笔记

# x86硬件

# 软件

## 编译器和语言

### GCC加内存barrier

// 通过此种方式可以阻止GCC编译时为了优化而颠倒该代码前后的内存访问顺序

#define barrier() \_\_asm\_\_ \_\_volatile\_\_("": : :"memory")

### 分支预测

#define likely(x) \_\_builtin\_expect(!!(x), 1)

#define unlikely(x) \_\_builtin\_expect(!!(x), 0)

如果条件多数情况下为TRUE则使用likely，否则使用unlikely。这样有助于编译器将可能的分支最优化。(这里可能涉及到流水线，具体情况参见GCC)

### 汇编中连接符

#define STR(x) (#x)

#define \_\_KERNEL\_DS 0x18

"movl $" STR(\_\_KERNEL\_DS) ",%edx\n\t"

// 预处理之后

movl $0x18, %edx

如果连接的是原文，则用两个#

如果连接的是宏，则用一个#

### 内嵌汇编

#define spin\_lock\_string \

"\n1:\t" \

"lock ; decb %0\n\t" \

"js 2f\n" \

".section .text.lock,\"ax\"\n" \

"2:\t" \

"cmpb $0,%0\n\t" \

"rep;nop\n\t" \

"jle 2b\n\t" \

"jmp 1b\n" \

".previous"

%0代表输入输出列表中第一个，%b0则是使用列表中第一个的低8bit。

另外.section和.previous配套使用可以将一部分代码从当前段中摘出到一个指定的段中。

## GCC Tools Chain

### 静态库

Linux中将静态库称为archive，即所谓的归档文件，后缀为.a，它实际是对一系列.o文件的打包。对于创建静态库，使用ar工具。例如目前有lib1.o和lib2.o，我们可以使用如下的命令：

xxx@xxx:~/ar cr libtest.a lib1.o lib2.o

### 动态库

Linux将动态库称为共享库，后缀为.so。对于创建动态库使用ld(gcc当然也可以)。例如目前有lib1.o和lib2.o,我们可以使用如下的命令:

xxx@xxx:~/ld lib1.o lib2.o -o libtest.so -shared –fPIC

其中-fPIC指明生成地址无关代码(Position-Independent Code)，而-shared只生成共享库。

### 使用库

在gcc创建可执行文件时，使用-l参数来指定使用什么库，这里需要注意的有两点

1. “-l”后紧接着就是库的名称，即-lnameoflib
2. 静态库或者动态库的文件名是在实际库名称前面加上了lib的，即libnameoflib

举例说明，例如前面我们已经创建了一个libtest.a的静态库，则使用-ltest来指明这个库，如果同时存在静态库libtest.a和动态库libtest.so，则ld会默认使用动态库，除非使用参数-static显式地指明全部使用静态库，但是如果这么做则所有的库全部以静态方式链入，不再使用任何动态库。

例如存在一个main.c，需要编译并静态链接libtest.a的命令如下

xxx@xxx:~/gcc -o test main.c –ltest -static

这里需要指出的是，gcc根据-l参数自动查找库，而libtest.a目前尚不在gcc的搜索路径中，可以有两种选择，第一通过增加一个参数-L.即把.这个路径增加到搜索路径中，从而使得链接器能够找到这个libtest.a，第二通过在环境变量LIBRARY\_PATH中添加libtest.a所在的路径。同样的问题也出现在使用动态库的情况下，也可以通过上面提到的两种方法解决。

另外在执行使用动态库的程序时，也需要保证所用的动态库的路径在环境变量LD\_LIBRARY\_PATH中。添加环境变量可以通过如下的指令：

xxx@xxx:~/export LD\_LIBRARY\_PATH=$(LD\_LIBRARY\_PATH)+*path*

## x86硬件相关数据结构

### MMU

#ifdef CONFIG\_X86\_PAE

extern unsigned long long \_\_supported\_pte\_mask;

typedef struct { unsigned long pte\_low, pte\_high; } pte\_t;

typedef struct { unsigned long long pmd; } pmd\_t;

typedef struct { unsigned long long pgd; } pgd\_t;

typedef struct { unsigned long long pgprot; } pgprot\_t;

#define pmd\_val(x) ((x).pmd)

#define pte\_val(x) ((x).pte\_low | ((unsigned long long)(x).pte\_high << 32))

