观察 Linux 内核

BG6RDF

随着 Linux 系统的发展，越来越多的应用系统被移植到 Linux 系统中来。对 Linux 系统运行状态进行监控并进行优化的需求日益突出，为此我们需要深入到内核中去了解 Linux 的运转情况。下面我们就从观察 i386 平台的 Linux 内核开始。

一．问题

既然要观察 Linux 内核，就必须能够访问 Linux 内核空间，但是我们知道一般的用户进程是无法访问内核空间的，操作系统设置了重重限制，那么如何绕过这些限制是我们首先需要解决的问题。下面我们来看看 Linux 中用户进程和内核进程的主要差异：

1.他们的特权级别不同，内核进程运行在 ring 0，用户进程运行在 ring 3。

2.用户进程逻辑地址空间在 0x00000000 到 0xbfffffff，内核的地址空间在 0xc0000000 到 0xffffffff。内核在初始化页表时，将内核空间的页表设置为仅允许 ring 0 读写。用户进程如果试图访问内核空间，会产生异常中断。

这样用户进程似乎无法完成观察 Linux 内核的任务，只能通过修改内核的方式了。本文下面将介绍使用模块(Module)和/dev/kmem 文件的方法绕过这个问题。

既然是观察内核，我们需要解决的第二问题是内核数据存储在内存中的哪个位置。实际上内核在装载后，会在内存中生成一张符号表，该表记录了很多重要的数据结构和函数的逻辑地址。同时内核在编译时会生成一个 System.map 文件，该文件描述了内核中所有变量和函数的逻辑地址。但随着 Linux 内核版本的变化，System.map 中的这些地址也会改变，有的系统甚至可能没有 System.map 这个文件，因此最好能通过内存中的符号表取得这些地址。遗憾的是，这个符号表没有包含所有数据结构和函数的地址，而且只有采用模块的方式才能方便地使用符号表。

二．使用模块

Linux 的模块运行在内核地址空间，很自然地绕过了存储保护，而且通过模块可以了解内核中各个寄存器的状态，并执行各种特权指令，同时模块的编写、安装和卸载都很方便。本文给出的例子模块的主要功能是输出内核寄存器的内容，报告内存区和 slab 分配器的使用情况。

1．确定地址

上面提到了在模块中可使用内核符号表定位内核地址，而且这些定位是由模块的装载程序自动完成的，我们要做的只是引用相应的头文件就可以了，如例子中的 init_mm。我们可以在/proc/ksyms 中查看符号表中所包含的所有符号。对于那些符号表中没有指明地址的变量我们只能查 System.map 文件了。如果大家在自己的系统上运行例子模块，可能需要根据 System.map 修改一些地址，如 cache_chain_sem 等。

2．说明

模块的安装和卸载都必须用 root 的身份。

因为 C 语言标准库运行在用户进程空间，所以模块中不能引用 C 语言标准库。例子中的__attribute__((packed))是 gcc 对标准 c 的扩展，使用该命令可使编译器不进行内存对齐。大家可以用 sizeof()在程序中试一下该命令的作用。

例子中的数据结构 kmem_cache_s 并不包含于 Linux 内核头文件中，我们是从源码中将他摘出来的，在 Linux 内核版本发生变化后，这些数据结构也可能发生变化。

如果 PAE 打开了说明 Linux 使用了三级页表，否则只使用两级。Linux 中 mem_map 是一个管理所有页的 struct pages 类型的数组，可通过该数组了解每页内存的使用情况。

Linux 对内存页的管理分为两个层次：节点(node)和区(zone)。I386 平台上只有一个节点，

但有三个区：ZONE_DMA，ZONE_NORMAL 和 ZONE_HIMEM。例子中的 GetNodeInfo 用以查看内核内存的这些数据。在 Linux 内核中分配和释放内存有两种机制 Buddy(伙伴)系统和 Slab 系统。前者用于按页分配，后者在 Buddy 系统基础上通过实现数据对象的缓冲池，用于频繁分配和释放的小对象。本例中列出了 Slab 系统的使用情况。

