# 9. 内核访问外设I/O资源

## 9.1 MIPS的内存映射

在32位MIPS体系结构下，最多可寻址4GB地址空间。这4GB空间的分配是怎样的呢？让我们看下面这张图：

 　　　　　　+----------------------------------------------+

　　　　　　|　　　　　　　　　　　　　　　　　　　　　　　| 0xFFFFFFFF

　　　　　　|　　　　　　　　　　　　　　　　　　　　　　　|

　　　　　　|　　　　　　　　　　　　　　　　　　　　　　　|

　　　　　　|　Kernel Space Mapped Cached　　　　　　 　|

　　　　　　|　　　　　　　　　　　　　　　　　　　　　　　|

　　　　　　|　　　　　　　　　　　　　　　　　　　　　　　|

0xC0000000 |　　　　　　　　　　　　　　　　　　　　 　　|

　　　　　　+----------------------------------------------+

　　　　　　|　　　　　　　　　　　　　　　　　　　　　　　| 0xBFFFFFFF

　　　　　　| Kernel Space Unmapped Uncached 　　 　 　 　 |

　　　　　　|　　　　　　　　　　　　　　　　　　　　　　　|

0xA0000000 +----------------------------------------------+

　　　　　　|　　　　　　　　　　　　　　　　　　　　　　　| 0x9FFFFFFF

　　　　　　| Kernel Space Unmapped Cached 　　 　 　 |

　　　　　　|　　　　　　　　　　　　　　　　　　　　　　　|

0x80000000 +----------------------------------------------+

　　　　　　|　　　　　　　　　　　　　　　　　　　　　　　| 0x7FFFFFFF

　　　　　　|　　　　　　　　　　　　　　　　　　　　　　　|

　　　　　　|　　　　　　　　　　　　　　　　　　　　　　　|

　　　　　　|　　　　　　　　　　　　　　　　　　　　　　　|

　　　　　　|　　　　User Space　　　　　　　　　 　　　　|

　　　　　　|　　　　　　　　　　　　　　　　　　　　　　　|

　　　　　　|　　　　　　　　　　　　　　　　　　　　　　　|

　　　　　　|　　　　　　　　　　　　　　　　　　　　　　　|

　　　　　　|　　　　　　　　　　　　　　　　　　　　　　　|

　　　　　　|　　　　　　　　　　　　　　　　　　　　　　　|

0x00000000 +----------------------------------------------+

上图是MIPS处理器的逻辑寻址空间分布图。我们看到，2GB以下的地址空间，也就是从0x00000000到0x7FFFFFFF的这一段空间，为User Space，可以在User Mode下访问，当然，在Kernel Mode下也是可以访问的。程序在访问User Space的内存时，会通过MMU的TLB，映射到实际的物理地址上。也就是说，这一段逻辑地址空间和物理地址空间的对应关系，是由MMU中的TLB表项决定的。

 从0x80000000到0xFFFFFFFF的一段为Kernel Space，仅限于Kernel Mode访问。如果在User Mode下试图访问这一段内存，将会引发系统的一个Exception。MIPS的　　Kernel Space又可以划分为三部分。首先是通过MMU映射到物理地址的1GB空间，地址范围从0xC0000000到0xFFFFFFFF。这1GB空间可以用来访问实际的DRAM内存，可以为操作系统的内核所用。

MIPS的Kernel Space中，还有两段特殊的地址空间，分别是从0x80000000到0x9FFFFFFF的Kernel Space Unmapped Cached  和  0xA0000000到0xBFFFFFFF的Kernel Space Unmapped Uncached。之所以说它们特殊，是因为这两段逻辑地址到物理地址的映射关系是硬件直接确定的，不通过MMU，而且两段实际上是重叠的，均对应0x00000000到0x20000000的物理地址。那么，为什么一段同样的物理地址有两个逻辑地址对应呢？它们的区别又在哪里呢？

原来，这是MIPS的设计特色之一。软件在访问Kernel Space Unmapped Uncached这段地址空间的时候，不经过MIPS的Cache。这样，虽然速度会比较慢，但是，对于硬件I/O寄存器来说，就不存在所谓的Cache一致性问题。Cache一致性问题，是指硬件将某个地址的内容跳过软件而改变了，Cache中的内容尚未同步。这样，如果软件读取该地址，有可能从Cache中获取到错误的内容。将硬件I/O寄存器设定在这段地址空间，就可以避免Cache一致性带来的问题。MIPS的程序上电启动地址0xBFC00000，也落在这段地址空间内。——上电时，MMU和Cache均未初始化，因此，只有这段地址空间可以正常读取并处理。

另一段特殊的地址Kernel Space Unapped Cached，与前者类似，直接映射到0x00000000到0x20000000，与Kernel Space Unmapped Uncached重叠。因为通过Cache，这段地址空间的访问速度比前者为快。一般地，这段内存空间用于内核代码段，或者内核中的堆栈。

显然地，当工程师们换算Kernel Space中的这两段的物理地址和逻辑地址时，只需要改变地址的高3bit就可以了。

那么，什么时候需要使用物理地址，什么时候需要使用逻辑地址呢？我们知道，逻辑地址是程序中访问的内存地址，譬如，下面的这条指令：

lw a0, 128(t2)

