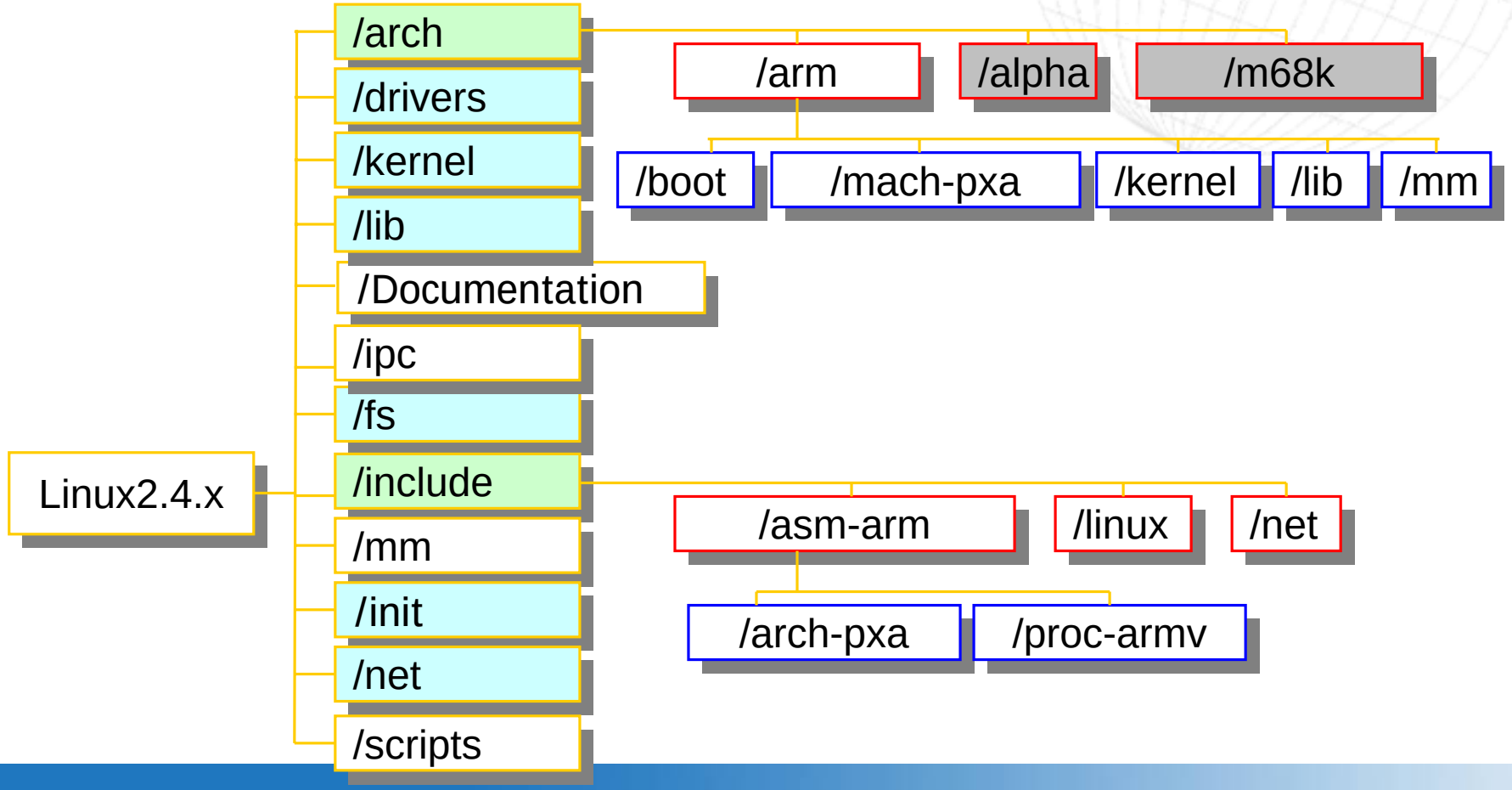
ARM上的Linux内核及启动过程

linux 2.4 的内核目录结构



读懂linux内核源码

- linux内核庞大，结构复杂
  - 对linux内核的统计：接近1万个文件，4百万行代码
- 内核编程习惯（技巧）不同于应用程序

(uC)linux内核的C代码

- Linux内核的主体使用GNU C，在ANSI C上进行了扩充
  - Linux内核必须由gcc编译编译
  - gcc和linux内核版本并行发展，对于版本的依赖性强
- 内核代码中使用的一些编程技巧，在通常的应用程序中很少遇到

