利用 mmap /dev/mem 读写 Linux 内存

使用 hexedit /dev/mem 可以显示所有物理内存中的信息。 运用 mmap 将 /dev/mem map 出来，然后直接对其读写可以实现用户空间的内核操作。

以下是我写的一个 sample

```
#include<stdio.h>
#include<unistd.h>
#include<sys/mman.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.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);
   }

   map_base = mmap(NULL, 0xff, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0x20000);

   if (map_base == 0)
   {
      printf("NULL pointer!\n");
   }
   else
   {
      printf("Successfull!\n");
   }

   unsigned long addr;
   unsigned char content;

   int i = 0;
   for (;i < 0xff; ++i)
   {
      addr = (unsigned long)(map_base + i);
      content = map_base[i];
```

```
        printf("address: 0x%lx   content 0x%x\t\t", addr, (unsigned int)content);

        map_base[i] = (unsigned char)i;
        content = map_base[i];
        printf("updated address: 0x%lx   content 0x%x\n", addr, (unsigned int)content);
    }

    close(fd);

    munmap(map_base, 0xff);

    return (1);
}
```

上面的例子将起始地址 0x20000（物理地址）， 长度为 0xff 映射出来。 然后就可以像普通数组一样操作内存。

下面是输出结果

address: 0x7f3f95391000 content 0x0 updated address: 0x7f3f95391000 content 0x0
address: 0x7f3f95391001 content 0x0 updated address: 0x7f3f95391001 content 0x1
address: 0x7f3f95391002 content 0x0 updated address: 0x7f3f95391002 content 0x2
address: 0x7f3f95391003 content 0x0 updated address: 0x7f3f95391003 content 0x3
address: 0x7f3f95391004 content 0x0 updated address: 0x7f3f95391004 content 0x4
。。。

我的测试机器是 64 位机。 该例子将物理地址 0x20000 映射到了虚拟地址 0x7f3f95391000。

首先将当前地址下的内容输出， 然后写入新值。

可以通过 hexedit /dev/mem 验证新值已经写入。

如果想在用户态处理 kernel 分配的地址可以这么做。 首先用 virt_addr = get_free_pages(GFP_KERNEL, order)分配内存，通过 phy_addr = __pa(virt_addr)得到物理地址，然后在用户态将/dev/mem 用 mmap 映射出来， offset 就是 phy_addr, length 设为 2^order。 此时就可以在用户态读写内核分配的内存了。

注：该操作需要有 root 权限。

转载地址：http://blog.csdn.net/zhanglei4214/article/details/6653568

==========

/dev/mem 详解

/dev/mem 是物理内存的全映像，可以用来访问物理内存，一般用法是 open("/dev/mem",O_RDWR|O_SYNC)，然后 mmap，接着就可以用 mmap 的地址来访问物理内存，这实际上就是实现用户空间驱动的一种方法。

有几个论据倾向于用户空间编程，有时编写一个所谓的用户空间设备驱动对比钻研内核是一个明智的选择，用户空间驱动的好处在于：

完整的 C 库可以链接，驱动可以进行许多奇怪的任务，而不用依靠外面的程序（实现使用策略的工具程序，常常随着驱动自身发布）。

程序员可以在驱动代码上运行常用的调试器，而不必调试一个运行中的内核的弯路。

如果一个用户空间驱动挂起了，你可以简单地杀死它。用户空间驱动出现问题不可能挂起整个系统，除非被控制的硬件真的疯掉了。

用户内存是可交换的，不像内核内存。这样一个不常使用却有很大一个驱动的设备不会占据别的程序可以用到的 RAM，除了在它实际在用时。

一个精心设计的 驱动程序仍然可以如同内核空间驱动一样允许对设备的并行存取。

如果你必须编写一个封闭源码的驱动，用户空间的选项使你容易辨明不明朗的许可的情况和改变的内核接口带来的问题。

完整的 C 库可以链接，驱动可以进行许多奇怪的任务，而不用依靠外面的程序（实现使用策略的工具程序，常常随着驱动自身发布）。

程序员可以在驱动代码上运行常用的调试器，而不必调试一个运行中的内核的弯路。

如果一个用户空间驱动挂起了，你可以简单地杀死它。用户空间驱动出现问题不可能挂起整个系统，除非被控制的硬件真的疯掉了。

用户内存是可交换的，不像内核内存。这样一个不常使用却有很大一个驱动的设备不会占据别的程序可以用到的 RAM，除了在它实际在用时。

一个精心设计的 驱动程序仍然可以如同内核空间驱动一样允许对设备的并行存取。

如果你必须编写一个封闭源码的驱动，用户空间的选项使你容易辨明不明朗的许可的情况和改变的内核接口带来的问题。

但是，用户空间的设备驱动有几个缺点，最重要的是：

中断在用户空间无法使用，在某些平台上有对这个限制的解决方法，例如在 IA32 体系结构上的 vm86 系统调用。

只可能通过内存映射/dev/mem 来使用 DMA，而且只有特权用户可以这样做。

存取 I/O 端口只能在调用 ioperm 或者 iopl 只有，此外，不是所有的平台都支持这些系统调用，而存取/dev/port 可能太慢而无效率，这些系统调用和设备文件都要求特权用户。

