# 我计算机体系结构概念与对象

# 应用接口

## 系统调用

### 概念和对象

1. 标准

POSIX 标准，System V标准，BSD标准

1. 重启系统调用

System V标准与BSD标准不同。System V 返回-EINTR给用户自行处理（Linux的默认情况）。BSD标准 SA\_RESTART自动重启。

1. 接口，调用方法与参数传递方法

IA32 int 80 调用门与sysenter。

Eax/ ebx, ecx, edx, esi, edi

1. Sysenter/sysexit的优化

特权级别检查与压栈工作

# 内核基础设施

## VFS

1. Uid, euid, suid, egid, sgid **FIXME**

## 任务调度子系统

### task\_struct

1. parent 与real\_parent的区别？**FIXME**

### do\_execve

do\_execve

🡪open\_exec 打开进程

🡪do\_filp\_open

🡪search\_binary\_handler 查找所有二进制文件格式，挨个试

🡪fmt->load\_binary (load\_elf\_binary) 加载二进制

🡪kernel\_read 读二进制各部分到内存

🡪do\_mmap 对二进制的一些部分映射为VMA

## 时间管理

### 需求

1. 维护“系统全局时间”。
2. 为CFS分时调度算法提供系统定时跳动的脉搏。
3. 为应用程序提供定时的系统服务。

### 概念与对象

#### 时钟源 struct clocksource

1. **概念**

时钟源是一种计时器或计数器硬件，仅提供read数值的功能，没有超时事件。可以通过读取精确的硬件计数来维护系统的“全局时间”，完成需求1.

1. **对象**

struct clocksource \*curr\_clocksource; 当前选择的最佳时钟源

#### 时钟事件设备 struct clock\_event\_device

1. **概念**

一种可提供超时事件的设备，支撑完成需求2和需求3。

**event\_handler**  该字段是一个回调函数指针，通常由通用框架层设置，在时间中断到来时，machine底层的中断服务程序会调用该回调，框架层利用该回调实现对时钟事件的处理。

提供了一个clock\_event\_device 状态变化的通知链，如ADD、SUSPEND等。

1. **对象**

#### Tick设备 struct tick\_device

1. **概念**

struct tick\_device {

struct clock\_event\_device \*evtdev;

enum tick\_device\_mode mode;

};

其实是clock\_event\_device的简单封装和应用者，为调度提供tick，mode是单触发或周期性触发。

利用 clock\_event\_device 的通知链来进行设备的evtdev的变更。

1. **对象**

DEFINE\_PER\_CPU(struct tick\_device, tick\_cpu\_device);  每个CPU有一个。

#### 系统时间 struct timekeeper

1. **RTC时间**：CMOS时间。
2. **Xtime[CLOCK\_REALTIME]** ：墙上时间wall。
3. **monotonic time[CLOCK\_MONOTONIC]**：从启动以来的单调递增。
4. **raw monotonic time[CLOCK\_MONOTONIC\_RAW]**：同**monotonic time，但不受NTP影响。**
5. **boot time[CLOCK\_BOOTTIME]**：同**monotonic time，但是包括休眠时间。**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **时间种类** | **精度（统计单位）** | **访问速度** | **累计休眠时间** | **受NTP调整的影响** |
| RTC | 低 | 慢 | Yes | Yes |
| xtime | 高 | 快 | Yes | Yes |
| monotonic | 高 | 快 | No | Yes |
| raw monotonic | 高 | 快 | No | No |
| boot time | 高 | 快 | Yes | Yes |

static struct timekeeper timekeeper; 该全局对象有所有的系统时间。

struct timekeeper {

/\* Current clocksource used for timekeeping. \*/

struct clocksource \*clock; 使用当前时钟源来更新系统时间。

。。。。。。

}

#### 高精度/低精度时钟

#### 动态时钟 NO\_HZ

### 架构

### 流程

#### 初始化tick\_device

clockevents\_register\_device🡪 clockevents\_do\_notify

🡪tick\_notify

🡪 tick\_check\_new\_device

🡪 tick\_setup\_device

if (td->mode == TICKDEV\_MODE\_PERIODIC)

tick\_setup\_periodic(newdev, 0);

else

tick\_setup\_oneshot(newdev, handler, next\_event);

## 定时器

### Hrtimer (hrtimer.c)

1. SMP下每个CPU有2颗不同类型时钟源（hrtimer\_clock\_base）的红黑树 hrtimer\_cpu\_base，分别是 CLOCK\_REALTIME 和CLOCK\_MONOTONIC。创建一个定时器，就在树上有一个结点（struct hrtimer）。可以看到hrtimer并非依赖于时钟中断，而是任意一个中断来临都会做超时计算。
2. hrtimer\_nanosleep 是关键的创建定时器函数，do\_nanosleep执行睡眠（TASK\_INTERRUPTIBLE）操作，可能被信号唤醒，会重新计时。
3. 初始化时open\_softirq(HRTIMER\_SOFTIRQ, run\_hrtimer\_softirq);会注册软中断。每一个软中断上下文都必须判断定时器的超时间。
4. FIXME: 没有那么高精度的中断，如何得到触发？

FIXME：没有发现产品说的高精度定时器

### Timer (timer.c)

### HPET

#### 初始化

setup\_arch

# 内核高级应用

## Ptrace

### 主要数据结构

#### 调试状态

1. 如果traced task正在运行，traced 的task\_strcuct 🡪 ptrace归traced task拥有
2. 如果traced task已经停止，traced 的task\_strcuct 🡪 ptrace归tracer拥有。

一个task只要被attach了就设置，不管其是运行的还是STOP的，还是处于被真正调试的状态

#define PT\_PTRACED 0x00000001

#define PT\_DTRACE 0x00000002 /\* delayed trace (used on m68k, i386) \*/

#define PT\_TRACESYSGOOD 0x00000004

#define PT\_PTRACE\_CAP 0x00000008 /\* ptracer can follow suid-exec \*/

#define PT\_TRACE\_FORK 0x00000010

#define PT\_TRACE\_VFORK 0x00000020

#define PT\_TRACE\_CLONE 0x00000040

#define PT\_TRACE\_EXEC 0x00000080

#define PT\_TRACE\_VFORK\_DONE 0x00000100

#define PT\_TRACE\_EXIT 0x00000200

#define PT\_TRACE\_MASK 0x000003f4

#### 进程状态

task被停止的时候设置，一个进程只有被STOP才能采取进一步的调试

#define \_\_TASK\_STOPPED 4

#define TASK\_STOPPED (TASK\_WAKEKILL | \_\_TASK\_STOPPED)

task被STOP之后被真正调试的时候设置

#define \_\_TASK\_TRACED 8

#define TASK\_TRACED (TASK\_WAKEKILL | \_\_TASK\_TRACED)

进程状态与跟踪状态的区别：

1. 跟踪状态表示一个进程在attach之后具体的调试状态，PT\_PTRACED 只要被attach之后或者PTRACE\_ME之后就会被永久设置。
2. task的进程状态仅仅被调度器感知。只要处于STOP状态或者真正调试时才会设置为TASK\_STOPPED或TASK\_TRACED。正常运行时，即使它被attach，也不会设置这两个值。

