实验3第二部分 Linux进程内存信息统计

实验目的

- 本实验为内存实验。目的是，通过代码填空的方式，让学生了解真实Linux系统的内存管理方式。需要了解的内容有：
  - 了解Linux系统虚拟内存的管理方式（ch7_part2.pdf, P3~17）；
  - 了解什么是VMA，并遍历、统计VMA；
  - 了解Linux的多级页表机制，并自己完成一个从虚拟地址到物理地址的转换函数（ch7_part2.pdf, P18~46）；
  - 了解Linux的页面置换算法，统计页面冷热并输出至图表观察（ch7_part2.pdf, P82~86）；
  - 了解用户内存地址空间的分布（ch7_part1.pdf, P4~23），并将用户进程中的数据转储至外存。
- 为实现实验内容，本实验使用模块完成。虽然本实验的框架较为复杂，但其功能已经完全由助教实现，各位同学只需完成代码填空(TODO)即可。代码填空只涉及内存管理。
  - 从另一角度而言，本实验也提供了丰富的扩展资料，如模块的编写与使用、 sysfs的编写与使用、内核线程的使用、shell脚本的控制/循环语句等。若感兴趣，可以通过阅读实验文档和代码的方式自修。

实验环境

- OS: Ubuntu 20.06 LTS
- Linux内核版本: 不限（但建议5以上）

> 注：实验代码中所用的函数在不同的内核版本中的实现略有差别，为此我们提供了两种解决策略。第一种（**推荐**）是依据内核版本替换所使用的函数，详情见下文。一般来说，对本次实验而言，5以上的版本在操作上没有差别。第二种是固定所用的内核版本为5.9.0，固定系统内核版本方法请参考附录“编译替换内核版本”这一节。

实验时间安排

> 注：此处为实验发布时的安排计划，请以课程主页和课程群内最新公告为准

- 5.8早习题课，讲解实验
- 5.9晚实验课，检查实验
- 5.16晚实验课，检查实验
- 5.23日晚实验课，检查实验
- 5.30日晚实验课，检查实验及发布实验四
- 6月6日及之后，检查实验四及补检查实验三

> 补检查分数照常给分，但会有标记记录此次检查未按时完成，标记会在最后综合分数时作为一种参考。

友情提示

- 本次实验以实验一为基础。一些步骤（如如何启动qemu运行Linux内核）不会在实验说明中详述。如果有不熟悉的步骤，请复习实验一。
- 在本次实验中，如果你写出的代码出现了数组越界，则会出现很多奇怪的错误（如段错误（Segmentation Fault）、输出其他数组的值或一段乱码（注意，烫烫烫是Visual C的特性，Linux下没有烫烫烫）、其他数组的值被改变等）。提问前请先排查是否出现了此类问题。
- 如果同学们遇到了问题，请先查询在线文档。在线文档地址：https://docs.qq.com/sheet/DR1dZTnFRTURHc051
- **建议同学们在拿到代码之后先尝试直接编译，如果遇到问题再尝试替换内核版本。一般来说Linux 5以上版本都无需替换内核版本。**

警告

本实验的一部分操作可能会对Linux系统产生一些可逆的破坏性。若内核模块的代码不正确，可能会导致：

现象	解决方案
内核模块无法卸载。	重启linux。
内核模块卸载后再重新载入模块时，命令一直卡住。	重启linux。
死循环导致Linux死机，或内核日志过长，或内存爆满。	重启linux。
虚拟机报错：客户机已禁用CPU。请关闭或重置虚拟机。	重启虚拟机。
Linux磁盘爆满，重启后卡在/dev/sda1:clean,.../...files,.../...blocks进不去图形界面。	按ctrl+alt+F3进入文本模式，删除一些无用内容以清理磁盘。
Ubuntu报错：“检测到系统程序出现问题”，但关闭提示窗口后系统仍能正常运行。	删除系统内核错误报告。 **参考链接**