该模块实际上在初始化后就退出了，因此不必卸载，那么 cleanup_module 函数也不会运行。加载该模块的方式是 insmod modtest.o。模块的输出无法在 X Window 的虚拟终端中看到，必须在控制台方式才能直接看到输出，另外输出也可用 dmesg 命令查看。

```
/*********************************************************
 * Module test program --- Weiwei Zheng 2002/10/04
 * Display kernel information using module.
 * Make: gcc -c -O2 -I/usr/src/linux-2.4/include modtest.c
 * *******************************************************/

#define MODULE
#define __KERNEL__

#include <linux/config.h>
#ifdef MODULE
#include <linux/module.h>
#include <linux/version.h>
#else
#define MOD_INC_USE_COUNT
#define MOD_DEC_USE_COUNT
#endif

#include <linux/types.h>
#include <linux/fs.h>
#include <linux/mm.h>
#include <linux/slab.h>
#include <linux/errno.h>
#include <asm/pgtable.h>
#include <asm/segment.h>
#include <sys/syscall.h>

#include <unistd.h>
#include <linux/unistd.h>

MODULE_LICENSE("GPL");

#define CACHE_NAMELEN   20      /* max name length for a slab cache */
typedef struct kmem_cache_s {
    struct list_head        slabs_full;
    struct list_head        slabs_partial;
```

```
    struct list_head        slabs_free;
    unsigned int            objsize;
    unsigned int            flags;  /* constant flags */
    unsigned int            num;    /* # of objs per slab */
    spinlock_t              spinlock;
#ifdef CONFIG_SMP
    unsigned int            batchcount;
#endif
    unsigned int            gfporder;
    unsigned int            gfpflags;
    size_t                  colour;         /* cache colouring range */
    unsigned int            colour_off;     /* colour offset */
    unsigned int            colour_next;    /* cache colouring */
    kmem_cache_t            *slabp_cache;
        unsigned int            growing;
    unsigned int            dflags;         /* dynamic flags */
    void (*ctor)(void *, kmem_cache_t *, unsigned long);
    void (*dtor)(void *, kmem_cache_t *, unsigned long);
    unsigned long           failures;
    char                    name[CACHE_NAMELEN];
    struct list_head        next;
#ifdef CONFIG_SMP
    cpucache_t              *cpudata[NR_CPUS];
#endif
#if STATS
    unsigned long           num_active;
    unsigned long           num_allocations;
    unsigned long           high_mark;
    unsigned long           grown;
        unsigned long           reaped;
    unsigned long           errors;
#ifdef CONFIG_SMP
    atomic_t                allochit;
    atomic_t                allocmiss;
    atomic_t                freehit;
    atomic_t                freemiss;
#endif
#endif
};

typedef struct {
    short limit;
    unsigned base;
} __attribute__((packed)) gdtr;
```

```
extern long gdt;
long ldt, cr3, cr4, gdt_limit, gdt_base;
long cs, ds, es, fs, gs, ss, tr;
pgd_t * page_dir;
gdtr gdt_buf;

long CheckPAE()
{
    return (cr4 & X86_CR4_PAE);
}

long CheckPSE()
{
    return (cr4 & X86_CR4_PSE);
}

void GetInfo()
{
    __asm__("xor %%eax, %%eax\n"
        "mov %%cs, %%ax\n"
         :"=a"(cs));
    __asm__("xor %%eax, %%eax\n"
        "mov %%ds, %%ax\n"
         :"=a"(ds));
    __asm__("xor %%eax, %%eax\n"
        "mov %%es, %%ax\n"
         :"=a"(es));
    __asm__("xor %%eax, %%eax\n"
        "mov %%fs, %%ax"
         :"=a"(fs));
    __asm__("xor %%eax, %%eax\n"
        "mov %%gs, %%ax\n"
         :"=a"(gs));
    __asm__("xor %%eax, %%eax\n"
        "mov %%ss, %%ax"
         :"=a"(ss));
    __asm__("xor %%eax, %%eax\n"
        "sldt %%ax\n"
         :"=a"(ldt));
    __asm__("mov %%cr3, %%eax":"=a"(cr3));
    __asm__("mov %%cr4, %%eax":"=a"(cr4));
    __asm__("sgdt %0\n":"=m"(gdt_buf));
    gdt_base=gdt_buf.base;
```

```
    gdt_limit=gdt_buf.limit;
    __asm__("xor %%eax, %%eax\n"
        "str %%ax\n"
        :"=a"(tr));
    printk("cs:%04x  ", cs);
    printk("ds:%04x  ", ds);
    printk("es:%04x  ", es);
    printk("fs:%04x  ", fs);
    printk("gs:%04x  ", gs);
    printk("ss:%04x\n", ss);
    printk("ldt:%04x  ", ldt);
    printk("cr3:%08x  ", cr3);
    printk("cr4:%08x\n", cr4);
    if (CheckPAE()) printk ("PAE enabled.\n");
    if (CheckPSE()) printk ("PSE enabled.\n");
    printk("gdt_base:%08x  ", gdt_base);
    printk("gdt_limit:%08x  ", gdt_limit);
    printk("gdt:%08x  ", gdt);
    printk("tr:%08x\n", tr);
    printk("init_mm:%08x ", &init_mm);
    printk("mem_map:%08x\n", mem_map);
    page_dir = pgd_offset(&init_mm, 0);
    printk("PGD linear address:%08x\n", page_dir);
    return;
}

int GetNodeInfo(void)
{
    int i;
    kmem_cache_t * cache_ptr;
    struct semaphore * cache_chain_sem=(struct semaphore *)0xc03907ec;

    /* The address of node can be found in file System.map. */
    pg_data_t * node=(pg_data_t *)0xc0390814; /* 'pgdat_list' in map */
    kmem_cache_t * cache_cache=(kmem_cache_t *)0xc0305220;

    printk("node:%08x node_zones:%08x\n", node, node->node_zones);
    printk("node_size:%016x\n", node->node_size);
    printk("nr_zones:%08x\n", node->nr_zones);
    printk("node_next:%08x\n", node->node_next);
    printk("node_map_map:%08x\n", node->node_mem_map);
    printk("-------------------Zone Info-------------------\n");
    printk("zone_table:%08x\n", zone_table);
    for (i=0; i<MAX_NR_ZONES; i++) {
```

```
    printk("zone[%d %08x] free_pages:%08lx zone_mem_map:%08x\n",
        i, zone_table[i],
        zone_table[i]->free_pages,
            zone_table[i]->zone_mem_map);
    printk("                      zone_start_paddr:%08lx  zone_start_mapnr:%08lx
size:%08lx\n",
        zone_table[i]->zone_start_paddr,
        zone_table[i]->zone_start_mapnr,
        zone_table[i]->size);
    }

    printk("-----------------Slab Info----------------------\n");
    down(cache_chain_sem);
    cache_ptr=cache_cache;
    do {
    printk("name:%s\n", cache_ptr->name);
    cache_ptr=list_entry(cache_ptr->next.next, kmem_cache_t, next);
    }while(cache_ptr!= cache_cache);
    up(cache_chain_sem);
}

int init_module(void)
{
    struct task_struct * task;
    long addr;
    GetInfo();
    GetNodeInfo();
    printk("modtest will not be loaded.\n");
    return -1;
}

void cleanup_module(void)
{
    printk("modtest unloaded.\n");
}
```