这条指令的内容是从t2寄存器内的地址 + 偏移128字节处，读取一个word (4Byte)到寄存器a0内。如果t2的值为0x88200100，则最终访问的物理地址为0x88200180。

而物理地址，从工程上可以理解为，将逻辑分析仪连接到内存总线(Memory Bus)上，逻辑分析仪指示的地址，就是物理地址。假如，在上一个例子中，我们把逻辑分析仪接到处理器的前端内存总线，我们就可以看到，执行该指令时，系统访问的物理地址为0x08200180。物理地址和逻辑地址的换算，不仅限于电子工程师在设计硬件线路时需要。在内核工程师编写支持DMA的外部设备驱动时，需要将向操作系统申请到的数据缓冲区地址（当然，这是一个逻辑地址）转换为物理地址，并“告诉”相关外设。这样，外设就可以在收到数据后，使用DMA模式储存在系统的主存中，并向系统发起一个IRQ。操作系统在IRQ的handler中，从外设的相应IO寄存器读取到这段内存的地址（当然，是物理地址）并转换为逻辑地址并处理之。这个过程中，如果没有正确使用和分辨物理地址和逻辑地址，驱动程序便会导致内核的一个panic错误。

物理地址到逻辑地址的映射关系是由什么决定的呢？除了上面提到的两段Unmapped的地址空间，其余都是由TLB确定的，由MMU来执行。

默认外设I/O资源是不在Linux内核空间中的（如sram或硬件接口寄存器等），若需要访问该外设I/O资源，必须先将其地址映射到内核空间中来，然后才能在内核空间中访问它。

Linux内核访问外设I/O内存资源的方式有两种：动态映射(ioremap)和静态映射(map\_desc)。

## 9.2动态映射(ioremap)方式

动态映射方式是大家使用了比较多的，也比较简单。即直接通过内核提供的ioremap函数动态创建一段外设I/O内存资源到内核虚拟地址的映射表，从而可以在内核空间中访问这段I/O资源。

ioremap宏定义在asm/io.h内：

#define ioremap(cookie,size) \_\_ioremap(cookie,size,0)

\_\_ioremap函数原型为(arm/mm/ioremap.c)：

void \_\_iomem \* \_\_ioremap(unsigned long phys\_addr, size\_t size, unsigned long flags);

phys\_addr：要映射的起始的IO地址

size：要映射的空间的大小

flags：要映射的IO空间和权限有关的标志

该函数返回映射后的内核虚拟地址(3G-4G). 接着便可以通过读写该返回的内核虚拟地址去访问之这段I/O内存资源。

示例程序regeditor\_drv.c。

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/fs.h>

#include <linux/init.h>

#include <linux/delay.h>

#include <linux/irq.h>

#include <asm/uaccess.h>

#include <asm/irq.h>

#include <asm/io.h>

#include <linux/device.h>

#include <linux/sched.h>

#include <linux/interrupt.h>

#include <linux/poll.h>

#include <linux/spinlock.h>

#define KER\_RW\_R8 100

#define KER\_RW\_R16 101 /\* 切记！cmd命令值不能乱定义 \*/

#define KER\_RW\_R32 102

#define KER\_RW\_W8 103

#define KER\_RW\_W16 104

#define KER\_RW\_W32 105

static int major;

static struct class \*class;

static long ker\_rw\_ioctl( struct file \*file, unsigned int cmd, unsigned long arg)

{

volatile unsigned char \*p8;

volatile unsigned short \*p16;

volatile unsigned int \*p32;

unsigned int val;

unsigned int addr;

unsigned int buf[2];

copy\_from\_user(buf, (const void \_\_user \*)arg,8);

addr = buf[0];

val = buf[1];

p8 = (volatile unsigned char \*)ioremap(addr, 4);

p16 = (volatile unsigned short \*) p8;

p32 = (volatile unsigned int \*)p8;

switch (cmd)

{

case KER\_RW\_R8:

{

val = \*p8;

copy\_to\_user((void \_\_user \*)(arg+4), &val, 4);

break;

}

case KER\_RW\_R16:

{

val = \*p16;

copy\_to\_user((void \_\_user \*)(arg+4), &val, 4);

break;

}

case KER\_RW\_R32:

{

val = \*p32;

copy\_to\_user((void \_\_user \*)(arg+4), &val, 4);

break;

}

case KER\_RW\_W8:

{

\*p8 = val;

break;

}

case KER\_RW\_W16:

{

\*p16 = val;

break;

}

case KER\_RW\_W32:

{

\*p32 = val;

break;

}

}

iounmap(p8);

return 0;

}

static struct file\_operations ker\_rw\_ops = {

.owner = THIS\_MODULE,

.unlocked\_ioctl = ker\_rw\_ioctl,

};

static int ker\_rw\_init(void)

{

major = register\_chrdev(0, "ker\_rw", &ker\_rw\_ops);

class = class\_create(THIS\_MODULE, "ker\_rw");

