乌有理工大学

《操作系统》课程实验报告

实验题目	**实验三：内存管理**		
姓　　名	**Nemo**	**学　　号**	**2019114514**
组　　别	–	**合 作 者**	–
班　　级	**计科 NaN 班**	**指导教师**	**Outis**

1　实验概述

1.1　实验介绍

本实验通过在内核态分配内存的任务操作，让学生们了解并掌握操作系统中内存管理的布局，内核态内存分配的实现，了解 openEuler 内核的物理内存管理，熟悉 `struct page` 数据结构。了解 openEuler 内核的物理内存管理，熟悉进程的地址空间的概念以及相关的数据结构。

1.2　任务描述

- 使用 `kmalloc` 分配 1KB、8KB 的内存，打印指针地址。
- 使用 `vmalloc` 分配 8KB、1MB、64MB 的内存，打印指针地址。
- 查看已分配的内存，根据机器是 32 位或 64 位的情况，分析地址落在的区域。
- 正确编写并安装设备驱动程序 pfstat.ko，用于读取物理页框属性。
- 正确编写并安装设备驱动程序 vma.ko，用于获得每个 VMA 的信息。

1.3　实验目的

- 正确编写满足功能的源文件，正确编译。
- 正常加载、卸载内核模块，且内核模块功能满足任务所述。
- 了解操作系统的内存管理。

2　实验内容

2.1　内核内存分配

2.1.1　实验过程

用 `make` 命令编译 kmalloc.c 和 vmalloc.c，用 `insmod` 命令加载编译完成的内核模块，用 `rmmod` 命令卸载内核模块，用 `dmesg` 命令查看内核模块在内核态的运行结果。

实验结果如下所示（openEuler 20.03）：

```
[root@ostest lab3]# cd task1
[root@ostest task1]# make
make -C /root/kernel M=/root/lab3/task1 modules
make[1]: Entering directory '/root/kernel'
  CC [M]  /root/lab3/task1/kmalloc.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /root/lab3/task1/kmalloc.mod.o
  LD [M]  /root/lab3/task1/kmalloc.ko
make[1]: Leaving directory '/root/kernel'
[root@ostest task1]# insmod kmalloc.ko
[root@ostest task1]# rmmod kmalloc.ko
[root@ostest task1]# dmesg | tail -n4
[  208.853053] Start kmalloc!
[  208.853241] kmallocmem1 addr = ffff80016fa4f000
[  208.853543] kmallocmem2 addr = ffff80016fcdc000
[  230.338834] Exit kmalloc!
[root@ostest task1]# cd ../task2
[root@ostest task2]# make
make -C /root/kernel M=/root/lab3/task2 modules
make[1]: Entering directory '/root/kernel'
  CC [M]  /root/lab3/task2/vmalloc.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /root/lab3/task2/vmalloc.mod.o
  LD [M]  /root/lab3/task2/vmalloc.ko
make[1]: Leaving directory '/root/kernel'
[root@ostest task2]# insmod vmalloc.ko
[root@ostest task2]# rmmod vmalloc.ko
[root@ostest task2]# dmesg | tail -n5
[  250.533425] Start vmalloc!
[  250.533651] vmallocmem1 addr = ffff00000b940000
[  250.534012] vmallocmem2 addr = ffff0000204c0000
[  250.534508] vmallocmem3 addr = ffff000023f00000
[  261.096602] Exit vmalloc!
```

使用 `vi kernel/arch/arm64/configs/openeuler_defconfig` 命令，结果如下：

```
#
# Automatically generated file; DO NOT EDIT.
# Linux/arm64 4.19.93 Kernel Configuration
#
...
CONFIG_ARM64=y
CONFIG_64BIT=y
CONFIG_MMU=y
CONFIG_ARM64_PAGE_SHIFT=16
...
# CONFIG_ARM64_4K_PAGES is not set
# CONFIG_ARM64_16K_PAGES is not set
CONFIG_ARM64_64K_PAGES=y
# CONFIG_ARM64_VA_BITS_42 is not set
CONFIG_ARM64_VA_BITS_48=y
CONFIG_ARM64_VA_BITS=48
...
CONFIG_PGTABLE_LEVELS=3
...
```

可知目前的配置是虚拟地址位数为 48 位，页表的大小是 64K，页表转化是 3 级。`kmalloc` 和 `vmalloc` 分配的内存地址，都位于内核空间。

2.1.2 思考题

内存泄露是指程序中已动态分配的内存未释放或无法释放；内存溢出是指程序在申请内存时，没有足够的内存空间供其使用；内存越界是指程序向系统申请一块内存后，使用时超出申请范围。

分析：下面两个程序是否会产生内存泄露、内存溢出或内存越界？

```
void function_which_allocates() { float *a = malloc(sizeof(float) * 45); }

int main() { function_which_allocates(); }
```