大部分情况均可通过表中所述解决方案解决。但考虑到重启系统、修复故障可能会花费一定时间，**建议使用快照保存系统状态**。但请注意，恢复快照前请保证自己已写的代码已被备份，即：不要让快照覆盖掉自己刚改好的代码。

致谢

感谢各位同学对本文档的纠错与补充提供的意见与建议。

第一部分 内存管理介绍

这一节中，我们给出一些内存管理中所必须的知识，帮助大家了解Linux系统内存管理子模块。主要涉及内存的基本管理单位页面（这里主要涉及4K大小的页面，暂时不考虑大页），进程虚拟空间的管理单位vma，系统如何判断页面非活跃以及页表的软件层次遍历。

> 前期准备工作（关闭透明大页）：

```
sudo sh -c "echo never > /sys/kernel/mm/transparent_hugepage/enabled"
sudo sh -c "echo never > /sys/kernel/mm/transparent_hugepage/defrag"
```

1.1 程序内存空间总述

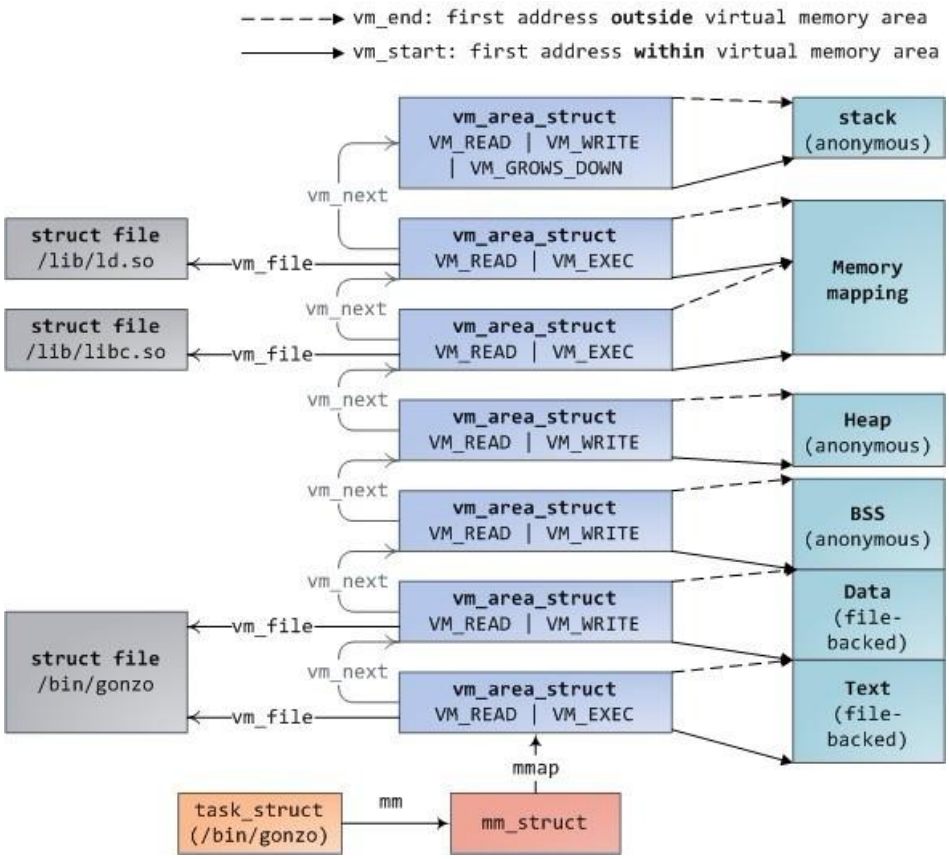
在Linux系统中，用户的内存空间可以分为以下几个层次：

- 段；
- VMA；
- 页。

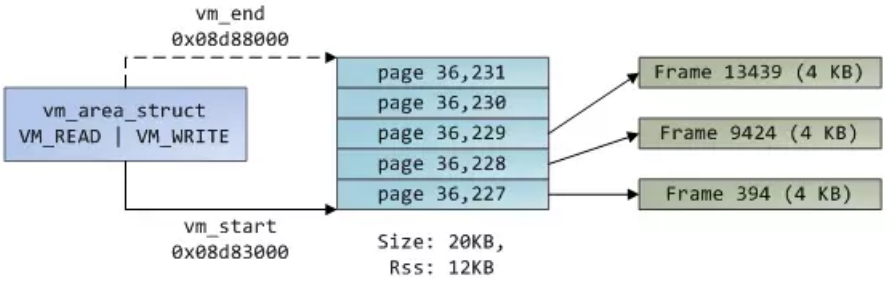
其中，如课程PPT(ch7_part1.pdf, P6)所述，用户的内存空间被划分为多个段，包括堆、栈、数据段、代码段等。然而这种分段方式存在一些问题：

- 为节省内存空间，动态链接库采用的是延迟加载机制，即程序在运行的过程中在被调用的时候才会被加载。如果进程的内存空间简单地按段分配，那么无法实现更细粒度的内存管理；
- 当程序要将数据写入文件时，若直接写入磁盘，开销将会很大，需要在内存进行缓冲以加快读写速度。
- 内存是按需分页的。当一个程序申请若干内存空间时，不需要真的给它分配那么多空间，可以等程序访问对应内存时再申请物理页。因此，内核也需要管理程序已经申请的虚拟空间。

为此，如下图所述，Linux将整个用户地址空间分为若干个**VMA（Virtual Memory Area，虚拟内存空间）**，即下图中的 `vm_area_struct` ，每一个段可能对应多个VMA。有的VMA与文件相关联，里面的页称为文件页；有的与文件无关，里面的页称为匿名页。