#### 跟踪链表

task\_struct 🡪 struct list\_head ptraced; // tracer保存它跟踪的所有task的链表头

task\_struct 🡪struct list\_head ptrace\_entry; // traced task在跟踪链表中的结点

#### 父子关系

Tracer task和traced task是一个临时的父子关系，traced task需要能够访问tracer，

task\_struct 🡪struct task\_struct \*parent; // traced task的该成员可访问tracer

### Trace操作

#define PTRACE\_TRACEME 0

#define PTRACE\_PEEKTEXT 1

#define PTRACE\_PEEKDATA 2

#define PTRACE\_PEEKUSR 3

#define PTRACE\_POKETEXT 4

#define PTRACE\_POKEDATA 5

#define PTRACE\_POKEUSR 6

#define PTRACE\_CONT 7

#define PTRACE\_KILL 8

#define PTRACE\_SINGLESTEP 9

#define PTRACE\_ATTACH 16

#define PTRACE\_DETACH 17

#define PTRACE\_SYSCALL 24

#### PTRACE\_TRACEME

进程主动让父进程进行跟踪。FIXME：父进程何时发送STOP信号？

#### PTRACE\_ATTACH

三步曲:

1. Traced task 🡪 task\_strcuct 🡪 ptrace 设置为 PF\_TRACED
2. Traced task 🡪 task\_strcuct 🡪parent 设置为 tracer
3. 发送SIG\_STOP信号到 traced task。
4. 一般发送完之后，tracer会调用wait\_pid等待Traced task在处理STOP信号时发送的SIGCHID信号，表示该Traced task完全停止下来了。

注意：要对跟踪进行处理，必须先让其STOP下来。

#### PTRACE\_SYSCALL和PTRACE\_CONT

系统调用调试：

int ptrace\_request(struct task\_struct \*child, long request,

long addr, long data)

{

case PTRACE\_SYSCALL: // 要求进行系统调用

case PTRACE\_CONT: // 如果是continue的话，则纯粹地唤醒traced task继续执行

return ptrace\_resume(child, request, data);

……

}

static int ptrace\_resume(struct task\_struct \*child, long request, long data)

{

if (request == PTRACE\_SYSCALL)

set\_tsk\_thread\_flag(child, TIF\_SYSCALL\_TRACE);

else

clear\_tsk\_thread\_flag(child, TIF\_SYSCALL\_TRACE);

…….

child->exit\_code = data; // 这里把exit\_code设置为 SIGKILL

wake\_up\_process(child);

return 0;

}

static inline void ptrace\_report\_syscall(struct pt\_regs \*regs)

{

int ptrace = task\_ptrace(current);

if (!(ptrace & PT\_PTRACED))

return;

// 给父进程发送SIGCHLD信号，然后自己睡眠，之后父进程使用SIGCONT唤醒traced task

ptrace\_notify(SIGTRAP | ((ptrace & PT\_TRACESYSGOOD) ? 0x80 : 0));

/\*

\* this isn't the same as continuing with a signal, but it will do

\* for normal use. strace only continues with a signal if the

\* stopping signal is not SIGTRAP. -brl

\*/

if (current->exit\_code) {

send\_sig(current->exit\_code, current, 1);

current->exit\_code = 0;

}

#### PTRACE\_KILL

给traced进程发送一个SIGKILL信号，不管该进程是否已经STOP。

int ptrace\_request(struct task\_struct \*child, long request,

long addr, long data)

{

……….

case PTRACE\_KILL:

if (child->exit\_state) /\* already dead \*/

return 0;

return ptrace\_resume(child, request, SIGKILL);

……

}

static int ptrace\_resume(struct task\_struct \*child, long request, long data)

{

…….

child->exit\_code = data; // 这里把exit\_code设置为 SIGKILL

wake\_up\_process(child);

return 0;

}

Traced task当进行一次系统调用时，FIXME，如果当时进程不处于系统调用调试阶段，不会进入ptrace\_report\_syscall，怎么办？

static inline void ptrace\_report\_syscall(struct pt\_regs \*regs)

{

………….

if (current->exit\_code) { // 这里会给自己发送一个SIGKILL信号

send\_sig(current->exit\_code, current, 1);

current->exit\_code = 0;

}

}

### gdb

# 硬件资源管理

## CPU

### 硬件视角资源管理

### 硬件驱动

# 内存

## 内存模型

1. **内存有两个好的设计**
2. 页表的通用设计可以应付各种体系架构
3. UMA与NUMA的统计设计
4. **重要概念**
5. UMA, NUMA
6. FLATMEM, DISCONTIGMEM, SPARSEMEM NUMA的话一定不会是FLATMEM
7. 结点 struct pglist\_data( pg\_data\_t); enum node\_states

UMA只有一个结点

1. 内存域 struct zone;
2. DMA/DMA32/NORMAL/HIGHMEM/MOVABLE ?
3. 64位系统是否还有HIGH域？DMA到底有什么用？ **FIXME**？
4. 页帧struct page **FIXME**：一定是4K一个对象么？

UMA：全局变量mem\_map 包含所有页帧

NUMA：???

1. 页表
2. 提供4级页表。
3. 页表相关宏与页表操作定义在不同的系统结构头文件中。（很复杂）
4. PTE的bitmap状态位（其实每一级对象都有这个bitmap状态位）

## 页表

#### 32位

1. pgtable-3level\_types.h和pgtable-2level\_types.h 是比较重要的模拟PMU，PUD的实现。

#### 64位

1. 真正的4级页表（共48位）：

PGD: 9

PUD: 9

PMD: 9

PTE: 9

Offset: 12

## 物理内存管理

### Page对象

#### \_refcount

\_refcount是分配使用free的计数，从buddy分配后为1。

get\_page ++

put\_page -- 当减为0，表示不需要了，释放回buddy。

get\_page

1. \_\_lru\_cache\_add：加入LRU链表。
2. copy\_one\_pte：映射到其它用户进程，写时拷贝
3. 其它地方，参考《奔跑》P190

#### \_mapcount

表示映射了多少个进程的页表。

==-1: 没有映射到pte

==0 : 只有父进程映射。

>0: 至少2个进程mount

#### PG\_locked锁定

其实相当于实现了一个互斥bit，同一时刻只能有一个thread对page进行lock。试图lock一个locked的page，会导致该thread睡眠。

FIXME，和\_\_SetPageLocked、lock\_page区别？

#### 【空间性能】

Page结构体有大量的union，以及复用的成员，可以极大地减少page对象的大小。

### Page向量对象【时间性能】

struct pagevec {

unsigned long nr;

unsigned long cold;

struct page \*pages[PAGEVEC\_SIZE]; //14，这个写死了

};

相当于多个page的缓存，当集中到14个之后，再集体批处理操作，会比一个page，一个page的操作性能要高。

include/linux/pagevec.h

### 初始化

**setup\_arch：**

1. setup\_arch🡪setup\_memory\_map (e820.c)

根据boot\_params的信息，未经修改的转储到e820变量中（**struct e820map e820**）

**boot\_params**：BIOS启动参数，在head.S（或head\_32.S）从某个内存拷贝。

boot\_params.e820\_entries bios传递的内存表项

boot\_params.e820\_map bios传递的内存表

FIXME, e820.c文件中其它疑问：

1. bios传的reserve是啥意思，与内核中的reserve有何区别？
2. e820与**e820\_save**的关系
3. 里面有很多update相关函数
4. setup\_arch🡪 parse\_setup\_data ：

FIXME：处理boot\_params.hdr.setup\_data和SETUP\_E820\_EXT (e820.c)，需要early\_ioremap

1. setup\_arch🡪e820\_reserve\_setup\_data：

FIXME：处理boot\_params.hdr.setup\_data和E820\_RESERVED\_KERN，操作e820\_save (e820.c)。

1. setup\_arch🡪 reserve\_early\_setup\_data：

FIXME：处理boot\_params.hdr.setup\_data和reserve\_early (e820.c)