三. /dev/kmem 文件

上面我们看到了用模块实现了对内核的观测，但是有些系统被编译成不支持内核模块，在这种系统中就无法使用上述方法，而且如果内核模块编写稍有错误很容易导致系统崩溃。下面我们介绍的通过/dev/kmem 文件访问内核数据的方法就可以解决这些问题。

/dev/kmem 文件是一个特殊的字符设备文件，该文件映射了内核逻辑地址空间。还有一个特殊的设备文件/dev/mem，映射的是物理地址空间。

为保证系统安全，这两个文件只有 root 用户才能读写。如同本例中 rkm 函数所示，我们可在用户程序中使用标准的 C 函数 lseek、read 和 write 实现对内存访问。

本例用于显示 Linux 内核中断描述符表，其中描述符表的地址来自 System.map 文件。有关中断描述符表的说明请见有关参考资料。本例中仅通过 kmem 文件对内存进行读取，实际上还可进行写入操作，甚至完成显存的读写，这可是个不错的功能。

当然使用/dev/kmem 文件的方式也有不足：一是不运行在内核空间，无法查看内核寄存器的状态，无法执行特权指令，无法调用内核函数；二是不便于定位内核地址；三是对内核中某些动态数据的访问需要通过信号灯或锁进行同步，而用户进程很难做到这点。

```
/*********************************************************
 * kmem test program --- Weiwei Zheng 2002/10/04
 * Display Interrupt Descriptor Table by reading kmem.
 * See kmem/mem man pages for more information.
 * Make: gcc -o kmemtest kmemtest.c
 * *******************************************************/
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#define mem_dev "/dev/kmem"
/* The following address of idt_table  */
/* should be get from file System.map. */
#define idt_table 0xc037d000

int rkm(int fd, int offset, void * buf, int size)
{
    if (lseek(fd, offset, SEEK_SET)!=offset) return 0;
    if (read(fd, buf, size)!=size) return 0;
    return size;
}

main()
{
    int i, fd;
    unsigned char idt[256][8];
    unsigned short seg;
    unsigned offset;
    unsigned char present;
    unsigned char dpl;
    unsigned char type;

    fd=open("/dev/kmem", O_RDONLY);
    if (fd==-1) {
        printf("Cannot open %s.\n",mem_dev);
        return -1;
```

```
    }

    rkm(fd, idt_table, idt, sizeof(idt));
    printf("idt\tseg\toffset\t\tpresent\tdpl\ttype\n");
    for (i=0; i<256; i++) {
        seg=*(unsigned short *)&idt[i][2];
        offset=*(unsigned short *)&idt[i][6];
            offset=(offset<<16)+*(unsigned short *)idt[i];
        present=(idt[i][5] & 0x80)>>7;
        dpl=(idt[i][5] & 0x60)>>5;
        type=(idt[i][5] & 0x07);
        printf("0x%x\t0x%x\t0x%x\t0x%x\t0x%x\t0x%x\n",
            i, seg, offset, present, dpl, type);
    }
    close(fd);
}
```