一个VMA内有多个页，如下图所示。



1.2 内存描述符

上节提到Linux操作系统使用虚拟内存方式来管理内存，即每个进程看到的都是只属于自己的虚拟地址空间。为此，Linux在每个进程的进程控制块结构体 `task_struct` 中记录了该进程的虚拟内存相关信息，即内存描述符成员 `struct mm_struct *mm` 。结构体 `mm_struct` 在 `include/linux/mm.h` 中定义。下面是该结构体的一些重要成员：

```
struct mm_struct {
    // 省略部分域展示
    /* 该进程的虚拟内存区域VMA链表 */
    struct vm_area_struct *mmap;
    /* 该进程的VMA红黑树根节点，用于加速查找特定VMA */
    struct rb_node vm_rb;
    /* 代码段和数据段的起始地址和结束地址 */
    unsigned long start_code, end_code, start_data, end_data;
    /* 堆的起始、结束地址；用户态栈的起始地址 */
    unsigned long start_brk, brk, start_stack;
```

```
    /* 进程在应用程序启动时传递参数字符串的起始、结束地址；环境变量的起始、结束地址 */
    unsigned long arg_start, arg_end, env_start, env_end;
    /* 页全局目录地址，详见1.6 */
    pgd_t * pgd;
}
```

1.3 虚拟内存区域VMA

本小节具体介绍VMA的数据结构以便各位同学更好的了解Linux如何管理内存。需要指出的是，Linux中将数据划分成了不同的区域，比如下面这些：

- 可执行文件的二进制代码，也就是程序的代码段；
- 存储全局变量的数据段；
- 用于保存局部变量和实现函数调用的栈；
- 环境变量和命令行参数；
- 程序使用的动态库的代码；
- 用于映射文件内容的区域。