1. setup\_arch🡪finish\_e820\_parsing

FIXME：最后校验并打印最终的图，根据命令行参数memmap

1. setup\_arch🡪 e820\_update\_range

setup\_arch {……

#ifdef CONFIG\_X86\_32

if (ppro\_with\_ram\_bug()) {

e820\_update\_range(0x70000000ULL, 0x40000ULL, E820\_RAM,

E820\_RESERVED);

sanitize\_e820\_map(e820.map, ARRAY\_SIZE(e820.map), &e820.nr\_map);

printk(KERN\_INFO "fixed physical RAM map:\n");

e820\_print\_map("bad\_ppro"); // FIXME:排除一些坏的区域

}

#else

early\_gart\_iommu\_check();

#endif

1. setup\_arch🡪 e820\_end\_of\_ram\_pfn

max\_pfn = e820\_end\_of\_ram\_pfn();

根据e820变量，根据可以usable可用的（类型为E820\_RAM）内存计算**max\_pfn**

1. setup\_arch🡪 early\_reserve\_e820\_mpc\_new FIXME：处理MP
2. setup\_arch🡪 处理mtrr FIXME：什么是mtrr

mtrr\_bp\_init();

if (mtrr\_trim\_uncached\_memory(max\_pfn))

max\_pfn = e820\_end\_of\_ram\_pfn(); // 重新计算**max\_pfn**，肯定是某些内存不能用了

1. **num\_physpages** = max\_pfn;
2. 计算关键变量**max\_low\_pfn**，**high\_memory**，**highmem\_pages**，**max\_pfn\_mapped**

#ifdef CONFIG\_X86\_32

/\* max\_low\_pfn get updated here \*/

find\_low\_pfn\_range();

#else

/\* How many end-of-memory variables you have, grandma! \*/

/\* need this before calling reserve\_initrd \*/

if (max\_pfn > (1UL<<(32 - PAGE\_SHIFT)))

max\_low\_pfn = e820\_end\_of\_low\_ram\_pfn();

else

max\_low\_pfn = max\_pfn; //说明没有highmem

high\_memory = (void \*)\_\_va(max\_pfn \* PAGE\_SIZE - 1) + 1;

max\_pfn\_mapped = KERNEL\_IMAGE\_SIZE >> PAGE\_SHIFT;

1. setup\_arch🡪 init\_gbpages(); 初始化direct\_gbpages，GB大小的页
2. 完成最终页表映射

/\* max\_pfn\_mapped is updated here \*/

**max\_low\_pfn\_mapped** = init\_memory\_mapping(0, max\_low\_pfn<<PAGE\_SHIFT);

max\_pfn\_mapped = max\_low\_pfn\_mapped;

#ifdef CONFIG\_X86\_64

if (max\_pfn > max\_low\_pfn) {

max\_pfn\_mapped = init\_memory\_mapping(1UL<<32,

max\_pfn<<PAGE\_SHIFT);

/\* can we preseve max\_low\_pfn ?\*/

max\_low\_pfn = max\_pfn;

}

#endif

1. setup\_arch🡪 initmem\_init 初始化bootmem

initmem\_init(0, max\_pfn);

1. FIXME，初始化bootmem？

x86\_init.paging.pagetable\_setup\_start(swapper\_pg\_dir);

paging\_init();

x86\_init.paging.pagetable\_setup\_done(swapper\_pg\_dir);

1. FIXME，初始化bootmem？

### Bootmem

### 伙伴系统

#### 正常申请

核心函数get\_page\_from\_freelist，在整个分配过程中会调用很多次，很多种情况。

\_\_alloc\_pages\_nodemask

🡪 get\_page\_from\_freelist 处理链表（快速申请）

🡪 \_\_alloc\_pages\_slowpath 快速申请失败了，走慢速通道

🡪 get\_page\_from\_freelist （很多条件，这里会调用若干次该函数）

🡪 \_\_alloc\_pages\_may\_oom 这里会oom

🡪 get\_page\_from\_freelist 处理链表

### SLAB/SLOB/SLUB

### 内存域”水印”

**FIXME**：需要详细理解

### 冷热页

**FIXME**：需要详细理解

### 复合页

**FIXME**：需要详细理解

### Lockpage锁定

其实相当于实现了一个互斥bit，同一时刻只能有一个thread对page进行lock。试图lock一个locked的page，会导致该thread睡眠。

FIXME，和\_\_SetPageLocked、lock\_page区别？

## 内核虚拟内存管理

#### IA32内核地址空间

1. 内核态/用户态地址空间的划分
2. 用户态和内核态各映射4G地址空间，内核态直接映射，每个用户态进程有一个非直接映射。坏处是ring0与ring3切换时需要切换页表，TLB要清零。
3. 基于上述原因，内核态与用户态要使用同一页表，只能共享4G地址空间。
4. 3:1 是经典模型，如果内核态应用比较多，可以调整。
5. 内核地址空间
6. C0000000 ~ C0000000 + high\_memory (直接映射边界)
7. VMALLOC\_OFFSET空洞
8. VMALLOC AREA (VMALLOC\_START ~ VMALLOC\_END)

不连续的物理内存，连续的地址空间

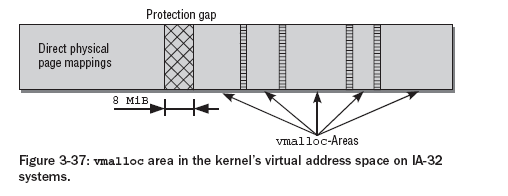
1. 空洞
2. 永久映射 ( PKMAP\_BASE ~ FIXADDR\_START)
3. 固定映射 (FIXADDR\_START ~ 4G )
4. MAXMEM，\_\_VMALLOC\_RESERVE ？ **FIXME**

#### IA64内核地址空间

1. 0000000000000000-0000000030000000 （max\_low\_pfn<<PAGE\_SHIFT）从这里可以看到64位地址空间没有highmem，max\_low\_pfn就是最大的物理页

#### VMALLOC

1. **结构**



struct vm\_struct {

struct vm\_struct \*next; // 链表，struct vm\_struct \*vmlist; 是表头

void \*addr; 线性地址

unsigned long size;

unsigned long flags;

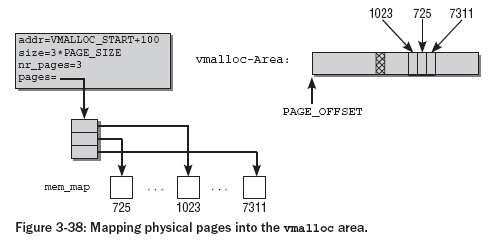
struct page \*\*pages; 页的数组

unsigned int nr\_pages;

unsigned long phys\_addr; ioremap的物理地址，因为ioremap的可能没有页

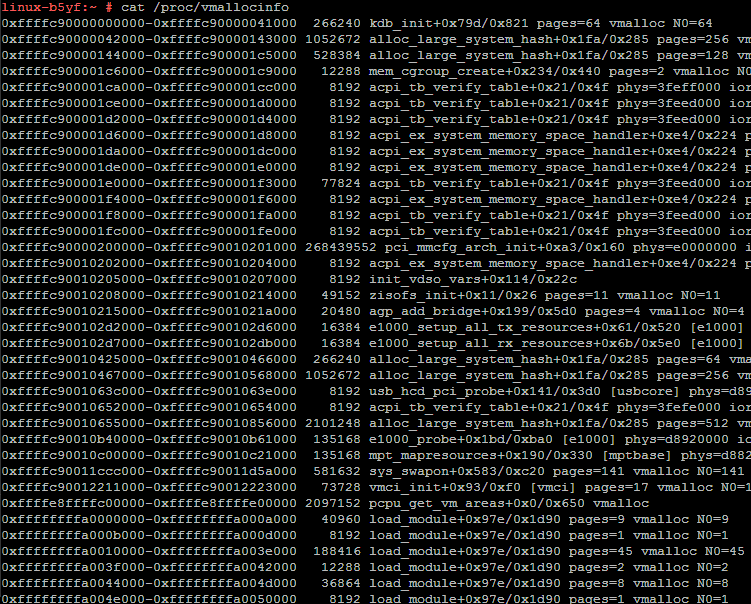
void \*caller;