#define \_\_pmd(x) ((pmd\_t) { (x) } )

#define HPAGE\_SHIFT 21

#include <asm-generic/pgtable-nopud.h>

#else

typedef struct { unsigned long pte\_low; } pte\_t; // phy addr

typedef struct { unsigned long pgd; } pgd\_t;

typedef struct { unsigned long pgprot; } pgprot\_t;// protect flag

#define boot\_pte\_t pte\_t /\* or would you rather have a typedef \*/

#define pte\_val(x) ((x).pte\_low)

#define HPAGE\_SHIFT 22

#include <asm-generic/pgtable-nopmd.h>

#endif

## 物理页面管理

### pg\_data\_t结构

typedef struct pglist\_data{}pg\_data\_t;

NUMA(None-Uniform Memory Architecture)模式

### page结构

struct page{};

struct page mem\_map; // 页面管理数组 global

每个页面由一个对应的page结构

### zone结构

struct zone{};

struct free\_area {

struct list\_head free\_list;

unsigned long nr\_free;

};

每个zone管理一块连续RAM，内部通过一个free\_area链表将page链在一起。

## 虚拟空间管理

按照任务的User空间来总领虚拟空间，由于虚拟空间中的系统空间部分实际是共用的，只有用户部分是按照各个任务不同的。

### vm\_area\_struct结构

struct vm\_area\_struct{};

指定一块虚存区域，每个任务可能被划分成不同的连续虚存区域。按照区域首地址顺序排列。其中包括一个mm\_struct结构的指针，该结构就是统一管理一个task中的所有虚存空间的结构。

### mm\_struct结构

struct mm\_struct{

struct vm\_area\_struct \* mmap; /\* list of VMAs \*/

struct rb\_root mm\_rb;

struct vm\_area\_struct \* mmap\_cache; /\* last find\_vma result \*/

.

.

pgd\_t \* pgd;

atomic\_t mm\_users;

atomic\_t mm\_count;

.

.

}；

每个task会有一个mm\_struct结构，在任务控制块中指出，这里可能出现多个task共享一个mm\_struct结构的情况，例如fork一个task，则childtask实际是与parenttask共享mm\_struct结构的。这里由mm\_users和mm\_count指出。

mmap是vma的顺序结构首地址，mm\_rb是vma红黑树结构的根节点，这两种方式是根据vma个数来决定使用哪一个。mmap\_cache则是最后一个找到的vma结构指针。

pgd指明页目录，实际每个mm\_struct都有一个页目录表。

## 中断管理

中断处理分为两部分，前部分在关中断情况下处理，CPU进中断门并最终进入do\_IRQ()函数，直到发出响应中断并最终重新开中断。这个过程实际并没有处理中断，而是紧接着由do\_IRQ()后续的一个循环来处理实际的中断部分，这里通过pandding机制将中断串行化了。第二部分则是do\_IRQ()函数尾部的处理软中断过程，这个过程发生在完整的硬件响应中断之后，是完全是开中断情况下完成的。

一下放上有关软中断的若干数据结构和全局变量代码，备查

struct tasklet\_struct

{

struct tasklet\_struct \*next;

unsigned long state;

atomic\_t count;

void (\*func)(unsigned long);

unsigned long data;

};

struct tasklet\_struct bh\_task\_vec[32];

// 该结构体数组在初始化时将每个func域挂载了bh\_action()函数

static void (\*bh\_base[32])(void);

// 同时每个bh对应bh\_base[]中的一项函数指针，作为这个bh的处理函数

注意到tasklet\_struct实际是相对于之前的bh机制而言更加宽松的一种机制，允许在多个CPU上并行执行不同的tasklet，而bh在此基础上更加严格，即任何时间只能有一个CPU处理bh.。32个bh\_task\_vec结构体，每一个带表一个待执行的bh队列。

而在bh之上又做了一层更高机制叫做softirq，最多支持32种软中断，目前存在4种。

enum

{

HI\_SOFTIRQ=0,

NET\_TX\_SOFTIRQ,

NET\_RX\_SOFTIRQ,

TASKLET\_SOFTIRQ

};