而且在上节的内存描述符 `mm_struct` 中记录了相应的起始和结束地址，如代码段的起始地址 `start_code` 、结束地址 `end_code` 。这些区域仍旧以虚拟内存区域VMA来管理。一个合法地址总是落在某个区域当中的，这些区域也不会重叠。一VMA就是一块连续的线性地址空间的抽象，它拥有自身的权限（可读，可写，可执行等等），每一个虚拟内存区域都由一个相关的 `struct vm_area_struct` 结构来描述，具体如下：

```
struct vm_area_struct {
    struct mm_struct * vm_mm;   /* 所属的内存描述符 */
    unsigned long vm_start;    /* vma的起始地址 */
    unsigned long vm_end;       /* vma的结束地址 */

    /* 该vma的在一个进程的vma链表中的前驱vma和后驱vma指针，链表中的vma都是按地址来排序的*/
    struct vm_area_struct *vm_next, *vm_prev;

    struct rb_node vm_rb;      /* 红黑树中对应的节点 */

    /*anno_vma_node和annon_vma用于管理源自匿名映射的共享页*/
    struct list_head anon_vma_node; /* Serialized by anon_vma->lock */
    struct anon_vma *anon_vma;  /* Serialized by page_table_lock */

    /* Information about our backing store: */
    struct file * vm_file;          /* 映射的文件，没有则为NULL */
};
```

进程的若干个vma区域都得按一定的形式组织在一起，这些VMA都包含在进程的内存描述符(`mm_struct`)中，它们主要以两种方式进行组织，一种是链表的方式（按虚拟地址大小的顺序）链接，对应于 `mm_struct` 中的 `mmap` 链表头，一种是红黑树方式，对应于 `mm_struct` 中的 `mm_rb` 根节点，和内核其他地方一样，链表用于遍历，红黑树用于查找。

参考资料： **linux进程地址空间--vma的基本操作**

1.4 内存基本管理单元——页与页框

第一部分实验介绍了用户态内存分配通常是以块的形式。而在Linux中，也是以块的形式进行分配，即分配基本单元为块。在Linux中，进程中的（逻辑）内存块被称为页（Page），物理内存中的内存块被称为页框（Frame）。一般来说，页与页框的大小相等。Linux 通常把物理内存按每 4KB划分为一个页框，把虚拟地址按照4KB划分成一个个的页面。上一节提到了虚拟内存区域VMA，而页面时虚拟内存的基本管理单元，所以一个虚拟内存中会包含许多个虚拟页面，而且虚拟机内存必须包含整数个虚拟内存页。

利用页框机制有助于灵活分配内存地址，因为分配时不必要求必须有大块的连续内存，系统可以离散寻找空闲页凑出所需要的内存供进程使用。也就是说，对于一个进程而言，虚拟地址连续并不意味这物理地址也是连续的，连续的虚拟页可能会对应多个离散的物理页框。

在内核中，每一个物理页框使用一个**页描述符**（`struct page`）进行对应，所有的页描述符存放在一个**mem_map**（定义在`mm/memory.c`）线性数组之中。每一个页框的页框号（`physical address >> PAGE_SHIFT`）是页描述符在`mem_map`数组的位置(下标)。页框的分配使用了**伙伴系统。**

> 页框号（Page Frame Number，简称PFN）就是内存地址除以4096（一个页大小为4K=20^12，即4096字节）的值。`PAGE_SHIFT`定义在`arch/所用架构/asm/page_types.h`中，x86架构下其值为12。

在Linux内核中可以使用`follow_page`函数通过虚拟地址获取页描述符`struct page`：

```
// include/linux/mm.h
// 获取某虚拟地址信息对应的页面结构体struct page，其中vma表示虚拟内存区域，address表示虚拟地址。
// 为获取一个页，请将foll_flags参数设置为宏FOLL_GET。
static inline struct page *follow_page(struct vm_area_struct *vma,
        unsigned long address, unsigned int foll_flags)
```

> 注意：
>
> 1. `follow_page`函数未被内核导出，实验脚本已经实现了如何从内核中导出该函数。
> 2. 在Linux 3.9至4.19.241版本内，`follow_page`函数无法使用，需要使用`follow_page_mask`函数。其使用方法请参考 https://elixir.bootlin.com/linux/v4.19.240/source/include/linux/mm.h#L2587，即：定义一个无用的`int`型变量作为其最后一个参数。
> 3. `follow_page`有时会返回NULL，这是正常的，原因是按需分页（相关内容参见ch7_part2.pdf P47），一个虚拟地址只有在真正使用时系统才会分配物理页面。**请务必使用宏`IS_ERR_OR_NULL(page)`检查page是否非空且有效。**若想了解什么是无效指针，可以参考**此链接**。