/\* 为了让mdev根据这些信息来创建设备节点 \*/

device\_create(class, NULL, MKDEV(major, 0), NULL, "ker\_rw"); /\* /dev/ker\_rw \*/

return 0;

}

static void ker\_rw\_exit(void)

{

device\_destroy(class,MKDEV(major, 0));

class\_destroy(class);

unregister\_chrdev(major, "ker\_rw");

}

module\_init(ker\_rw\_init);

module\_exit(ker\_rw\_exit);

MODULE\_LICENSE("GPL");

编写makefile。

obj-m := regeditor\_drv.o

#定义目录变量

KDIR := /Workstation/tools/kernel/linux-3.0.82-openloongson

PWD := $(shell pwd)

all:

# make文件

make -C $(KDIR) M=$(PWD) modules ARCH=mips CROSS\_COMPILE=mipsel-linux-

clean:

rm -rf \*.o \*.mod.c \*.ko

驱动编译后，在开发板加载regeditor\_drv.ko,后运行以下测试程序regeditor

/\*regeditor.c\*/

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <stdio.h>

#include <poll.h>

#include <signal.h>

#include <sys/types.h>

#include <unistd.h>

#include <fcntl.h>

#include <stdlib.h>

#include <string.h>

#define KER\_RW\_R8 100

#define KER\_RW\_R16 101

#define KER\_RW\_R32 102

#define KER\_RW\_W8 103

#define KER\_RW\_W16 104

#define KER\_RW\_W32 105

/\* Usage:

\* ./regeditor r8 addr [num] // 用法

\* ./regeditor r16 addr [num]

\* ./regeditor r32 addr [num]

\*

\* ./regeditor w8 addr val

\* ./regeditor w16 addr val

\* ./regeditor w32 addr val

\*/

void print\_usage(char \*file)

{

printf("Usage:\n");

printf("%s <r8 | r16 | r32> <phy addr> [num]\n", file);

printf("%s <w8 | w16 |w32> <phy addr> <val>\n", file);

}

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

{

int fd;

unsigned int buf[2];

unsigned int i;

unsigned int num;

if ((argc != 3) && (argc != 4))

{

print\_usage(argv[0]);

return -1;

}

fd = open("/dev/ker\_rw", O\_RDWR);

if (fd < 0)

{

printf("can't open /dev/ker\_rw\n");

return -2;

}

/\* addr \*/

buf[0] = strtoul(argv[2], NULL, 0);

if (argc == 4)

{

buf[1] = strtoul(argv[3], NULL, 0);

num = buf[1];

}

else

{

num = 1;

}

if (strcmp(argv[1], "r8") == 0)

{

for ( i = 0; i < num; i++)

{

ioctl(fd, KER\_RW\_R8, buf); /\* val = buf[1] \*/

printf("%02d. [%08x] = %02x\n", i, buf[0], (unsigned char)buf[1]);

buf[0] += 1;

}

}

else if (strcmp(argv[1], "r16") == 0)

{

for ( i = 0; i < num; i++)

{

ioctl(fd, KER\_RW\_R16, buf); /\* val = buf[1] \*/

printf("%02d. [%08x] = %04x\n", i, buf[0], (unsigned short)buf[1]);

buf[0] += 2;

}

}

else if (strcmp(argv[1], "r32") == 0)

{

for ( i = 0; i < num; i++)

{

ioctl(fd, KER\_RW\_R32, buf); /\* val = buf[1] \*/

printf("%02d. [%08x] = %08x\n", i, buf[0], (unsigned int)buf[1]);

buf[0] += 4;

}

}

else if (strcmp(argv[1], "w8") == 0)

{

ioctl(fd, KER\_RW\_W8, buf); /\* val = buf[1] \*/

}

else if (strcmp(argv[1], "w16") == 0)

{

ioctl(fd, KER\_RW\_W16, buf); /\* val = buf[1] \*/

}

else if (strcmp(argv[1], "w32") == 0)

{

ioctl(fd, KER\_RW\_W32, buf); /\* val = buf[1] \*/

}

else

{

printf(argv[0]);

return -1;

}

return 0;

}

运行结果：

[root@Loongson:/loongson\_tools]#./regeditor r32 0x1fd011e4

00. [1fd011e4] = 00c00000

## 9.3静态映射(map\_desc)方式

下面介绍静态映射方式即通过map\_desc结构体静态创建I/O资源映射表。

内核提供了在系统启动时通过map\_desc结构体静态创建I/O资源到内核地址空间的线性映射表(即page table)的方式，这种映射表是一种一一映射的关系。程序员可以自己定义该I/O内存资源映射后的虚拟地址。创建好了静态映射表，在内核或驱动中访问该I/O资源时则无需再进行ioreamp动态映射，可以直接通过映射后的I/O虚拟地址去访问它。