};



另外还有一颗红黑树，树的结点(vmap\_area)也与一个vm\_struct一一对应，主要是用于快速查找用。因为64位地址空间后，再线性查找可能性能会有下降。

下图是所有vmalloc区的列表：



1. **地址范围**
2. **接口**
3. **其它要点**

主要是内核模块使用，再者有不少驱动也需要申请内存。FIXME：为什么非要一片连续的线性地址空间呢？

#### 永久映射PKMAP

**FIXME**：如果没有HIGH\_MEM，则没有此区域

实例

#### 固定映射FIXMAP

**FIXME**：编译器如何优化？为何要比一般的映射要快？为什么页表项不会从TLB刷出？

实例？

## 进程虚拟内存管理

### 进程虚拟地址空间与布局

### 内存映射

### 反向映射

### 缺页异常

## Cache/TLB

#### HugeTLB

大页就是PSE。把原来的4级页表缩成3级页表。

64位x86的大页可能为1G。

## 页面回收

### LRU链表集(per node)

每个zone都有5个LRU链表:

enum lru\_list {

LRU\_INACTIVE\_ANON = LRU\_BASE,

LRU\_ACTIVE\_ANON = LRU\_BASE + LRU\_ACTIVE,

LRU\_INACTIVE\_FILE = LRU\_BASE + LRU\_FILE,

LRU\_ACTIVE\_FILE = LRU\_BASE + LRU\_FILE + LRU\_ACTIVE,

LRU\_UNEVICTABLE,

NR\_LRU\_LISTS

};

struct pglist\_data { struct lruvec lruvec; }

struct lruvec {

struct list\_head lists[NR\_LRU\_LISTS];

}

## 其它性能优化

1. zone对象访问非常频繁，因此必须要利用好cache，ZONE\_PADDING，是为了把不同进程访问的信息隔在不同的cache line中。

\_\_cacheline\_maxaligned\_in\_smp 关键字

1. page对象：384MB的内存就有100000页，因此page必须尽可能小。利用union来解决此问题。

# 文件系统

## VFS

### 文件系统级接口

#### 文件系统类型file\_system\_type

int register\_filesystem(struct file\_system\_type \* fs) 注册文件系统

struct file\_system\_type {

const char \*name; 名称如ext3

int fs\_flags;

#define FS\_REQUIRES\_DEV 1

#define FS\_BINARY\_MOUNTDATA 2

#define FS\_HAS\_SUBTYPE 4

#define FS\_USERNS\_MOUNT 8 /\* Can be mounted by userns root \*/

#define FS\_USERNS\_DEV\_MOUNT 16 /\* A userns mount does not imply MNT\_NODEV \*/

#define FS\_RENAME\_DOES\_D\_MOVE 32768 /\* FS will handle d\_move() during rename() internally. \*/

struct dentry \*(\*mount) (struct file\_system\_type \*, int, mount的时候调用

const char \*, void \*);

void (\*kill\_sb) (struct super\_block \*); umount的时候调用

struct module \*owner;

struct file\_system\_type \* next;

struct hlist\_head fs\_supers;

struct lock\_class\_key s\_lock\_key;

struct lock\_class\_key s\_umount\_key;

struct lock\_class\_key s\_vfs\_rename\_key;

struct lock\_class\_key s\_writers\_key[SB\_FREEZE\_LEVELS];

struct lock\_class\_key i\_lock\_key;

struct lock\_class\_key i\_mutex\_key;

struct lock\_class\_key i\_mutex\_dir\_key;

};

#### 超级块的钩子super\_operations

其实可以与file\_system\_type合并，它主要包括：

1. Inode操作
2. 文件系统分区的一些统计
3. 文件系统对超级块的一些附加处理

struct super\_operations {

struct inode \*(\*alloc\_inode)(struct super\_block \*sb); 创建FS-SPECIFIC inode 对象

void (\*destroy\_inode)(struct inode \*); 销毁FS-SPECIFIC inode 对象

void (\*dirty\_inode) (struct inode \*, int flags);

int (\*write\_inode) (struct inode \*, struct writeback\_control \*wbc);

int (\*drop\_inode) (struct inode \*);

void (\*evict\_inode) (struct inode \*);

void (\*put\_super) (struct super\_block \*);

同步整个FS分区：

1) sync系统调用

2) syncfs 系统调用

3) sync\_filesystem kAPI，比如umount

int (\*sync\_fs)(struct super\_block \*sb, int wait);

int (\*freeze\_fs) (struct super\_block \*);

int (\*unfreeze\_fs) (struct super\_block \*);

FS-SPECIFIC的统计函数：

1) ustat系统调用

2) fstatfs系统调用

3) fstatfs64系统调用

4) statfs系统调用

5) statfs64系统调用

6) vfs\_statfs kAPI

int (\*statfs) (struct dentry \*, struct kstatfs \*);

int (\*remount\_fs) (struct super\_block \*, int \*, char \*);

void (\*umount\_begin) (struct super\_block \*);

显示分区参数/proc/mounts和/proc/mountinfo

int (\*show\_options)(struct seq\_file \*, struct dentry \*);

int (\*show\_devname)(struct seq\_file \*, struct dentry \*);

int (\*show\_path)(struct seq\_file \*, struct dentry \*);

显示分区统计/proc/mountstats

int (\*show\_stats)(struct seq\_file \*, struct dentry \*);

#ifdef CONFIG\_QUOTA

ssize\_t (\*quota\_read)(struct super\_block \*, int, char \*, size\_t, loff\_t);

ssize\_t (\*quota\_write)(struct super\_block \*, int, const char \*, size\_t, loff\_t);

#endif

int (\*bdev\_try\_to\_free\_page)(struct super\_block\*, struct page\*, gfp\_t);

int (\*nr\_cached\_objects)(struct super\_block \*);

void (\*free\_cached\_objects)(struct super\_block \*, int);

};