GNU C的扩充举例

- 从C++中吸收了inline和const关键字
- ANSI C代码与GNU C中的保留关键字冲突的问题可以通过双下划线（__）解决
  - 例如：inline 等价于 __inline__、asm等价于 __asm__
- 结构体（struct）的初始化

结构体初始化

```
struct sample {
        int member_int;
        char *member_str;
        void (*member_fun)(void);
};
```

ANSI C中的实现

```
struct sample inst_c={
        100,            //member_int
        NULL,                   //*member_str;
        myfunc      //void (*member_fun)(void);
};
```

C99中的实现

```
struct sample inst_c99 = {
        .member_int = 100,
        .member_fun = myfun,
};
```

GCC中的实现

```
struct sample inst_gcc = {
        member_fun: myfun,
        member_int: 100,
};
```

与C99中的用法类似，不必关心struct定义的中的实际的顺序和其他未定义的数据，在复杂的结构体初始化的时候很有优势。

宏定义的灵活使用（1）

- 虽然GCC中定义了inline关键字，但是，宏操作（#define）仍然在系统中大量使用
- 举例：

```
#define DUMP_WRITE(addr,nr)        do\
    { memcpy(bufp,addr,nr); bufp += nr; } while(0)
```

应用DUMP_WRITE，就像使用C的函数一样：

```
if(addr)
    DUMP_WRITE(addr, nr);
else…
```

但是，如果如通过下的定义，都不能满足上述的情况

定义1：

```
#define DUMP_WRITE(addr,nr) memcpy(bufp,addr,nr);\          bufp += nr ;
```

定义2：

```
#define DUMP_WRITE(addr,nr) {memcpy(bufp,addr,nr);\         bufp += nr;}
```

宏定义的灵活使用（2）

```
#define OFFSETOF(strct, elem)    \
        ((long)&(((struct strct *)0)->elem))
```

- 1、((struct strct *)0) 结构体strct的指针
- 2、&((struct strct *)0)->elem)成员的地址，也就是相对于0的偏移
- 3、结果：OFFSETOF(strct,elem)返回的是，结构体strct中成员elem的偏移量

C语言中goto的使用

- 在应用程序的C编程中，为了保证程序的模块化，建议不使用goto
- 内核代码需要兼顾到效率，所以，大量使用goto
  - 整个内核的比例大概是每260行一个goto语句——速度优先
  - 短距离的goto

Linux内核加载过程

通常，Linux内核是经过gzip压缩之后的映象文件

- bootloader复制压缩内核到内存空间
- 内核自解压
- 运行内核

编译完成的Linux内核在哪里？

- ./vmlinux，elf格式未压缩内核
- arch/arm/boot/compressed/vmlinux，压缩以后的elf格式内核
- arch/arm/boot/zImage，压缩内核

压缩内核（zImage）的入口

- /arch/arm/boot/compressed/vmlinux.lds 文件为编译器指定连接（link）顺序
- ENTRY(_start)，压缩内核从.start段开始
- 在arch/arm/boot/compressed/head.S中
  - 检测系统空间
  - 初始化C代码空间
  - 跳转到C代码decompress_kernel，arch/arm/boot/compressed/misc.c中

解压之前的串口输出

- include/asm-arm/arch-s3c2410/uncompress.h定义了puts作为串口输出函数
- 解压结束以后，跳转到r5：解压之后内核的起始地址

开始真正的Linux内核

- 入口在arch/arm/kernel/head-armv.S
- 查找处理器类型
  - __lookup_processor_type
  - __lookup_architecture_type
- 初始化页表（__create_page_tables）
- 初始化C代码空间
- 跳转到C代码中，start_kernel