下面详细分析这种机制的原理并举例说明如何通过这种静态映射的方式访问外设I/O内存资源。

内核提供了一个重要的结构体struct machine\_desc ,这个结构体在内核移植中起到相当重要的作用,内核通过machine\_desc结构体来控制系统体系[**架构**](http://lib.csdn.net/base/16)相关部分的初始化。

machine\_desc结构体的成员包含了体系架构相关部分的几个最重要的初始化函数，包括map\_io, init\_irq, init\_machine以及phys\_io , timer成员等。

machine\_desc结构体定义如下：

**struct map\_desc** {//用到的结构体

/\*

\* Note! The first four elements are used

\* by assembler code in head-armv.S

\*/

unsigned long virtual; //虚拟地址

unsigned long pfn; //\_\_phys\_to\_pfn(物理地址) , 就是物理页框号

unsigned long length; //大小

unsigned int type; //类型

};

**常见问题：**

1、用户空间（进程）是否有高端内存概念？

用户进程没有高端内存概念。只有在内核空间才存在高端内存。用户进程最多只可以访问3G物理内存，而内核进程可以访问所有物理内存。

2、64位内核中有高端内存吗？

目前现实中，64位Linux内核不存在高端内存，因为64位内核可以支持超过512GB内存。若机器安装的物理内存超过内核地址空间范围，就会存在高端内存。

3、用户进程能访问多少物理内存？内核代码能访问多少物理内存？

32位系统用户进程最大可以访问3GB，内核代码可以访问所有物理内存。

64位系统用户进程最大可以访问超过512GB，内核代码可以访问所有物理内存。

4、高端内存和物理地址、逻辑地址、线性地址的关系？

高端内存只和逻辑地址有关系，和逻辑地址、物理地址没有直接关系。

5、为什么不把所有的地址空间都分配给内核？

若把所有地址空间都给内存，那么用户进程怎么使用内存？怎么保证内核使用内存和用户进程不起冲突？

（1）让我们忽略Linux对段式内存映射的支持。 在保护模式下，我们知道无论CPU运行于用户态还是核心态，CPU执行程序所访问的地址都是虚拟地址，MMU 必须通过读取控制寄存器CR3中的值作为当前页面目录的指针，进而根据分页内存映射机制（参看相关文档）将该虚拟地址转换为真正的物理地址才能让CPU真 正的访问到物理地址。

（2）对于32位的Linux，其每一个进程都有4G的寻址空间，但当一个进程访问其虚拟内存空间中的某个地址时又是怎样实现不与其它进程的虚拟空间混淆 的呢？每个进程都有其自身的页面目录PGD，Linux将该目录的指针存放在与进程对应的内存结构task\_struct.(struct mm\_struct)mm->pgd中。每当一个进程被调度（schedule()）即将进入运行态时，Linux内核都要用该进程的PGD指针设 置CR3（switch\_mm()）。

（3）当创建一个新的进程时，都要为新进程创建一个新的页面目录PGD，并从内核的页面目录swapper\_pg\_dir中复制内核区间页面目录项至新建进程页面目录PGD的相应位置，具体过程如下：

do\_fork() --> copy\_mm() --> mm\_init() --> pgd\_alloc() --> set\_pgd\_fast() --> get\_pgd\_slow() --> memcpy(&PGD + USER\_PTRS\_PER\_PGD, swapper\_pg\_dir + USER\_PTRS\_PER\_PGD, (PTRS\_PER\_PGD - USER\_PTRS\_PER\_PGD) \* sizeof(pgd\_t))  
这样一来，每个进程的页面目录就分成了两部分，第一部分为“用户空间”，用来映射其整个进程空间（0x0000 0000－0xBFFF FFFF）即3G字节的虚拟地址；第二部分为“系统空间”，用来映射（0xC000 0000－0xFFFF FFFF）1G字节的虚拟地址。可以看出Linux系统中每个进程的页面目录的第二部分是相同的，所以从进程的角度来看，每个进程有4G字节的虚拟空间， 较低的3G字节是自己的用户空间，最高的1G字节则为与所有进程以及内核共享的系统空间。

（4）现在假设我们有如下一个情景：

在进程A中通过系统调用sethostname(const char \*name,seze\_t len)设置计算机在网络中的“主机名”。

在该情景中我们势必涉及到从用户空间向内核空间传递数据的问题，name是用户空间中的地址，它要通过系统调用设置到内核中的某个地址中。让我们看看这个 过程中的一些细节问题：系统调用的具体实现是将系统调用的参数依次存入寄存器ebx,ecx,edx,esi,edi（最多5个参数，该情景有两个 name和len），接着将系统调用号存入寄存器eax，然后通过中断指令“int 80”使进程A进入系统空间。由于进程的CPU运行级别小于等于为系统调用设置的陷阱门的准入级别3，所以可以畅通无阻的进入系统空间去执行为int 80设置的函数指针system\_call()。由于system\_call()属于内核空间，其运行级别DPL为0，CPU要将堆栈切换到内核堆栈，即 进程A的系统空间堆栈。我们知道内核为新建进程创建task\_struct结构时，共分配了两个连续的页面，即8K的大小，并将底部约1k的大小用于 task\_struct（如#define alloc\_task\_struct() ((struct task\_struct \*) \_\_get\_free\_pages(GFP\_KERNEL,1))）,而其余部分内存用于系统空间的堆栈空间，即当从用户空间转入系统空间时，堆栈指针 esp变成了（alloc\_task\_struct()+8192），这也是为什么系统空间通常用宏定义current（参看其实现）获取当前进程的 task\_struct地址的原因。每次在进程从用户空间进入系统空间之初，系统堆栈就已经被依次压入用户堆栈SS、用户堆栈指针ESP、EFLAGS、 用户空间CS、EIP，接着system\_call()将eax压入，再接着调用SAVE\_ALL依次压入ES、DS、EAX、EBP、EDI、ESI、 EDX、ECX、EBX，然后调用sys\_call\_table+4\*%EAX，本情景为sys\_sethostname()。