### 文件或目录元数据接口

#### inode\_operations (FILE/DIR/SYMLINK各一个对象)

struct inode\_operations {

到父目录中查找子项，dentry入参只有名称，这里补充dentry作为出参，并创建相应inode对象。

Open的核心函数是link\_path\_walk的遍历 与 lookup\_real：dir->i\_op->lookup

struct dentry \* (\*lookup) (struct inode \*,struct dentry \*, unsigned int);

void \* (\*follow\_link) (struct dentry \*, struct nameidata \*);

int (\*permission) (struct inode \*, int);

struct posix\_acl \* (\*get\_acl)(struct inode \*, int);

int (\*readlink) (struct dentry \*, char \_\_user \*,int);

void (\*put\_link) (struct dentry \*, struct nameidata \*, void \*);

int (\*create) (struct inode \*,struct dentry \*, umode\_t, bool);

int (\*link) (struct dentry \*,struct inode \*,struct dentry \*);

int (\*unlink) (struct inode \*,struct dentry \*);

int (\*symlink) (struct inode \*,struct dentry \*,const char \*);

int (\*mkdir) (struct inode \*,struct dentry \*,umode\_t);

int (\*rmdir) (struct inode \*,struct dentry \*);

int (\*mknod) (struct inode \*,struct dentry \*,umode\_t,dev\_t);

int (\*rename) (struct inode \*, struct dentry \*,

struct inode \*, struct dentry \*);

int (\*setattr) (struct dentry \*, struct iattr \*);

int (\*getattr) (struct vfsmount \*mnt, struct dentry \*, struct kstat \*);

int (\*setxattr) (struct dentry \*, const char \*,const void \*,size\_t,int);

ssize\_t (\*getxattr) (struct dentry \*, const char \*, void \*, size\_t);

ssize\_t (\*listxattr) (struct dentry \*, char \*, size\_t);

int (\*removexattr) (struct dentry \*, const char \*);

int (\*fiemap)(struct inode \*, struct fiemap\_extent\_info \*, u64 start,

u64 len);

int (\*update\_time)(struct inode \*, struct timespec \*, int);

int (\*atomic\_open)(struct inode \*, struct dentry \*,

struct file \*, unsigned open\_flag,

umode\_t create\_mode, int \*opened);

} \_\_\_\_cacheline\_aligned;

通常会有3个对象，一个文件的，一个目录的，一个链接

const struct inode\_operations jffs2\_file\_inode\_operations;

const struct inode\_operations jffs2\_dir\_inode\_operations;

const struct inode\_operations jffs2\_symlink\_inode\_operations

#### iattr，xattr\_handler 属性与扩展属性

文件属性的结构体iattr的属性个数是固定的，为了扩展，就创造了扩展属性xattr的概念。

扩展属性的对象非常简单，一个名称、一个值。并有一组操作集合。

struct xattr {

char \*name;

void \*value;

size\_t value\_len;

};

struct xattr\_handler {

const char \*prefix;

int flags; /\* fs private flags passed back to the handlers \*/

size\_t (\*list)(struct dentry \*dentry, char \*list, size\_t list\_size,

const char \*name, size\_t name\_len, int handler\_flags);

int (\*get)(struct dentry \*dentry, const char \*name, void \*buffer,

size\_t size, int handler\_flags);

int (\*set)(struct dentry \*dentry, const char \*name, const void \*buffer,

size\_t size, int flags, int handler\_flags);

};

一般一个文件系统会有多个扩展属性，并有一个数组保存，一个属性有一组操作集合（xattr\_handler）。

const struct xattr\_handler \*jffs2\_xattr\_handlers[]; 数组

const struct xattr\_handler jffs2\_acl\_access\_xattr\_handler;

const struct xattr\_handler jffs2\_acl\_default\_xattr\_handler;

const struct xattr\_handler jffs2\_security\_xattr\_handler;

const struct xattr\_handler jffs2\_trusted\_xattr\_handler;

const struct xattr\_handler jffs2\_user\_xattr\_handler;

xattr的设置获取总控制，相关的接口在inode\_operations对象中。

const struct inode\_operations jffs2\_dir\_inode\_operations =

.getattr = NULL,

.setattr = jffs2\_setattr,

.setxattr = jffs2\_setxattr, 设置一个扩展属性信息

.getxattr = jffs2\_getxattr, 得到一个扩展属性信息

.listxattr = jffs2\_listxattr, 列出所有的扩展属性

.removexattr = jffs2\_removexattr 删除一个扩展属性

}

### 数据读写接口

#### file\_operations (FILE/DIR各一个对象)

struct file\_operations{

struct module \*owner;

loff\_t (\*llseek) (struct file \*, loff\_t, int);

ssize\_t (\*read) (struct file \*, char \_\_user \*, size\_t, loff\_t \*);

ssize\_t (\*write) (struct file \*, const char \_\_user \*, size\_t, loff\_t \*);

ssize\_t (\*aio\_read) (struct kiocb \*, const struct iovec \*, unsigned long, loff\_t);

ssize\_t (\*aio\_write) (struct kiocb \*, const struct iovec \*, unsigned long, loff\_t);

int (\*readdir) (struct file \*, void \*, filldir\_t);

unsigned int (\*poll) (struct file \*, struct poll\_table\_struct \*);

long (\*unlocked\_ioctl) (struct file \*, unsigned int, unsigned long);

long (\*compat\_ioctl) (struct file \*, unsigned int, unsigned long);

int (\*mmap) (struct file \*, struct vm\_area\_struct \*);

int (\*open) (struct inode \*, struct file \*);

int (\*flush) (struct file \*, fl\_owner\_t id);

int (\*release) (struct inode \*, struct file \*);

int (\*fsync) (struct file \*, loff\_t, loff\_t, int datasync);

int (\*aio\_fsync) (struct kiocb \*, int datasync);

int (\*fasync) (int, struct file \*, int);

int (\*lock) (struct file \*, int, struct file\_lock \*);

ssize\_t (\*sendpage) (struct file \*, struct page \*, int, size\_t, loff\_t \*, int);

unsigned long (\*get\_unmapped\_area)(struct file \*, unsigned long, unsigned long, unsigned long, unsigned long);

int (\*check\_flags)(int);

int (\*flock) (struct file \*, int, struct file\_lock \*);

ssize\_t (\*splice\_write)(struct pipe\_inode\_info \*, struct file \*, loff\_t \*, size\_t, unsigned int);

ssize\_t (\*splice\_read)(struct file \*, loff\_t \*, struct pipe\_inode\_info \*, size\_t, unsigned int);

int (\*setlease)(struct file \*, long, struct file\_lock \*\*);

long (\*fallocate)(struct file \*file, int mode, loff\_t offset,

loff\_t len);

int (\*show\_fdinfo)(struct seq\_file \*m, struct file \*f);

};

通常会有两个对象，一个文件的，一个目录的。

const struct file\_operations jffs2\_file\_operations;

const struct file\_operations jffs2\_dir\_operations;

#### address\_space\_operations

在以下两种情况下使用：

* + 1. 正常读写，该结构体位于file\_operations对象的接口的下层，一般file\_operations对象的读写函数都是通用VFS函数，本接口的读写函数是FS-SPECIFIC的。
    2. mmap缺页读写

struct address\_space\_operations {

int (\*writepage)(struct page \*page, struct writeback\_control \*wbc);

int (\*readpage)(struct file \*, struct page \*);

/\* Write back some dirty pages from this mapping. \*/

int (\*writepages)(struct address\_space \*, struct writeback\_control \*);

/\* Set a page dirty. Return true if this dirtied it \*/

int (\*set\_page\_dirty)(struct page \*page);

int (\*readpages)(struct file \*filp, struct address\_space \*mapping,

struct list\_head \*pages, unsigned nr\_pages);

int (\*write\_begin)(struct file \*, struct address\_space \*mapping,

loff\_t pos, unsigned len, unsigned flags,

struct page \*\*pagep, void \*\*fsdata);

int (\*write\_end)(struct file \*, struct address\_space \*mapping,

loff\_t pos, unsigned len, unsigned copied,

struct page \*page, void \*fsdata);

/\* Unfortunately this kludge is needed for FIBMAP. Don't use it \*/

sector\_t (\*bmap)(struct address\_space \*, sector\_t);

void (\*invalidatepage) (struct page \*, unsigned long);

int (\*releasepage) (struct page \*, gfp\_t);

void (\*freepage)(struct page \*);

ssize\_t (\*direct\_IO)(int, struct kiocb \*, const struct iovec \*iov,

loff\_t offset, unsigned long nr\_segs);

int (\*get\_xip\_mem)(struct address\_space \*, pgoff\_t, int,

void \*\*, unsigned long \*);

/\*

\* migrate the contents of a page to the specified target. If sync

\* is false, it must not block.