在 function_which_allocates 函数中的指针 a 在离开函数作用域后就销毁了，所以 malloc 给它的大小为 sizeof(float) * 45 = 180 B 的内存泄露了。如欲不泄露内存，需要在函数结束之前对指针进行 free 处理。

```
void func(char* input) {
  char buffer[16];
  strcpy(buffer, input);
}

int main() {
  char longstring[256];
  int i;
  for (i = 0; i < 255; i++) {
    longstring[i] = 'B';
  }
  func(longstring);
}
```

使用字符串复制函数 strcpy 需要考虑来源字符串和目标字符串的长度。strcpy 在 Linux 0.11 中的实现方式如下：

```
extern inline char *strcpy(char *dest, const char *src) {
  __asm__(
      "cld\n"
      "1:\tlodsb\n\t"
      "stosb\n\t"
      "testb %%al,%%al\n\t"
      "jne 1b"
      ::"S"(src), "D"(dest));
  return dest;
}
```

观察内嵌的汇编代码可知，如果 src 的长度大于 dest 的长度，当复制到目的缓冲器时，它会改写到连接目的缓冲器后方的存储器，导致无法预期的结果。这里 src 的长度是 256 而 dest 长度是 16，所以会产生内存越界。本程序在本机运行的结果是 illegal hardware instruction，在有些计算机上结果会是 segmentation fault。

2.2 物理页框属性

本实验期望编写一个内核模块，统计当前系统中的页情况，通过设备文件的 open 接口返回给用户态并打印。

本实验使用 get_num_physpages() (linux/mm.h) 来获取物理页数目,使用 pfn_to_page (asm-generic/memory_model.h) 宏来获得当前页的 struct page 指针。

查阅资料可知,struct page 拥有一个 32 位的 flags 域,用来存放页的状态。flag 的每一位单独表示一种状态，所以总共可以有 32 种不同的状态，这些状态在 linux/page-flags.h 中定义。linux/page-flags.h 中还定义了众多内联函数，用于测试页是否有某种状态，这些函数的原型是 static __always_inline int Page##uname(struct page *page)，如果测试页有状态 uname 则返回 1，否则返回 0。

核心代码和实验结果如下所示（openEuler 20.03）。

```
static ssize_t pfstat_read(struct file *file, char __user *out, size_t size,
                           loff_t *off) {
  struct page_info *buf = file->private_data;
  memset(buf, 0, sizeof(struct page_info));

  buf->num_physpages = get_num_physpages();
  printk(KERN_INFO "OK: %lu phys page(s) found. \n", buf->num_physpages);

  for (int cur_pfn = 0; cur_pfn < buf->num_physpages; cur_pfn++) {
    struct page *hoc = pfn_to_page(cur_pfn + ARCH_PFN_OFFSET);
    buf->valid += pfn_valid(cur_pfn + ARCH_PFN_OFFSET);
    buf->free += hoc->_refcount.counter == 0;
    buf->locked += PageLocked(hoc);
    buf->reserved += PageReserved(hoc);
    buf->swapcache += PageSwapCache(hoc);
    buf->referenced += PageReferenced(hoc);
    buf->slab += PageSlab(hoc);
    buf->private += PagePrivate(hoc);
    buf->uptodate += PageUptodate(hoc);
    buf->dirty += PageDirty(hoc);
    buf->active += PageActive(hoc);
    buf->writeback += PageWriteback(hoc);
    buf->mappedtodisk += PageMappedToDisk(hoc);
  }
  printk(KERN_INFO "OK: flags unwrapped in %lu page(s).\n", buf->num_physpages);

  copy_to_user(out, buf, sizeof(struct page_info));
  printk(KERN_INFO "OK: copy_to_user done.\n");
  return size;
}
```

```
[root@ostest pfstat]# ./run.sh
make -C /lib/modules/4.19.188/build M=/root/lab3/pfstat modules
make[1]: Entering directory '/root/kernel'
  Building modules, stage 2.
  MODPOST 1 modules
make[1]: Leaving directory '/root/kernel'
131072 physpages checked, total memory = 8192 MB
130978 valid pages, valid memory = 8186 MB
free: 96698
...↓
```

```
...↑
locked: 0
reserved: 21989
swapcache: 0
referenced: 4624
slab: 1123
private: 888
uptodate: 9490
dirty: 255
active: 6500
writeback: 0
mappedtodisk: 4614
```