响应时间慢，因为需要上下文切换在用户和硬件之间传递消息和动作。

更坏的是，如果驱动已经被交换到硬盘，响应时间会长到不可接受，使用 mlock 系统调用可能会有帮助，但是你需要经常锁住许多内存页，因为一个用户空间程序依赖大量的库代码，mlock 也限制在授权用户上。

最重要的设备不能在用户空间处理，包括网络接口和块设备。

中断在用户空间无法使用，在某些平台上有对这个限制的解决方法，例如在 IA32 体系结构上的 vm86 系统调用。

只可能通过内存映射/dev/mem 来使用 DMA，而且只有特权用户可以这样做。

存取 I/O 端口只能在调用 ioperm 或者 iopl 只有，此外，不是所有的平台都支持这些系统调用，而存取/dev/port 可能太慢而无效率，这些系统调用和设备文件都要求特权用户。

响应时间慢，因为需要上下文切换在用户和硬件之间传递消息和动作。

更坏的是，如果驱动已经被交换到硬盘，响应时间会长到不可接受，使用 mlock 系统调用可能会有帮助，但是你需要经常锁住许多内存页，因为一个用户空间程序依赖大量的库代码，mlock 也限制在授权用户上。

最重要的设备不能在用户空间处理，包括网络接口和块设备。

综上，用户空间驱动不能做的事情毕竟太多，感兴趣的应用程序还是存在：对

SCSI 扫描器设备的支持(由 SANE 包实现)和 CD 刻录机(由 cdrecord 和别的工具实现)。在两种情况下，用户级别的设备情况依赖"SCSI generic"内核驱动，它输出了底层的 SCSI 功能给用户程序，因此它们可以驱动它们自己的硬件。

当你开始处理新的没有用过的硬件时，通过开发用户空间驱动，你可以学习去管理你的硬件，不必担心挂起整个系统，一旦你完成了，在一个内核模块中封装软件就会是一个简单的操作了。

Notes: 新内核已经限制/dev/mem 中 1M 以上的内存访问，这是一个可配置的选项 CONFIG_STRICT_DEVMEM，在我的机器上已经选择 n 了，可是好像还是只能映射 1M 一下物理内存。

在 ULCC 中通过 mmap 映射物理内存地址到虚拟地址，实现物理页的 noncacheable，这主要是通过映射/dev/mem 修改 PAT 中的页面属性达到的。X86 的页面属性表(PAT)能够在页面粒度上设置内存属性，PAT 是对 MTRR 的补充，通过 MTRR 可以为物理地址区域设置内存类型，但是 PAT 比 MTRR 更灵活，因为它可以在页面级别设置属性，而且硬件上也不限制属性设置的数量。PAT 相当灵活，即使多个虚拟地址映射到同一个物理内存地址，也不会引起内存类型的冲突，通过 PAT 能够设置多种类型的内存属性，其中最常用的有 4 种：Write-Back、Ucached、Write-Combined、Uncached-Minux。

```
fm = open("/dev/mem", O_RDWR|O_SYNC);
mmap((void *)addr, PAGE_SIZE, PROT_READ|PROT_WRITE, MAP_SHARED|MAP_FIXED, fm, pfn<
<PAGE_SHIFT);
fm = open("/dev/mem", O_RDWR|O_SYNC);
mmap((void *)addr, PAGE_SIZE, PROT_READ|PROT_WRITE, MAP_SHARED|MAP_FIXED, fm, pfn<
<PAGE_SHIFT);
```

/dev/kmem：内核看到的虚拟内存的全映像，可以用来访问 kernel 的内容。

What Is /proc/kcore?
None of the files in /proc are really there--they're all, "pretend," files made up by the kernel, to give you information about the system and don't take up any hard disk space.

/proc/kcore is like an "alias" for the memory in your computer. Its size is the same as the amount of RAM you have, and if you read it as a file, the kernel does memory reads.

转载地址：http://blog.csdn.net/su_linux/article/details/8737690