\*/

int (\*migratepage) (struct address\_space \*,

struct page \*, struct page \*, enum migrate\_mode);

int (\*launder\_page) (struct page \*);

int (\*is\_partially\_uptodate) (struct page \*, read\_descriptor\_t \*,

unsigned long);

int (\*error\_remove\_page)(struct address\_space \*, struct page \*);

/\* swapfile support \*/

int (\*swap\_activate)(struct swap\_info\_struct \*sis, struct file \*file,

sector\_t \*span);

void (\*swap\_deactivate)(struct file \*file);

};

const struct address\_space\_operations jffs2\_file\_address\_operations

## 页缓存

相关代码：(filemap.c)

### 关键数据结构

最关键是address\_space 和radix\_tree\_root 两个数据结构，都是以inode为单位。

struct radix\_tree\_root {

gfp\_t gfp\_mask;

struct radix\_tree\_node \_\_rcu \*rnode; // page节点

};

inode的page缓存管理，行为ops由不同的ops由lowerfs有不同的处理, ops由lowerfs决定

struct address\_space {

struct inode \*host; /\* owner: inode, block\_device \*/ 所属的inode

struct radix\_tree\_root page\_tree; /\* radix tree of all pages \*/ 树根

const struct address\_space\_operations \*a\_ops; /\* methods \*/ /\*不同的ops由lowerfs有不同的处理, ops由lowerfs决定\*/

### 关键对象

static const struct address\_space\_operations def\_blk\_aops = {

.readpage = blkdev\_readpage,

.readpages = blkdev\_readpages,

.writepage = blkdev\_writepage,

.write\_begin = blkdev\_write\_begin,

.write\_end = blkdev\_write\_end,

.writepages = blkdev\_writepages,

.releasepage = blkdev\_releasepage,

.direct\_IO = blkdev\_direct\_IO,

.is\_dirty\_writeback = buffer\_check\_dirty\_writeback,

};

### 关键操作

#### 插入

add\_to\_page\_cache

add\_to\_page\_cache\_locked

\_\_add\_to\_page\_cache\_locked

add\_to\_page\_cache\_lru

\_\_add\_to\_page\_cache\_locked

\_\_SetPageLocked(page); 锁住

page->mapping = mapping; 所属的address\_space

page->index = offset; 当前的page位于文件的偏移

get\_page(page); 引用计数

#### 查找

find\_get\_page

#### 预读

只发生在buffer读流程，预读对于page cache来说不是必要的，默认它会开启？FIXME。如何配置。(readahead.c)

### 参考

《PLKA 16.4》页缓存

## 块缓存

相关代码：fs/buffer.c、fs/mpage.c

### 关键数据结构

struct buffer\_head

### 关键对象

static const struct address\_space\_operations def\_blk\_aops = {

.readpage = blkdev\_readpage,

.readpages = blkdev\_readpages,

.writepage = blkdev\_writepage,

.write\_begin = blkdev\_write\_begin,

.write\_end = blkdev\_write\_end,

.writepages = blkdev\_writepages,

.releasepage = blkdev\_releasepage,

.direct\_IO = blkdev\_direct\_IO,

.is\_dirty\_writeback = buffer\_check\_dirty\_writeback,

};

### 关键操作

#### 操作buffer header

create\_empty\_buffers 根据一个page创建一个buffer\_head

block\_read\_full\_page 读一个完整页，流程很像do\_mpage\_readpage，如果部分页是新的，block\_read\_full\_page效率比较高。否则用do\_mpage\_readpage。

block\_write\_full\_page 写一个完整页

上述接口，直接下发读bh需求，并读取数据填充到参数中的page，和pagecache无关。

#### 关联块缓存和页缓存

do\_mpage\_readpage

/\*这个函数试图读取文件中的一个page大小的数据，最理想的情况下就是这个page大小

的数据都是在连续的物理磁盘上面的，然后函数只需要提交一个bio请求就可以获取

所有的数据，这个函数大部分工作在检查page上所有的物理块是否连续，检查的方法

就是调用文件系统提供的get\_block函数，如果不连续，需要调用block\_read\_full\_page

函数采用buffer 缓冲区的形式来逐个块获取数据\*/

/\*

1、调用get\_block函数检查page中是不是所有的物理块都连续

2、如果连续调用mpage\_bio\_submit函数请求整个page的数据

3、如果不连续调用block\_read\_full\_page逐个block读取

\*/

来源于 <http://blog.chinaunix.net/uid-28236237-id-4028521.html> 《do\_mpage\_readpage函数详细分析》

另外参考：<http://blog.csdn.net/yangp01/article/details/5417569>

mpage\_readpages

mpage\_readpage

上述几个函数，作为页缓存和块缓存之间的衔接函数，下发读bh需求到Block层，并填充pagecache的page。

### 参考

《PLKA 16.5》

## 数据同步

backing-dev.c

page-writeback.c

fs-writeback.c

### 数据结构

struct bdi\_writeback

### 周期性flush线程

wb\_workfn的结尾有一个5秒（dirty\_writeback\_interval）的定时器，定时器又会触发wb\_workfn这个work。

### BDI对象创建和初始化

创建：

blk\_alloc\_queue

🡪 blk\_alloc\_queue\_node

🡪 bdi\_alloc\_node (申请backing\_dev\_info对象)

🡪 bdi\_init

初始化：

bdi\_register\_owner

🡪 bdi\_register

🡪 bdi\_register\_va

# 磁盘存储

### 应用视角资源管理

#### Mtd

1. Mtd给YAFFS2的接口

struct yaffs\_dev {

void \*driver\_context; /\* struct mtd\_info \*/

mtd->read： 读数据

mtd->read\_oob： 读OOB及数据

mtd->write\_oob： 写OOB及数据

mtd->block\_markbad： 把块标记为坏块

mtd->block\_isbad： 查询块是否为坏块

mtd->erase： 擦块

mtd->type： MTD\_NANDFLASH

mtd->flags： MTD\_WRITEABLE

mtd->oobsize： oob大小

mtd->erasesize： 擦写单元，128K

mtd->writesize： chunk大小

mtd->size：

1. Mtd给YAFFS2的接口

struct jffs2\_sb\_info {

struct mtd\_info \*mtd;

