节 \*/

char dog; /\* 1个字节 \*/

char fox; /\* 1个字节 \*/

/\* 此处填充了4个字节 \*/

};

调整后的代码会输出 16, 不是之前的24

#include <stdio.h>

struct animal\_struct

{

unsigned long cat; /\* 8个字节 \*/

unsigned short pig; /\* 2个字节 \*/

char dog; /\* 1个字节 \*/

char fox; /\* 1个字节 \*/

};

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

{

/\* 在我的64bit 系统中是按8位对齐, 下面的代码输出 16 \*/

printf ("sizeof(animal\_struct)=%d\n", sizeof(struct animal\_struct));

return 0;

}

测试方法:

gcc -o test struct\_align.c

./test # 输出16

**注意:** 虽然调整结构体中元素的顺序可以减少填充的字节, 从而降低内存的消耗.

但是对于内核中已有的那些结构, 千万不能随便调整其元素顺序, 因为内核中很多现存的方法都是通过元素在结构体中位置偏移来获取元素的.

### ****4. 字节顺序****

字节顺序其实只有2种:

* 低位优先 :: little-endian 数据由低位地址->高位地址存放
* 高位优先 :: big-endian 数据由高位地址->低位地址存放

比如占有四个字节的整数的二进制表示如下:

00000001 00000002 00000003 00000004

内存地址方向:   高位  <--------------------> 低位

little-endian 表示如下:

00000001 00000002 00000003 00000004

big-endian 表示如下:

00000004 00000003 00000002 00000001

判断一个体系结构是 big-endian 还是 little-endian 非常简单.

int x = 1; /\* 二进制 00000000 00000000 00000000 00000001 \*/

/\*

\* 内存地址方向: 高位 <--------------------> 低位

\* little-endian 表示: 00000000 00000000 00000000 00000001

\* big-endian 表示: 00000001 00000000 00000000 00000000

\*/

if (\*(char \*) &x == 1) /\* 这句话把int型转为char型, 相当于只取了int型的最低8bit \*/

/\* little-endian \*/

else

/\* big-endian \*/

### ****5. 时间****

内核中使用到时间相关概念时, 为了提高可移植性, 不要使用时间中断的发生频率(也就是每秒产生的jiffies), 而应该使用 HZ 来正确使用时间.

关于 jiffies 和 HZ 的概念, 可以参考之前的博客: [《Linux内核设计与实现》读书笔记（十一）- 定时器和时间管理](http://www.cnblogs.com/wang_yb/archive/2013/05/10/3070373.html" \t "https://www.cnblogs.com/wang_yb/p/_blank)

### ****6. 页长度****

当处理用页管理的内存时, 不要既定页的长度为 4KB, 在不同的体系结构中长度会不一样.

而应该使用 **PAGE\_SIZE** 以字节数来表示页长度, 使用 **PAGE\_SHIFT** 表示从最右端屏蔽了多少位能够得到该地址对应的页的页号.

**PAGE\_SIZE** 和 **PAGE\_SHIFT** 都是宏, 定义在 include/asm-generic/page.h 中

下表是一些体系结构中页长度:

|  |  |  |
| --- | --- | --- |
| **体系结构** | **PAGE\_SHIFT** | **PAGE\_SIZE** |
| alpha | 13 | 8KB |
| arm | 12, 14, 15 | 4KB, 16KB, 32KB |
| avr | 12 | 4KB |
| cris | 13 | 8KB |
| blackfin | 12 | 16KB |
| h8300 | 14 | 4KB |
|  | 12 | 4KB, 8KB, 16KB, 32KB |
| m32r | 12, 13, 14, 16 | 4KB |
| m68k | 12 | 4KB, 8KB |
| m68knommu | 12, 13 | 4KB |
| mips | 12 | 4KB |
| min10300 | 12 | 4KB |
| parisc | 12 | 4KB |
| powerpc | 12 | 4KB |
| s390 | 12 | 4KB |
| sh | 12 | 4KB |
| sparc | 12, 13 | 4KB, 8KB |
| um | 12 | 4KB |
| x86 | 12 | 4KB |
| xtensa | 12 | 4KB |

### ****7. 处理器顺序****

还有最后一个和可移植性相关的注意点就是处理器对代码的执行顺序, 在有些体系结构中, 处理器并不是严格按照代码编写的顺序执行的,

可能为了优化性能或者其他原因, 处理器执行指令的顺序与编写的代码的顺序稍有出入.

如果我们的某段代码需要严格的执行顺序, 需要在代码中使用 rmb() wmb() 等内存屏障来确保处理器的执行顺序.

关于rmb和wmb可以参考之前的博客: [《Linux内核设计与实现》读书笔记（十）- 内核同步方法](http://www.cnblogs.com/wang_yb/archive/2013/05/01/3052865.html" \t "https://www.cnblogs.com/wang_yb/p/_blank)  第 11 小节

### ****8. SMP, 内核抢占, 高端内存****

SMP, 内核抢占和高端内存本身虽然和可移植性没有太大的关系, 但它们都是内核中重要的配置选项,

如果编码时能够考虑到这些的话, 那么即使内核修改SMP等这些配置选项, 我们的代码仍然可以安全可靠的运行.

所以, 在编写内核代码时最好加上如下假设:

* 假设代码会在SMP系统上运行, 要正确选择和使用锁
* 假设代码会在支持内核抢占的情况下运行, 要正确使用锁和内核抢占语句
* 假设代码会运行在使用高端内存(非永久映射内存)的系统上, 必要时使用 kmap()

### ****9. 总结****

编写简洁, 可移植性的代码还需要通过实践来积累经验, 上面的准则可以作为代码是否满足可移植性的一些检测条件.

书中还提到的2点注意事项, 我觉得不仅是编写内核代码, 编写任何代码时, 都应该注意:

* 编码尽量选取最大公因子 :: 假定任何事情都有可能发生, 任何潜在的约束也都存在
* 编码尽量选取最小公约数 :: 不要假定给定的内核特性是可用的, 仅仅需要最小的体系结构功能