而其中的HI\_SOFTIRQ就是针对bh机制的，即将bh机制融合在了软中断机制中。定义数组softirq\_vec[32]分别对应各种不同的软中断类型，尽管目前只有前4项有效。而每一项会有一个队列用于记录需要执行的tasklet\_struct结构体的队列，其中HI\_SOFTIRQ类软中断对应的执行队列就是tasklet\_hi\_vec。

struct softirq\_action

{

void (\*action)(struct softirq\_action \*);

void \*data;

};

static struct softirq\_action softirq\_vec[32] \_\_cacheline\_aligned;

struct tasklet\_head

{

struct tasklet\_struct \*list;

} \_\_attribute\_\_ ((\_\_aligned\_\_(SMP\_CACHE\_BYTES)));

extern struct tasklet\_head tasklet\_vec[NR\_CPUS];

extern struct tasklet\_head tasklet\_hi\_vec[NR\_CPUS];

对于bh而言通过mark\_bh()函数(其实内部是通过tasklet\_hi\_schedule()函数实现的)的调用向tasklet\_hi\_vec队列中插入一个tasklet\_struct结构体等待执行，共有15种不同的bh结构，一旦调用mark\_bh()将其挂入tasklet\_hi\_vec则不能再次挂入。

enum {

TIMER\_BH = 0,

TQUEUE\_BH,

DIGI\_BH,

SERIAL\_BH,

RISCOM8\_BH,

SPECIALIX\_BH,

AURORA\_BH,

ESP\_BH,

SCSI\_BH,

IMMEDIATE\_BH,

CYCLADES\_BH,

CM206\_BH,

JS\_BH,

MACSERIAL\_BH,

ISICOM\_BH

};

每次do\_softirq()函数执行会遍历softirq\_vec[]，凡是存在待执行的软中断的都会调用对应软中断大类的处理函数(softirq\_action结构体中的函数指针action)，针对HI\_SOFTIRQ的处理函数就是tasklet\_hi\_action()，该函数负责遍历tasklet\_hi\_vec中的所有tasklet\_struct结构体，即通过mark\_bh()挂入的代表每个hb的结构体，并调用其中的func域函数指针进行具体的bh处理操作。

## 任务切换

系统调用和中断服务基本是一致的

#define RESTORE\_ALL \

popl %ebx; \

popl %ecx; \

popl %edx; \

popl %esi; \

popl %edi; \

popl %ebp; \

popl %eax; \

1: popl %ds; \

2: popl %es; \

addl $4,%esp; \

3: iret; \

注意到返回值放在栈中的eax位置，然后最后popl出来放回eax最后iret返回ring3，系统调用的返回值就在eax中。

ENTRY(system\_call)

pushl %eax # save orig\_eax

SAVE\_ALL

GET\_CURRENT(%ebx)

cmpl $(NR\_syscalls),%eax

jae badsys

testb $0x02,tsk\_ptrace(%ebx) # PT\_TRACESYS

jne tracesys

call \*SYMBOL\_NAME(sys\_call\_table)(,%eax,4)

movl %eax,EAX(%esp) # save the return value

ENTRY(ret\_from\_sys\_call)

#ifdef CONFIG\_SMP

movl processor(%ebx),%eax

shll $CONFIG\_X86\_L1\_CACHE\_SHIFT,%eax

movl SYMBOL\_NAME(irq\_stat)(,%eax),%ecx # softirq\_active

testl SYMBOL\_NAME(irq\_stat)+4(,%eax),%ecx # softirq\_mask

#else

movl SYMBOL\_NAME(irq\_stat),%ecx # softirq\_active

testl SYMBOL\_NAME(irq\_stat)+4,%ecx # softirq\_mask

#endif

jne handle\_softirq

ret\_with\_reschedule:

cmpl $0,need\_resched(%ebx)

jne reschedule

cmpl $0,sigpending(%ebx)

jne signal\_return

restore\_all:

RESTORE\_ALL

# Linux使用笔记

## swap交换区

swap本身可以以一个文件的形式存在(对于交换分区本身而言就是/dev下的一个设备)，可以使用swapon或者swapoff打开或者关闭一个交换文件，前提是这个文件被格式化为一个swap设备，使用mkswap指令可以将一个文件格式化为一个交换文件。/proc/swaps这个虚拟文件中记录了所以系统中的交换文件。