ARM的MMU

内存管理单元memory management unit

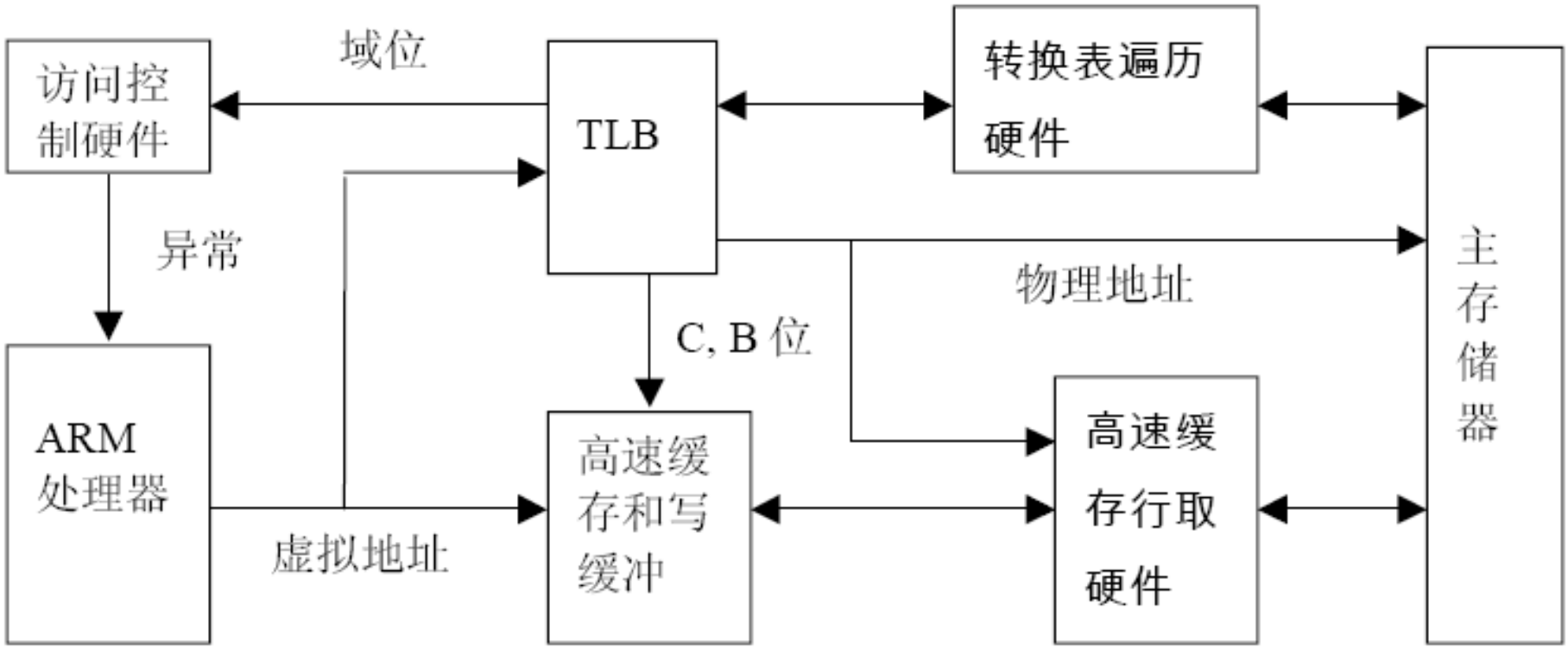
- 虚拟地址到物理地址的映射
- 存储器访问权限
- 控制Cache

通过MMU的访存

- MMU 先查找TLB（Translation Lookaside Buffers）中的虚拟地址表
- 如果TLB 中没有虚拟地址的入口，硬件从主存储器中的转换表中获取转换和访问权限