四. 地址转换

在编写查看内核程序的过程中，经常要在逻辑地址和物理地址间转换。下面我们就来介绍一下转换的方法。

1.内核空间

内核空间中物理地址和逻辑地址的关系是：物理地址+0xc0000000=逻辑地址。有了这个公式，转换就非常方便了。

2.用户空间

因为每个用户进程一般都有自己的页表，用户地址空间的转换就复杂多了。如果需要将用户进程的逻辑地址转换到物理地址可参考下面的函数 virt2phys。这个函数的实质是根据进程描述符 task 找到该进程的页全局目录(PGD)，然后找页中间目录(PMD)，最后找到页表(PTE)后，就能确定物理地址了。在 i386CPU 的 PAE 打开时，Linux 使用三级页表，因此在使用此函数时，需要定义 CONFIG_X86_PAE，这样才能正确的引用头文件。

```
unsigned long virt2phys (struct task_struct *task, unsigned long virt_addr)
{
    pgd_t *pgd;              /* page directory */
    pgd_t *pgd_entry;        /* entry in page directory for given address */
    pmd_t *pmd_entry;        /* entry in middle directory for buffer */
    pte_t *pte_entry;        /* entry in page table for buffer */
    unsigned long page_addr; /* address of page containing buffer */
    unsigned long phys_addr; /* physical address to be returned */
    int* ptr;

    /* get page directory entry */
    if (!task->mm) {
     printk("task->mm is null.\n");
     return 0;
    }
```

```
    pgd_entry = pgd_offset(task->mm, virt_addr);
    printk("pgd_entry:%08x\n",pgd_entry);
    if (!(pgd_val(*pgd_entry) & 0x1)) {
     printk("pmd is not present\n");
     return 0;
    }

    /* get page middle directory entry */
    /* note: on x86 without PAE, this is the same as pgd_entry */
    pmd_entry = pmd_offset(pgd_entry, virt_addr);
    printk("pmd_entry:%08x\n", pmd_entry);
    if (!pmd_present(*pmd_entry)) {
     printk("pte is not present\n");
     return 0;
    }

    /* get page table entry */
    pte_entry = pte_offset(pmd_entry, virt_addr);
    printk("pte_entry:%08x\n",pte_entry);

    /* check if page is present */
    if (!pte_present(*pte_entry)) {
         printk("page is not present\n");
     return 0;
    }

    /* get page address, can use macro pte_page too */
    /* pte_page return a pointer to the associated page structure */
    page_addr = pte_val(*pte_entry);

    /* finally we can compute the physical address */
    phys_addr = page_addr + (virt_addr & ~PAGE_MASK);

    return phys_addr;
}
```

五. 其他方法

自己编写查看 Linux 内核的程序是一个有趣的任务，但是实际上 Linux 内核已经提供了很多观察内核处理情况的窗口，透过他们我们也可以完成很多工作。内核通常是通过/proc 文件系统提供这些信息的，例如/proc/<进程号>中提供了每个进程(用户进程和内核线程)的状态，/proc/meminfo 提供了内存使用的统计，/proc/slabinfo 提供了 slab 分配器信息，/proc/loadavg 提供了系统负载统计等等。实际上很多系统监控程序就是通过这些文件提取内核信息的。

本文中所有例程在 2.4.18 核心的 Redhat 7.3/8.0 下调试通过。要说明的是 Redhat 7.3 默认安装的内核头文件(kernel-header)不正确，需要到 Redhat 网站上下载升级才能正确编译。

参考资料：

1．http://www.intel.com/design/PentiumIII/manuals/index.htm
2．《深入理解 Linux 内核》O’Reilly——中国电力出版社
3．http://www.xml.com/ldd/chapter/book——Linux 设备驱动程序第二版(英文)
4．http://home.earthlink.net/~jknapka
5．http://www.csn.ul.ie/~mel/projects/vm/
6．http://www-900.ibm.com/developerWorks/cn/linux/kernel/startup/index.shtml