（5）在sys\_sethostname()中，经过一些保护考虑后，调用copy\_from\_user(to,from,n），其中to指向内核空间 system\_utsname.nodename，譬如0xE625A000，from指向用户空间譬如0x8010FE00。现在进程A进入了内核，在 系统空间中运行，MMU根据其PGD将虚拟地址完成到物理地址的映射，最终完成从用户空间到系统空间数据的复制。准备复制之前内核先要确定用户空间地址和 长度的合法性，至于从该用户空间地址开始的某个长度的整个区间是否已经映射并不去检查，如果区间内某个地址未映射或读写权限等问题出现时，则视为坏地址， 就产生一个页面异常，让页面异常服务程序处理。过程如 下：copy\_from\_user()->generic\_copy\_from\_user()->access\_ok()+\_\_copy\_user\_zeroing().

（6）小结：

\*进程寻址空间0~4G

\*进程在用户态只能访问0~3G，只有进入内核态才能访问3G~4G

\*进程通过系统调用进入内核态

\*每个进程虚拟空间的3G~4G部分是相同的

\*进程从用户态进入内核态不会引起CR3的改变但会引起堆栈的改变

## 9.4 mmap系统调用

### 1）mmap系统调用

mmap将一个文件或者其它对象映射进内存。文件被映射到多个页上，如果文件的大小不是所有页的大小之和，最后一个页不被使用的空间将会清零。munmap执行相反的操作，删除特定地址区域的对象映射。

当使用mmap映射文件到进程后,就可以直接操作这段虚拟地址进行文件的读写等操作,不必再调用read,write等系统调用.但需注意,直接对该段内存写时不会写入超过当前文件大小的内容.

采用共享内存通信的一个显而易见的好处是效率高，因为进程可以直接读写内存，而不需要任何数据的拷贝。对于像管道和消息队列等通信方式，则需要在内核和用户空间进行四次的数据拷贝，而共享内存则只拷贝两次数据：一次从输入文件到共享内存区，另一次从共享内存区到输出文件。实际上，进程之间在共享内存时，并不总是读写少量数据后就解除映射，有新的通信时，再重新建立共享内存区域。而是保持共享区域，直到通信完毕为止，这样，数据内容一直保存在共享内存中，并没有写回文件。共享内存中的内容往往是在解除映射时才写回文件的。因此，采用共享内存的通信方式效率是非常高的。

基于文件的映射，在mmap和munmap执行过程的任何时刻，被映射文件的st\_atime可能被更新。如果st\_atime字段在前述的情况下没有得到更新，首次对映射区的第一个页索引时会更新该字段的值。用PROT\_WRITE 和 MAP\_SHARED标志建立起来的文件映射，其st\_ctime 和 st\_mtime在对映射区写入之后，但在msync()通过MS\_SYNC 和 MS\_ASYNC两个标志调用之前会被更新。

用法：

#include <sys/mman.h>

#include <unistd.h>

void \*mmap(void \*start, size\_t length, int prot, int flags,

int fd, off\_t offset);

int munmap(void \*start, size\_t length);

返回说明：

成功执行时，mmap()返回被映射区的指针，munmap()返回0。失败时，mmap()返回MAP\_FAILED[其值为(void \*)-1]，munmap返回-1。errno被设为以下的某个值

EACCES：访问出错

EAGAIN：文件已被锁定，或者太多的内存已被锁定

EBADF：fd不是有效的文件描述词

EINVAL：一个或者多个参数无效

ENFILE：已达到系统对打开文件的限制

ENODEV：指定文件所在的文件系统不支持内存映射

ENOMEM：内存不足，或者进程已超出最大内存映射数量

EPERM：权能不足，操作不允许

ETXTBSY：已写的方式打开文件，同时指定MAP\_DENYWRITE标志

SIGSEGV：试着向只读区写入

SIGBUS：试着访问不属于进程的内存区

参数：

start：映射区的开始地址。

length：映射区的长度。

prot：期望的内存保护标志，不能与文件的打开模式冲突。是以下的某个值，可以通过or运算合理地组合在一起

PROT\_EXEC //页内容可以被执行

PROT\_READ //页内容可以被读取

PROT\_WRITE //页可以被写入

PROT\_NONE //页不可访问

flags：指定映射对象的类型，映射选项和映射页是否可以共享。它的值可以是一个或者多个以下位的组合体

MAP\_FIXED //使用指定的映射起始地址，如果由start和len参数指定的内存区重叠于现存的映射空间，重叠部分将会被丢弃。如果指定的起始地址不可用，操作将会失败。并且起始地址必须落在页的边界上。