使用 vi /proc/meminfo 命令，结果如下:

```
MemTotal:        6975296 kB
MemFree:         6177152 kB
MemAvailable:    6348672 kB
Buffers:           42752 kB
Cached:           453312 kB
SwapCached:            0 kB
Active:           416128 kB
Inactive:         230080 kB
Active(anon):     155264 kB
Inactive(anon):     7936 kB
Active(file):     260864 kB
Inactive(file):   222144 kB
Unevictable:           0 kB
Mlocked:               0 kB
SwapTotal:             0 kB
SwapFree:              0 kB
Dirty:               640 kB
Writeback:             0 kB
AnonPages:        150528 kB
Mapped:           104320 kB
Shmem:             13120 kB
KReclaimable:      28160 kB
Slab:              72000 kB
SReclaimable:      28160 kB
SUnreclaim:        43840 kB
KernelStack:       10048 kB
PageTables:         7936 kB
NFS_Unstable:          0 kB
Bounce:                0 kB
WritebackTmp:          0 kB
CommitLimit:     3487616 kB
Committed_AS:     517056 kB
VmallocTotal:   133009637312 kB
VmallocUsed:           0 kB
VmallocChunk:          0 kB
Percpu:             2304 kB
HardwareCorrupted:     0 kB
AnonHugePages:         0 kB
ShmemHugePages:        0 kB
ShmemPmdMapped:        0 kB
CmaTotal:         524288 kB
CmaFree:          523776 kB
HugePages_Total:       0
HugePages_Free:        0
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:     524288 kB
Hugetlb:               0 kB
```

发现部分结果符合内核模块的运行结果，内存分配是动态变化的。

2.3 进程地址空间

本实验期望编写一个内核模块，获取当前进程的地址空间信息以及每个 VMA 的信息，通过设备文件的 open 接口返回给用户态并打印。

本内核模块先在 Ubuntu 下编译运行。在 Ubuntu 的 mm_struct 中 locked_vm 是 unsigned long 型的，而 pinned_vm 是 atomic64_t 型的。第一次在 openEuler 运行时，编译报错：

```
/root/lab3/vma/vma.c: In function 'vma_read':
/root/lab3/vma/vma.c:42:20: error: incompatible types when assigning to type 'long unsigned
int' from type 'atomic_long_t {aka struct <anonymous>}'
     buf->locked_vm = cur_mm->locked_vm;
                    ^
/root/lab3/vma/vma.c:43:39: error: request for member 'counter' in something not a structure
or union
     buf->pinned_vm = cur_mm->pinned_vm.counter;
```