开始MMU之前必须创建转换表

ARM的MMU访存原理



ARM的MMU页表格式

MMU 支持基于节或页的存储器访问：

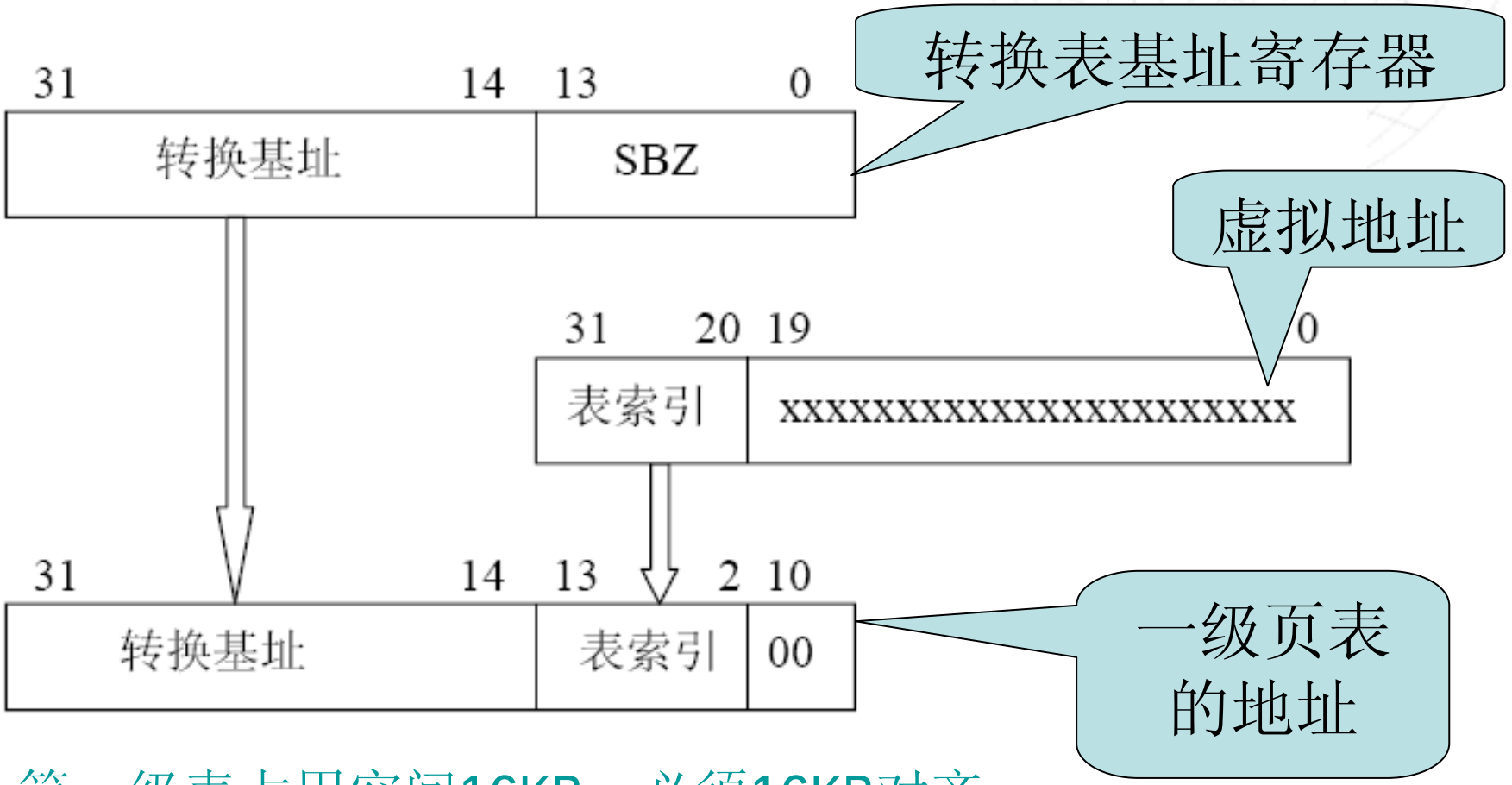
- 节（Section） 1MB 的存储器块
- 大页（Large page）64KB 的存储器块
- 小页（Small page）4KB 的存储器块
- 微页（Tiny page）1KB 的存储器块