MAP\_SHARED //与其它所有映射这个对象的进程共享映射空间。对共享区的写入，相当于输出到文件。直到msync()或者munmap()被调用，文件实际上不会被更新。

MAP\_PRIVATE //建立一个写入时拷贝的私有映射。内存区域的写入不会影响到原文件。这个标志和以上标志是互斥的，只能使用其中一个。

MAP\_DENYWRITE //这个标志被忽略。

MAP\_EXECUTABLE //同上

MAP\_NORESERVE //不要为这个映射保留交换空间。当交换空间被保留，对映射区修改的可能会得到保证。当交换空间不被保留，同时内存不足，对映射区的修改会引起段违例信号。

MAP\_LOCKED //锁定映射区的页面，从而防止页面被交换出内存。

MAP\_GROWSDOWN //用于堆栈，告诉内核VM系统，映射区可以向下扩展。

MAP\_ANONYMOUS //匿名映射，映射区不与任何文件关联。

MAP\_ANON //MAP\_ANONYMOUS的别称，不再被使用。

MAP\_FILE //兼容标志，被忽略。

MAP\_32BIT //将映射区放在进程地址空间的低2GB，MAP\_FIXED指定时会被忽略。当前这个标志只在x86-64平台上得到支持。

MAP\_POPULATE //为文件映射通过预读的方式准备好页表。随后对映射区的访问不会被页违例阻塞。

MAP\_NONBLOCK //仅和MAP\_POPULATE一起使用时才有意义。不执行预读，只为已存在于内存中的页面建立页表入口。

fd：有效的文件描述词。如果MAP\_ANONYMOUS被设定，为了兼容问题，其值应为-1。

offset：被映射对象内容的起点。

2. 系统调用munmap()

#include <unistd.h>

#include <sys/mman.h>

int munmap( void \* addr, size\_t len )

该调用在进程地址空间中解除一个映射关系，addr是调用mmap()时返回的地址，len是映射区的大小。当映射关系解除后，对原来映射地址的访问将导致段错误发生。

3. 系统调用msync()

#include <sys/mman.h>

#include <unistd.h>

int msync ( void \* addr , size\_t len, int flags)

一般说来，进程在映射空间的对共享内容的改变并不直接写回到磁盘文件中，往往在调用munmap（）后才执行该操作。可以通过调用msync()实现磁盘上文件内容与共享内存区的内容一致。

### 2）系统调用mmap()用于共享内存的两种方式

（1）使用普通文件提供的内存映射：适用于任何进程之间；此时，需要打开或创建一个文件，然后再调用mmap()；典型调用代码如下：

fd=open(name, flag, mode);

if(fd<0)

...

ptr=mmap(NULL, len , PROT\_READ|PROT\_WRITE, MAP\_SHARED , fd , 0);

通过mmap()实现共享内存的通信方式有许多特点和要注意的地方

（2）使用特殊文件提供匿名内存映射：适用于具有亲缘关系的进程之间；由于父子进程特殊的亲缘关系，在父进程中先调用mmap()，然后调用fork()。那么在调用fork()之后，子进程继承父进程匿名映射后的地址空间，同样也继承mmap()返回的地址，这样，父子进程就可以通过映射区域进行通信了。注意，这里不是一般的继承关系。一般来说，子进程单独维护从父进程继承下来的一些变量。而mmap()返回的地址，却由父子进程共同维护。

对于具有亲缘关系的进程实现共享内存最好的方式应该是采用匿名内存映射的方式。此时，不必指定具体的文件，只要设置相应的标志即可.

### 3） mmap进行内存映射的原理

mmap系统调用的最终目的是将,设备或文件映射到用户进程的虚拟地址空间,实现用户进程对文件的直接读写,这个任务可以分为以下三步:

（1）在用户虚拟地址空间中寻找空闲的满足要求的一段连续的虚拟地址空间,为映射做准备(由内核mmap系统调用完成)

每个进程拥有3G字节的用户虚存空间。但是，这并不意味着用户进程在这3G的范围内可以任意使用，因为虚存空间最终得映射到某个物理存储空间（内存或磁盘空间），才真正可以使用。

那么，内核怎样管理每个进程3G的虚存空间呢？概括地说，用户进程经过编译、链接后形成的映象文件有一个代码段和数据段（包括data段和bss段），其中代码段在下，数据段在上。数据段中包括了所有静态分配的数据空间，即全局变量和所有申明为static的局部变量，这些空间是进程所必需的基本要求，这些空间是在建立一个进程的运行映像时就分配好的。除此之外，堆栈使用的空间也属于基本要求，所以也是在建立进程时就分配好的，如图3.1所示：

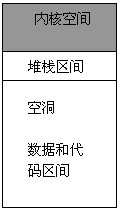


图3.1 进程虚拟空间的划分

在内核中,这样每个区域用一个结构struct vm\_area\_struct 来表示.它描述的是一段连续的、具有相同访问属性的虚存空间，该虚存空间的大小为物理内存页面的整数倍。可以使用 cat /proc//maps来查看一个进程的内存使用情况,pid是进程号.其中显示的每一行对应进程的一个vm\_area\_struct结构.