发现在 openEuler 中反之，locked_vm 是 atomic_long_t 型的。修改代码以便能在 openEuler 下运行，核心代码和实验结果如下所示。

```
static ssize_t vma_read(struct file *file, char __user *out, size_t size,
                        loff_t *off) {
  struct mm_struct *cur_mm = current->mm;
  struct buffer *buf = file->private_data;
  if (size == sizeof(struct buffer)) {
    buf->map_count = cur_mm->map_count;
    printk(KERN_INFO "OK: %d vma(s) found.\n", buf->map_count);
  } else {
    buf->task_size = current->mm->task_size;
    buf->highest_vm_end = cur_mm->highest_vm_end;
    buf->mm_users = cur_mm->mm_users.counter;
    buf->mm_count = cur_mm->mm_count.counter;
    buf->pgtables_bytes = cur_mm->pgtables_bytes.counter;
    buf->map_count = cur_mm->map_count;
    buf->total_vm = cur_mm->total_vm;
    buf->locked_vm = cur_mm->locked_vm.counter;
    buf->pinned_vm = cur_mm->pinned_vm;
    buf->data_vm = cur_mm->data_vm;
    buf->exec_vm = cur_mm->exec_vm;
    buf->stack_vm = cur_mm->stack_vm;
    buf->start_code = cur_mm->start_code;
    buf->end_code = cur_mm->end_code;
    buf->start_data = cur_mm->start_data;
    buf->end_data = cur_mm->end_data;
    buf->start_brk = cur_mm->start_brk;
    buf->brk = cur_mm->brk;
    buf->start_stack = cur_mm->start_stack;
    buf->arg_start = cur_mm->arg_start;
    buf->arg_end = cur_mm->arg_end;
    buf->env_start = cur_mm->env_start;
    buf->env_end = cur_mm->env_end;
    printk(KERN_INFO "OK: information of mm_struct got.\n");

    int nvma = (size - sizeof(struct buffer)) / sizeof(struct vma_struct);
    struct vma_struct *ret_vma_list = (struct vma_struct *)(buf + 1);
    struct vm_area_struct *vma_list = cur_mm->mmap;
    for (int i = 0; i < nvma; i++) {
...↓
```

```
...↑
      ret_vma_list[i].vm_start = vma_list[i].vm_start;
      ret_vma_list[i].vm_end = vma_list[i].vm_end;
      ret_vma_list[i].vm_flags = vma_list[i].vm_flags;
      if (vma_list[i].vm_file && vma_list[i].vm_file->f_path.dentry) {
        strncpy(ret_vma_list[i].filename,
                vma_list[i].vm_file->f_path.dentry->d_iname, 40);
      } else {
        strncpy(ret_vma_list[i].filename, "NULL", 5);
      }
    }
    printk(KERN_INFO "OK: information of %d vma(s) got.\n", nvma);
  }
  copy_to_user(out, buf, size);
  printk(KERN_INFO "OK: copy_to_user done.\n");
  return size;
}
```

```
[root@ostest vma]# ./run.sh
make -C /lib/modules/4.19.188/build M=/root/lab3/vma modules
make[1]: Entering directory '/root/kernel'
  CC [M]  /root/lab3/vma/vma.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /root/lab3/vma/vma.mod.o
  LD [M]  /root/lab3/vma/vma.ko
make[1]: Leaving directory '/root/kernel'
file opened.
t = 176, map_count: 20
datasize = 4656
task_size: 281474976710656, highest_vm_end: 281474958884864, mm_users: 1, mm_count: 1
map_count: 20, total_vm: 58, locked_vm: 0, pinned_vm: 0
data_vm: 8, exec_vm: 40, stack_vm: 3
start_code: 0x400000, end_code: 0x401070, start_data: 0x41fdc8, end_data: 0x420070
start_brk: 0x3c900000, brk: 0x3c930000, start_stack: 0xffffffeef0d50
arg_start: 0xffffffeeffd68, arg_end: 0xffffffeeffd71, env_start: 0xffffffeeffd71, env_end: 0xf...
addr_space print ok.
start: 0x400000, end: 0x410000, flags: 0x875, filename: test.o
start: 0xffff80016f3602b8, end: 0xffff4e230000, flags: 0x100073, filename: libbfd-2.33.1.so
start: 0xfffe04550000, end: 0xfffe04570000, flags: 0x75, filename: libpthread-2.28.so
start: 0xfffe03620000, end: 0xfffe03630000, flags: 0x100073, filename: libargon2.so.0
start: 0xffff80016f363ae8, end: 0xfffc8e610000, flags: 0x75, filename: timer.h
start: 0xaaadc67d0000, end: 0xaaadc67e0000, flags: 0x100871, filename: systemd-logind
start: 0xffff80016f3674e8, end: 0xaaadb9480000, flags: 0x100871, filename: gcc
start: 0xfffc474f0000, end: 0xfffc47500000, flags: 0xfb, filename: vma.o
start: 0xfffe03d10000, end: 0xfffe03d20000, flags: 0x100073, filename: libselinux.so.1
start: 0xffff80016f3601d0, end: 0xfffea07b0000, flags: 0x100073, filename: libm-2.28.so
start: 0xffff80016f3645c8, end: 0xfffc08980000, flags: 0x75, filename: rwlock.h
start: 0xfffe035c0000, end: 0xfffe035d0000, flags: 0x100073, filename: libjson-c.so.4.0.0
start: 0xfffe034a0000, end: 0xfffe034b0000, flags: 0x100073, filename: libpcre2-8.so.0.8.0
start: 0xfffe03be0000, end: 0xfffe03c20000, flags: 0x75, filename: liblz4.so.1.9.2
start: 0xffff80016f3677a0, end: 0xfffea0f20000, flags: 0x100073, filename: libpthread-2.28.so
start: 0xfffe04010000, end: 0xfffe04020000, flags: 0x100073, filename: libcap.so.2.27
start: 0xffff80016f36bab0, end: 0xaaabf2930000, flags: 0x100073, filename: NULL
start: 0xfffe03e10000, end: 0xfffe03e20000, flags: 0x100073, filename: libmount.so.1.1.0
start: 0xfffe03f50000, end: 0xfffe03f60000, flags: 0x100071, filename: libgcrypt.so.20.2.3
start: 0xffff80016f36e290, end: 0xfffea08d0000, flags: 0x100071, filename: libgmp.so.10.3.2
address of i is 0x0xffffffeef0bbc
start of print_mm: 0x400854
global val: 0x420068
local val: 0xffffffeef0bec
static val: 0x42006c
malloc alocated mem: 0x3c9007e0
```

使用命令 `vi /proc/vmallocinfo`，发现部分结果符合内核模块的运行结果。

3 小结

通过在内核态分配内存，了解并掌握操作系统中内存管理的布局和内核态内存分配的实现，了解 openEuler 内核的物理内存管理。通过编写内核模块熟悉了 `struct page` 数据结构，了解了 openEuler 内核的物理内存管理，熟悉了进程的地址空间的概念以及相关的数据结构。

本实验在华为弹性云服务器 root@121.36.*.*（openEuler 20.03）完成。