页表的级别

存在主存储器内的转换表有两个级别：

- 第一级表 存储节转换表和指向第二级表的指针
- 第二级表
  - 存储大页和小页的转换表。
  - 存储微页转换表

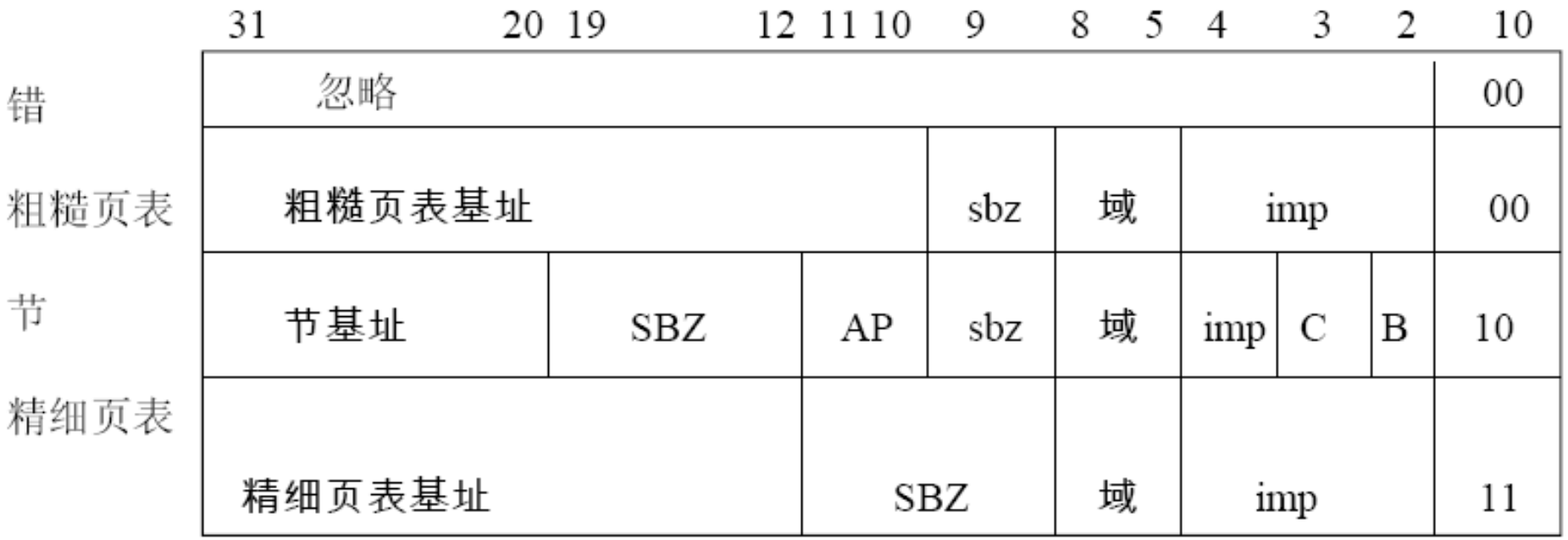
一级页表的地址



第一级表占用空间16KB，必须16KB对齐

第一级描述符

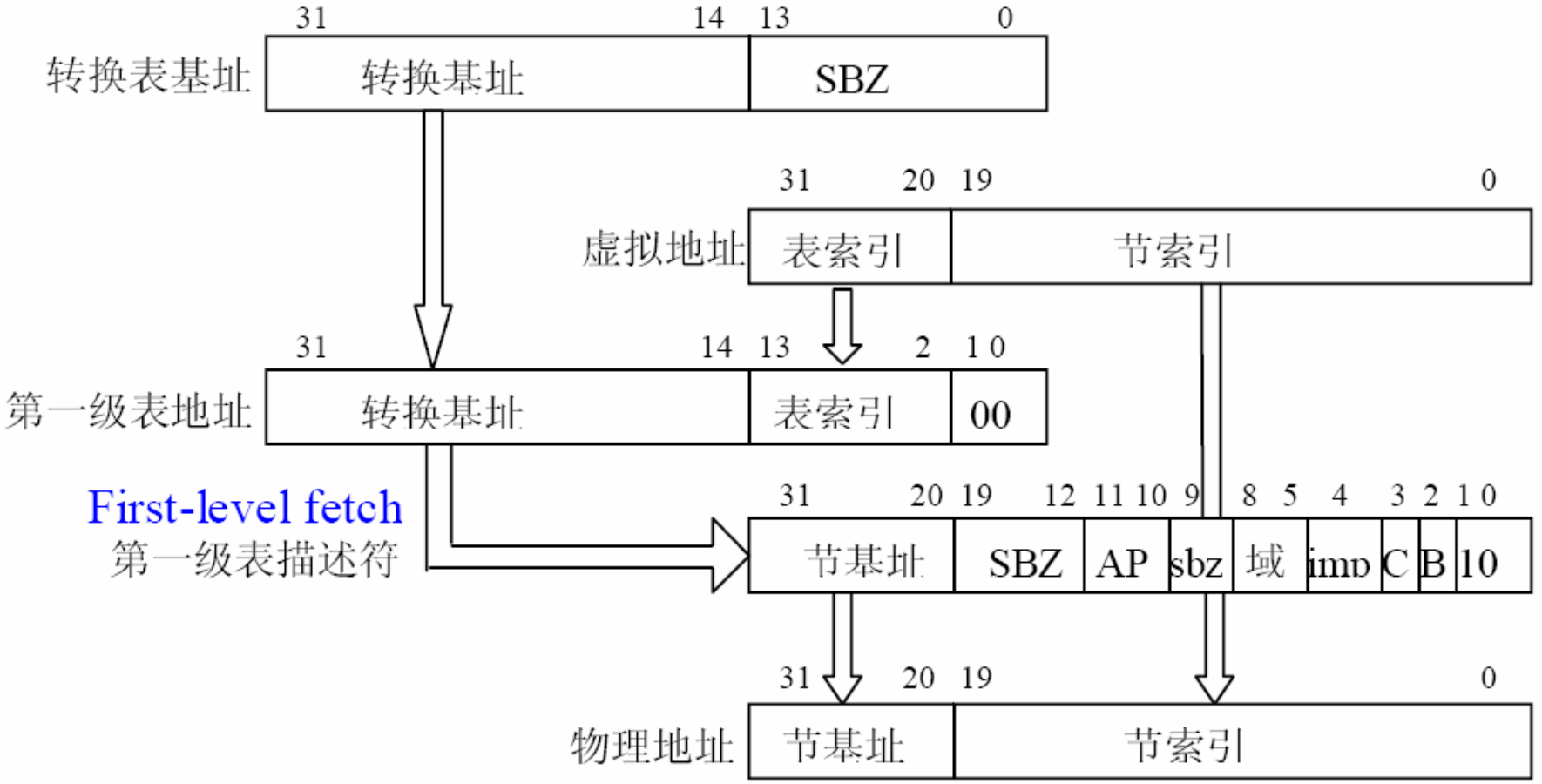
- 一级表每个入口描述了它所关联的1MB 虚拟地址是如何映射的



节描述符

- Bits[1:0] 描述符类型（10b 表示节描述符）
- Bits[3:2] 高速缓存（cache）和缓冲位（buffer）
- Bits[4] 由具体实现定义
- Bits[8:5] 控制的节的16 种域之一
- Bits[9] 现在没有使用，应该为零
- Bits[11:10] 访问控制（AP）
- Bits[19:12] 现在没有使用，应该为零
- Bits[31:20] 节基址，形成物理地址的高12 位

节的转换过程


转换表基址
31
14
13
0
转换基址
SBZ
虚拟地址
31
20
19
0
表索引
节索引
第一级表地址
31
14
13
2
1
0
转换基址
表索引
00
First-level fetch
第一级表描述符
31
20
19
12
11
10
9
8
5
4
3
2
1
0
节基址
SBZ
AP
sbz
域
imp
C
B
10
物理地址
31
20
19
0
节基址
节索引

__create_page_tables（1）

pgtbl r4 @ page table address 0x30008000-0x4000

```
    mov     r0, r4        @r0=0x30004000
    mov     r3, #0
    add     r2, r0, #0x4000
1:  str     r3, [r0], #4
    str     r3, [r0], #4
    str     r3, [r0], #4
    str     r3, [r0], #4
    teq     r0, r2