下面是struct vm\_area\_struct结构体的定义：

#include <linux/mm\_types.h>

/\* This struct defines a memory VMM memory area. \*/

struct vm\_area\_struct {

struct mm\_struct \* vm\_mm; /\* VM area parameters \*/

unsigned long vm\_start;

unsigned long vm\_end;

/\* linked list of VM areas per task, sorted by address \*/

struct vm\_area\_struct \*vm\_next;

pgprot\_t vm\_page\_prot;

unsigned long vm\_flags;

/\* AVL tree of VM areas per task, sorted by address \*/

short vm\_avl\_height;

struct vm\_area\_struct \* vm\_avl\_left;

struct vm\_area\_struct \* vm\_avl\_right;

/\* For areas with an address space and backing store,

vm\_area\_struct \*vm\_next\_share;

struct vm\_area\_struct \*\*vm\_pprev\_share;

struct vm\_operations\_struct \* vm\_ops;

unsigned long vm\_pgoff; /\* offset in PAGE\_SIZE units, \*not\* PAGE\_CACHE\_SIZE \*/

struct file \* vm\_file;

unsigned long vm\_raend;

void \* vm\_private\_data; /\* was vm\_pte (shared mem) \*/

};

通常，进程所使用到的虚存空间不连续，且各部分虚存空间的访问属性也可能不同。所以一个进程的虚存空间需要多个vm\_area\_struct结构来描述。在vm\_area\_struct结构的数目较少的时候，各个vm\_area\_struct按照升序排序，以单链表的形式组织数据（通过vm\_next指针指向下一个vm\_area\_struct结构）。但是当vm\_area\_struct结构的数据较多的时候，仍然采用链表组织的化，势必会影响到它的搜索速度。针对这个问题，vm\_area\_struct还添加了vm\_avl\_hight（树高）、vm\_avl\_left（左子节点）、vm\_avl\_right（右子节点）三个成员来实现AVL树，以提高vm\_area\_struct的搜索速度。

假如该vm\_area\_struct描述的是一个文件映射的虚存空间，成员vm\_file便指向被映射的文件的file结构，vm\_pgoff是该虚存空间起始地址在vm\_file文件里面的文件偏移，单位为物理页面。

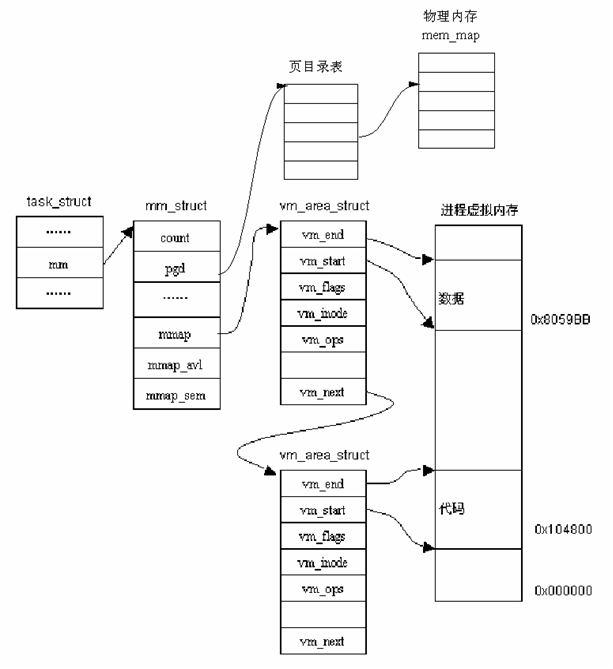


图3.2 进程虚拟地址示意图

因此,mmap系统调用所完成的工作就是准备这样一段虚存空间,并建立vm\_area\_struct结构体,将其传给具体的设备驱动程序.

（2）建立虚拟地址空间和文件或设备的物理地址之间的映射(设备驱动完成)

建立文件映射的第二步就是建立虚拟地址和具体的物理地址之间的映射,这是通过修改进程页表来实现的.mmap方法是file\_opeartions结构的成员:

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

linux有2个方法建立页表:

(1) 使用remap\_pfn\_range一次建立所有页表.

int remap\_pfn\_range(struct vm\_area\_struct \*vma, unsigned long virt\_addr, unsigned long pfn, unsigned long size, pgprot\_t prot);

返回值:

成功返回 0, 失败返回一个负的错误值

参数说明:

vma 用户进程创建一个vma区域

virt\_addr 重新映射应当开始的用户虚拟地址. 这个函数建立页表为这个虚拟地址范围从 virt\_addr 到 virt\_addr\_size.

pfn 页帧号, 对应虚拟地址应当被映射的物理地址. 这个页帧号简单地是物理地址右移 PAGE\_SHIFT 位. 对大部分使用, VMA 结构的 vm\_paoff 成员正好包含你需要的值. 这个函数影响物理地址从 (pfn<

size 正在被重新映射的区的大小, 以字节.

prot 给新 VMA 要求的"protection". 驱动可(并且应当)使用在vma->vm\_page\_prot 中找到的值.