## 磁盘分区

参见《Linux内核完全注释》p336~p338

## 建立文件系统

在一个已有的设备上建立文件系统，实际上

# 待查问题

目前已解决的问题

4.1~4.8

## EXPORT\_SYMBOL()宏

该宏似乎是将一串字符串输入一个指定的段(\_\_ksymtab\_strings)，同时定义一个结构体struct kernel\_symbol并将其输入一个指定的段(\_\_\_ksymtab)中。

#define \_\_CRC\_SYMBOL(sym, sec) \

extern void \*\_\_crc\_##sym \_\_attribute\_\_((weak)); \

static const unsigned long \_\_kcrctab\_##sym \

\_\_used \

\_\_attribute\_\_((section("\_\_\_kcrctab" sec "+" #sym), unused)) \

= (unsigned long) &\_\_crc\_##sym;

#define \_\_EXPORT\_SYMBOL(sym, sec) \

extern typeof(sym) sym; \

\_\_CRC\_SYMBOL(sym, sec) \

static const char \_\_kstrtab\_##sym[] \

\_\_attribute\_\_((section("\_\_ksymtab\_strings"), aligned(1))) \

= MODULE\_SYMBOL\_PREFIX #sym; \

static const struct kernel\_symbol \_\_ksymtab\_##sym \

\_\_used \

\_\_attribute\_\_((section("\_\_\_ksymtab" sec "+" #sym), unused)) \

= { (unsigned long)&sym, \_\_kstrtab\_##sym }

#define EXPORT\_SYMBOL(sym) \

\_\_EXPORT\_SYMBOL(sym, "")

该宏定义在include/linux/export.h文件中

## 查看.data.read\_mostly段的具体分配

unsigned long totalreserve\_pages \_\_read\_mostly;

#define \_\_read\_mostly \_\_attribute\_\_((\_\_section\_\_(".data.read\_mostly")))

## 查看bitwise控制属性

#define \_\_bitwise\_\_ \_\_attribute\_\_((bitwise))

## prefetch()宏

代码如下

static inline void prefetch(const void \*x)

{

alternative\_input(ASM\_NOP4,

"prefetchnta (%1)",

X86\_FEATURE\_XMM,

"r" (x));

}

## typeof()宏

该宏似乎是一个编译器功能，尚不知道定义，估计是传入一个结构体名称可以将这个结构体类型替换到代码中。

查阅GCC手册后得知，将该宏作用于一个变量，可以得到这个变量的类型，代码示例如下：

int a;

typeof(a) b; // 此时b变量类型也是int

## ENTRY()汇编中到底什么意思

#define ENTRY(name) \

.globl SYMBOL\_NAME(name); \

ALIGN; \

SYMBOL\_NAME\_LABEL(name)

该宏定义于include/linux/linkage.h，还一直以为是个gas特殊扩展，汗...

## 关于中断处理串行话

如果只是记录一个panding标记就结束这次中断处理流程，而留待本中断第一次进入的中断处理现场中的for()来处理，那和中断现场有关的数据岂不是就得不到了。

另有如果一个for中处理时来了两次中断，那第一次会将pandding位置起，那第二次再来就根本无法再给上任何标记，等到执行的那个CPU缓过来执行第一次pandding的那次中断，则第二次岂不是彻底漏掉了。

## 这个有什么意义

#define likely(x) \_\_builtin\_expect(!!(x), 1)

#define unlikely(x) \_\_builtin\_expect(!!(x), 0)

百度之后得到答案，参见2.1.2

## 查看\_\_cacheline\_aligned

static struct softirq\_action softirq\_vec[32] \_\_cacheline\_aligned;

估计这里\_\_cacheline\_aligned与\_\_read\_mostly有类似的作用，都是将该变量定义到指定的段中统一管理吧。具体尚未找到定义处。

## 原子操作

#ifdef CONFIG\_SMP

#define LOCK "lock ; "

#else

#define LOCK ""

#endif

typedef struct { volatile int counter; } atomic\_t;

static \_\_inline\_\_ void atomic\_inc(atomic\_t \*v)

{

\_\_asm\_\_ \_\_volatile\_\_(

LOCK "incl %0"

:"=m" (v->counter)

:"m" (v->counter));

}

注意到对于SMP系统需要加lock前缀，而单核系统则无需该项。查看Intel手册Vol2的inc指令描述，提到如果加lock前缀可以保证操作的完全原子性，目前尚没有思考出一种如果不是原子操作可能带来的问题。

## spin\_lock是否会导致task挂起