mtd->read 读数据

mtd->write： 写数据

mtd\_writev： 写数据

mtd->read\_oob： 读OOB及数据（NAND\_FLASH）

mtd->write\_oob： 写OOB及数据（NAND\_FLASH）

mtd->sync： 同步

mtd->erase： 擦块

mtd->block\_isbad 判断块是否是坏块

mtd->block\_markbad: 把块标记为坏块

mtd->point：

mtd->unpoint：

### 硬件视角资源管理

### 硬件驱动

## 网络

### 总体体系结构

1. 网络子系统设计分层模型

用户空间应用层

内核空间应用层

传输层

网络层

网络访问层

各层之间通过大量的钩子函数，面向对象的思想调用实现多种组合，降低耦合。

1. 网络子系统数据传输分层模型

各层数据分首部与净荷。高层数据的首部与净荷是底层数据的首部。

以太网：帧

网络层：数据包

传输层：

1. 网络命名空间(namespace) struct net

类似于系统的命名空间概念，建立网络的虚拟试图。通常只需要一个namespace。

因此网络子系统各对象不是全局的，而是依附于某一个namespace而存在。类似的概念类似于内存子系统的Node。

struct list\_head dev\_base\_head; 本space的net\_device链表

struct hlist\_head \*dev\_name\_head; 本space的net\_device的name hash链表

struct hlist\_head \*dev\_index\_head; 本space的net\_device的index hash链表

1. 套接字缓冲区 socket buffer --- struct sk\_buff与所有分组数据buffer

sk\_buff指向所有分组数据的buffer，贯穿整个协议栈，免除在层间进行数据拷贝，提升性能。

Head: 所有数据的头，在层间移动时不变

End: 所有数据的尾，在层间移动时不变

Data: 特定层协议数据的头，在层间移动时**变化**

Tail: 特定层协议数据的尾，在层间移动时不变

发送时：sk\_buffer和整个分组数据buffer，由高层协议分配，何时释放？

接受时：sk\_buffer和整个分组数据buffer，由低层协议分配，何时释放？

1. 套接字缓冲区 socket buffer --- struct sk\_buff

### 网络访问层

1. 网络设备 struct net\_device
   1. alloc\_netdev 申请一个设备
   2. register\_netdev / register\_netdevice完成注册，通常由网络设备驱动程序调用。

1. API与NAPI

API 是传统的IRQ处理方式。

NAPI是高速网卡提供的一种IRQ与轮询的组合方式，以防止过多的中断产生的“中断风暴”。NAPI的逻辑：依次轮询轮询表中的设备(napi\_struct)，依次调用设备的poll接口进行轮询收包。支持NAPI方式的设备需要支持：

* 1. 仅禁用收包的IRQ。
  2. 禁用收包的IRQ后，网卡也能够自动收包保存在DMA环形缓冲区(Rx缓冲区)中。

1. napi\_struct

该对象是NAPI轮询表中的设备表示。

1. skb接收和发送队列，每CPU一个

softnet\_data {

struct sk\_buff\_head input\_pkt\_queue;

struct napi\_struct backlog;

/\* 伪网络设备，用于在poll(process\_backlog)中进行轮询操作，用于处理那些非NAPI 设备收到的skb队列\*/

struct list\_head poll\_list; 轮询列表（napi\_struct对象）

}

sk\_buff\_head 是队列头。

1. 接收分组

中断上下文：

* 1. net\_interrupt, net\_rx (驱动)
  2. netif\_rx (dev.c)：将skb置于softnet\_data队列中。

SoftIRQ上下文：

从softnet\_data队列中找到skb，并找到合适的网络层协议，并调用

* 1. do\_softirq 🡪 net\_rx\_actiion (dev.c)：

🡪 process\_backlog 伪NAPI设备

🡪 \_\_skb\_dqueue

🡪netif\_receive\_skb

🡪 deliver\_skb 🡪 packet\_type 🡪 func （遍历IPv4等网络层协议函数）

## 其它外设

### 异常中断模型

1. 为什么要中断（包括同步与异步）？

更高效地实现设备驱动程序和，引起处理器自身对异常或错误的关注。

1. 同步中断与异步中断区别

也叫异常与硬件IRQ，分别来自于CPU与外部硬件。

1. 中断向量与IRQ号
   1. IRQ与异常进行统一地“中断向量”编号。由于各CPU给IRQ的中断编号限制，因此多个硬件IRQ会共享同一中断向量。
   2. 中断向量中IRQ与异常的划分是CPU内部规定**，但是要通过编程告诉中断控制器？**
   3. IRQ与中断向量号的映射也是CPU内部规定**，但是要通过编程告诉中断控制器？**
   4. 硬件设备只能编程指定IRQ号**，决定了IRQ号就决定了中断向量？**
2. 中断控制器

负责接收外部信息，对IRQ🡪 中断号转换。

* 1. 经典IA-32 8259A 中断控制器支持15个IRQ，类似于IO-APIC。
  2. IA-32 IO-APIC，支持224个IRQ。IO-APIC位于CPU之外，Local-APIC位于CPU内部，还负责产生CPU异常。

平台最大IRQ数目为 NR\_IRQS。

1. 中断模型的设计目标
   1. 使用汇编处理与系统结构高度相关的底层细节。
   2. 使用抽象接口处理绝大部分与体系结构无关的对象。
2. 中断处理
   1. 进入路径
      1. 模式及堆栈转换（CPU完成）
      2. 保存寄存器(CPU与汇编 entry.S完成)
   2. 中断服务例程(ISR)，也即中断处理程序，C语言
      1. 关键操作必须关中断。**（是否可再被本中断打断？）**
      2. 非关键操作，允许启用中断。（可能被其它事件中断，中断间不能彼此干扰）
      3. 可延迟操作。**（不在ISR中？）**
   3. 退出路径
      1. 调度和信号投递（汇编 entry.S调用C例程完成）。
      2. 还原寄存器(CPU与汇编 entry.S完成)
      3. 模式及堆栈转换（CPU处理）

其它想法：

1. 如何收集其它CPU正在运行的任务的eip，堆栈以及寄存器信息。可以通过发送一个IPI中断，中断正在运行的任务，中断处理的进入路径会收集所有信息。然后分析调用栈。

# RT

## 两个指标

中断延迟：从一个外部事件发生到相应的中断处理函数的第一条指令开始执行所需要的时间。（包括中断禁止，其实如果重要中断被打断的话，也会影响延迟性。如果重要中断被禁止）

抢占延迟：有时也称调度延迟，抢占延迟就是从一个外部事件发生到相应的处理该事件的任务的第一条命令开始执行的时间。

## 制约实时性的因素(不确定性)

### 影响系统实时性的因素

1. CPU Cache。（我觉得不是）
2. 虚存。（我觉得不是）
3. O(n)算法。

### Linux中限制因素

#### 中断延迟

1. 中断关闭。

在同步操作或者主动关闭中断。

#### 抢占延迟

1. 内核不可抢占（内核抢占开关未打开或使用preempt\_disable或一些同步机制spinlock中）。

时间片也不管用，时钟中断判断时间片超时后只是置位该任务的NEED\_SCHED标志，但是如果不返回用户态，也不会切换任务；另外有更高优先级任务也不会抢占该任务。

1. 中断总是最高优先级的，如果有多个中断嵌套， FIFO任务的执行就会延迟。另外软中断的执行时间也是不可预知的。

解决此问题的办法就是中断线程化。

1. 调度算法和调度点

调度点不多，只有系统调用或自动放弃时才会切换。

1. 2.4的O(n)算法，导致一个高优先级任务的执行时间不可控。O(1)已经解决。

## 检测实时指标的工具

## 现存的Linux实时技术

### RTLinux

RTLinux是著名的研究机构FSMLab研发的一款实时Linux，既有GPL和Free版本，又有商业版本。它使用的实现方式是子内核方法，即把Linux内核作为一个新实现的子内核的闲暇任务，子内核位于Linux内核和硬件抽象层之间，实时任务运行于子内核之上，只有当没有实时任务需要运行时，Linux内核才有机会运行。

特别是对中断的管理，它采用了一种软件的方式来处理Linux内的中断关闭，当Linux内核关闭中断后，并不是真正地屏蔽了硬件中断，相反，它使用了一个变量来保存Linux内核的中断标志位，Linux内核的开关中断只是影响了该变量，硬件的中断由子内核来接管，当Linux内核关闭了中断，子内核仍然可以响应任何中断，只是如果子内核不需要处理的中断才交给Linux内核来处理，如果Linux内核关闭了中断，子内核将记录该中断并在Linux内核打开中断后提交它处理。