    bne     1b
```

把一级页表0x30004000－ 0xa0080000清空

__create_page_tables（2）

krnladr r2, r4　　@ start of kernel

@ r4=0xa0004000，r2 = 内核起始地址所在1MB对齐空间，0x30000000

add r3, r8, r2　@ flags + kernel base

@r8 为从处理器信息中得到的MMU 页表标志，r8=0xc0e, r3=0x30000c0e

str r3, [r4, r2, lsr #18]@ identity mapping

@地址:0x300068000, value:0x30000c0e

__create_page_tables（3）

```
 add     r0, r4, #(TEXTADDR & 0xff000000) >> 18
@ start of kernel

bic     r2, r3, #0x00f00000
str     r2, [r0]            @ PAGE_OFFSET + 0MB
add     r0, r0, #(TEXTADDR & 0x00f00000) >> 18
str     r3, [r0], #4        @ KERNEL + 0MB
……
```

映射表内容

物理地址	数据
0x3000e8000	0x30000c0e
0x3000f000	0xc0000c0e
0x3000f004	0xc0100c0e
0x3000f008	0xc0200c0e
0x3000f00c	0xc0300c0e

映射结果

物理地址	
偏移量	大小
0x30000000	1MB
0x30100000	3MB

虚拟地址	
偏移量	大小
0x30000000	1MB
0xc0000000	1MB
0xc0100000	3MB

进入C代码

init/main.c中的start_kernel函数，进入到了Linux内核代码中。

- printk函数
- 重新初始化页表
- 初始化中断，trap_init
- 设置系统定时器、控制台…
- 创建内核进程 init