(2) 使用nopage VMA方法每次建立一个页表项.

struct page \*(\*nopage)(struct vm\_area\_struct \*vma, unsigned long address, int \*type);

返回值:

成功则返回一个有效映射页,失败返回NULL.

参数说明:

address 代表从用户空间传过来的用户空间虚拟地址.

返回一个有效映射页.

(3) 使用方面的限制：

remap\_pfn\_range不能映射常规内存，只存取保留页和在物理内存顶之上的物理地址。因为保留页和在物理内存顶之上的物理地址内存管理系统的各个子模块管理不到。640 KB 和 1MB 是保留页可能映射，设备I/O内存也可以映射。如果想把kmalloc()申请的内存映射到用户空间，则可以通过mem\_map\_reserve()把相应的内存设置为保留后就可以。

（3）当实际访问新映射的页面时的操作(由缺页中断完成)

(1) page cache及swap cache中页面的区分：一个被访问文件的物理页面都驻留在page cache或swap cache中，一个页面的所有信息由struct page来描述。struct page中有一个域为指针mapping ，它指向一个struct address\_space类型结构。page cache或swap cache中的所有页面就是根据address\_space结构以及一个偏移量来区分的。

(2) 文件与 address\_space结构的对应：一个具体的文件在打开后，内核会在内存中为之建立一个struct inode结构，其中的i\_mapping域指向一个address\_space结构。这样，一个文件就对应一个address\_space结构，一个 address\_space与一个偏移量能够确定一个page cache 或swap cache中的一个页面。因此，当要寻址某个数据时，很容易根据给定的文件及数据在文件内的偏移量而找到相应的页面。

(3) 进程调用mmap()时，只是在进程空间内新增了一块相应大小的缓冲区，并设置了相应的访问标识，但并没有建立进程空间到物理页面的映射。因此，第一次访问该空间时，会引发一个缺页异常。

(4) 对于共享内存映射情况，缺页异常处理程序首先在swap cache中寻找目标页（符合address\_space以及偏移量的物理页），如果找到，则直接返回地址；如果没有找到，则判断该页是否在交换区 (swap area)，如果在，则执行一个换入操作；如果上述两种情况都不满足，处理程序将分配新的物理页面，并把它插入到page cache中。进程最终将更新进程页表。

注：对于映射普通文件情况（非共享映射），缺页异常处理程序首先会在page cache中根据address\_space以及数据偏移量寻找相应的页面。如果没有找到，则说明文件数据还没有读入内存，处理程序会从磁盘读入相应的页面，并返回相应地址，同时，进程页表也会更新.

(5) 所有进程在映射同一个共享内存区域时，情况都一样，在建立线性地址与物理地址之间的映射之后，不论进程各自的返回地址如何，实际访问的必然是同一个共享内存区域对应的物理页面。

### 4）内存映射的步骤:

用open系统调用打开文件, 并返回描述符fd.

用mmap建立内存映射, 并返回映射首地址指针start.

对映射(文件)进行各种操作, 显示(printf), 修改(sprintf).

用munmap(void \*start, size\_t lenght)关闭内存映射.

用close系统调用关闭文件fd.

## 9.5 mmap编程示例

修改自刘世伟的<https://github.com/lshw/loongson_tools>.

读取GPIO的复用寄存器：cbus\_dump.c

#include<stdio.h>

#include<unistd.h>

#include<sys/mman.h>

#include<sys/types.h>

#include<sys/stat.h>

#include<fcntl.h>

#include "ls1c\_regs.h"

int main()

{

unsigned char \* map\_base;

FILE \*f;

int n, fd;

fd = open("/dev/mem", O\_RDWR|O\_SYNC);

if (fd == -1) {

return (-1);

}

/\* 把bfd01000开始0x1000字节，映射到map\_base \*/

map\_base = mmap(NULL, 0x1000, PROT\_READ|PROT\_WRITE, MAP\_SHARED, fd, 0x1fd01000);

if (map\_base == 0) {

printf("NULL pointer!\n");

}

unsigned long addr;

unsigned char content;

int i = 0,i1 = 0;

printf("[gpio\_number]: 0 8 16 24 32 40 48 56 64 72 80 88 96 104 112 120");

for(i1 = 0 ; i1 < 5 ; i1 ++) { //每个gpio5个功能的定义分别是bfd0110c0,bfd0110d0,bfd0110e0,bfd0110f0,bfd011200开始的16个字节，

printf("\nfun%d[1fd01%03x]:",i1 + 1,LS1X\_CBUS\_FIRST0 + i1 \* 0x10);

for(i = 0 ; i < 0x10 ; i ++) {

printf(" %02X",\*(volatile unsigned long \*)(map\_base+ LS1X\_CBUS\_FIRST0 + i1 \* 0x10 + i)&0xFF);

}

}

close(fd);

printf("\n");

munmap(map\_base, 0x1000);

return (0);

}

读取GPIO的复用配置：gpio\_func.c

清除寄存器中的某一位：regs\_bit\_clr.c

读取寄存器中的某一位：regs\_bit\_get.c

设置寄存器中的某一位：regs\_bit\_set.c

读取某一个寄存器：regs\_read.c

写值某一个寄存器：regs\_write.c