在RTLinux里，每一个实时任务都是内核线程，运行在内核空间，RTLinux提供了一套专门的机制来在实时任务和普通的Linux任务之间进行进程间通信。

这种子内核的实现提供了非常好的实时性，完全是一个硬实时的Linux。

### TimeSys Linux

Timesys很早就发布了实时Linux的商业版以及GPL版，它采用了与RTLinux完全不同的实现方式。前面已经提到了标准Linux内核的实时限制，TimeSys Linux就是通过消除这些限制来达到实时性的。它把中断(IRQ)和软中断(softIRQ)全部线程化并赋予不同的优先级，实时任务可以有比中断线程更高的优先级，它使用Mutex替代spinlock来使得自旋锁完全可抢占。它也对调度器做了优化使它是O(1)的（注：因为使用2.4内核）。由于中断已经线程化了，很多中断关闭就没必要了，因而消除了很多中断关闭区域。它还实现了对CPU和网络资源的预定来改善实时性。后面将说的Ingo's RT patch就是借鉴这些思路来实现实时性的。

这种实现方式保持了全部的Linux应用编程模式，实时应用和普通的应用采用同样的编程方式，使用同样的API，只是实时任务需要明确指定自己的优先级与调度策略。但是这种实现方式也有弊病，那就是它满足硬实时性有一定的困难，因为即使中断关闭和不可抢占区大为减少，但是还是存在，一些中断还是无法线程化，如时钟中断等。

### Ingo's RT patch

Ingo's RT patch是又一个Linux实时实现，它采用了与TimeSys完全相同的技术路线，而且有一些实现是基于TimeSys的源代码的，如IRQ和softirq线程化。但是它与前面提到的两个实时实现不同的地方是，它可能并入到标准Linux内核。在最新的标准内核Linux 2.6.11中，已经出现了这个补丁曾经包含的部分代码，如IRQ子系统，那是IRQ和softirq线程化的基础，已经隐含了一些线程化的代码，如自愿抢占代码，那是2.4的低延迟补丁（low latency patch）和Ingo的一些自愿抢占代码以及Robert Love的锁分解补丁的集合。总的来说，它包括：

1. 中断线程化（包括IRQ和softirq）
2. 用Mutex取代spinlock。
3. 优先级继承和死锁检测
4. 等待队列优先级化
5. 大内核锁（BKL-Big Kernel Lock）可抢占等。
6. Robert Love的锁分解补丁。

# 性能设计和实现

1. Sk\_buff贯穿整个系统免拷贝使性能提升。
2. 减少Sk\_buff结构体大小，如64位CPU下，使用int替代char \*。
3. \_\_\_\_cacheline\_aligned\_in\_smp

# CPU结构

## Pipeline

Burst，ILP，TLP，Intel NetBurst微结构，再谈Intel CPU微结构

## TLB

## Cache line

Locality，cache coherency，

## 四大问题

* + 1. program model
    2. cache coherency
    3. interconnect
    4. lock & spin…..
    5. 并发问题，局部性

# 内存结构

## ARM内存初始化

### 主要函数

setup\_kernel

🡪page\_address\_init

🡪setup\_arch

🡪 parse\_early\_param();

🡪 early\_mem

🡪 arm\_add\_memory 初始化 meminfo

🡪 setup\_dma\_zone

🡪 sanity\_check\_meminfo

🡪 arm\_memblock\_init

🡪 paging\_init 初始化页表、启用bootmem分配器

🡪 bootmem\_init

🡪 arm\_bootmem\_init 启用bootmem分配器

🡪 arm\_memory\_present 从bootmem申请各mem\_section对象

🡪 sparse\_init 初始化各mem\_section对象，如申请memmap数组

🡪 arm\_bootmem\_free

🡪 free\_area\_init\_node

🡪mm\_init\_owner(&init\_mm, &init\_task);

🡪mm\_init\_cpumask(&init\_mm);

🡪build\_all\_zonelists(NULL, NULL); **用于结点和内存域的初始化**

🡪page\_alloc\_init();

🡪mm\_init(); **用于停用bootmem分配器并迁移到伙伴系统**

🡪 **mem\_init**

🡪 **free\_unused\_memmap 释放bank之间无用区域对应的memmap page结构体回bootmem**

🡪kmem\_cache\_init\_late();

🡪page\_cgroup\_init();

🡪debug\_objects\_mem\_init();

🡪kmemleak\_init();

🡪setup\_per\_cpu\_pageset();

🡪numa\_policy\_init();

## ARM初始化

### 启动条件

通常从系统上电到运行到linux kenel这部分的任务是由boot loader来完成。Boot loader在跳转到kernel之前要完成一些限制条件:

1、CPU必须处于SVC(supervisor)模式,并且IRQ和FIQ中断必须是禁止的；

2、MMU(内存管理单元)必须是关闭的,此时虚拟地址对应物理地址；

3、数据cache(Data cache)必须是关闭的；

4、指令cache(Instruction cache)没有强制要求；

5、CPU通用寄存器0(r0)必须是0;

6、CPU通用寄存器1(r1)必须是ARM Linux machine type；

7、CPU通用寄存器2(r2)必须是kernel parameter list的物理地址；

### head.S的汇编

确定processor type

确定machine type

创建临时页表

调用平台特定的\_\_cpu\_flush函数

开启mmu

切换数据

最终跳转到start\_kernel

.align 2

.type \_\_switch\_data, %object

\_\_switch\_data:

.long \_\_mmap\_switched

.long \_\_data\_loc @ r4

.long \_data @ r5

.long \_\_bss\_start @ r6

.long \_end @ r7

.long processor\_id @ r4

.long \_\_machine\_arch\_type @ r5

.long \_\_atags\_pointer @ r6

.long cr\_alignment @ r7

.long init\_thread\_union + THREAD\_START\_SP @ sp

@ 下面的汇编分别从系统中读取相关值，保存到上述变量中

/\*

\* The following fragment of code is executed with the MMU on in MMU mode,

\* and uses absolute addresses; this is not position independent.

\*

\* r0 = cp#15 control register

\* r1 = machine ID

\* r2 = atags pointer

\* r9 = processor ID

\*/

\_\_mmap\_switched:

adr r3, \_\_switch\_data + 4

ldmia r3!, {r4, r5, r6, r7}

cmp r4, r5 @ Copy data segment if needed

1: cmpne r5, r6

ldrne fp, [r4], #4

strne fp, [r5], #4

bne 1b

mov fp, #0 @ Clear BSS (and zero fp)

1: cmp r6, r7

strcc fp, [r6],#4

bcc 1b

ARM( ldmia r3, {r4, r5, r6, r7, sp})

THUMB( ldmia r3, {r4, r5, r6, r7} )

THUMB( ldr sp, [r3, #16] )

str r9, [r4] @ Save processor ID

str r1, [r5] @ Save machine type

str r2, [r6] @ Save atags pointer

bic r4, r0, #CR\_A @ Clear 'A' bit 清除r0中的CR\_A 位并将值存到r4中

stmia r7, {r0, r4} @ Save control register values

@r0存储到了cr\_alignment, r4存储到了cr\_no\_alignment

b start\_kernel

ENDPROC(\_\_mmap\_switched)

# 性能优化

## CPU密集型

一段程序的CPU所需执行时间 = (CPU时钟周期数/时钟频率) =

## IO密集型