页描述符`struct page`定义在`include/linux/mm.h`，新版本的定义在`include/linux/mm_types.h`中。下面是它的一些重要成员：

```
// 仅列出部分内容
struct page {
  /**
     * 一组标志，
     *   - 也对页框所在的管理区进行编号
     *   - 描述页框的状态：如 页被锁定，IO错误，被访问，被修改位于活动页，位于slab，页面活跃信息等。
     */
    page_flags_t flags;
  /**
```

```
     * 页框的引用计数。
     *   - 小于0表示没有人使用。
     *   - 大于0表示使用人数目
     */
    atomic_t _count;
  /**
     * 页框中的页表项数目（没有则为-1）
     *      -1:      表示没有页表项引用该页框。
     *      0:       表明页是非共享的。
     *      >0:      表示页是共享的。
     */
    atomic_t _mapcount;
  /**
     * 可用于正在使用页的内核成分（如在缓冲页的情况下，它是一个缓冲器头指针。）
     * 如果页是空闲的，则该字段由伙伴系统使用。
     * 当用于伙伴系统时，如果该页是一个2^k的空闲页块的第一个页，那么它的值就是k.
     * 这样，伙伴系统可以查找相邻的伙伴，以确定是否可以将空闲块合并成2^(k+1)大小的空闲块。
     */
    unsigned long private;
  /**
     * 当页被插入页高速缓存时使用或者当页属于匿名页时使用）。
     * - 如果mapping字段为空，则该页属于交换高速缓存(swap cache)。
     * - 如果mapping字段不为空，且最低位为1，表示该页为匿名页。同时该字段中存放的是指向anon_vma描述符的指针。
     * - 如果mapping字段不为空，且最低位为0，表示该页为映射页。同时该字段指向对应文件的address_space对象。
     */
    struct address_space *mapping;
  /**
     * 作为不同的含义被几种内核成分使用。
     *  - 在页磁盘映象或匿名区中表示存放在页框中的数据的位置。不是页内偏移量，而是该页面相对于文件起始位置，以页面为大小的偏移量
     *  - 或者它存放在一个换出页标志符。表示所有者的地址空间中以页大小为单位的偏移量，也就是磁盘映像中页中数据的位置 page->index是区域内的页索引或是页的线性地址除以PAGE_SIZE
     */
    pgoff_t index;
  struct list_head lru; // 包含页的最近最少使用的双向链表的指针， 用于伙伴系统。
#if defined(WANT_PAGE_VIRTUAL)
  /**
     * 如果进行了内存映射，就是内核虚拟地址。对存在高端内存的系统来说有意义。
     * 否则是NULL
     */
    void *virtual;
#endif /* WANT_PAGE_VIRTUAL */
}
```

在 `struct page` 中，flags是一个比较重要的成员，它记录了页面各种重要的标志位。这些标志位具体定义在 `include/linux/page-flags.h` 的 `enum pageflags` 中，具体如下：

```
enum pageflags {
    PG_locked,      /* 表示页面已上锁，不要访问 */
    PG_error,     /* 表示页面发生IO错误 */
```

```
    PG_referenced, /* 用于RCU和LRU算法 */
    PG_uptodate,   /* 表示页面内容有效，当该页面上读操作完成后，设置该标志位 */
    PG_dirty,      /* 表示页面是脏页，内容被修改过 */
    PG_lru,        /* 表示该页面在LRU链表中 */
    PG_active,     /* 表示该页面在活跃LRU链表中 */
    PG_slab,       /* 表示该页面是属于slab分配器创建的slab */
    PG_owner_priv_1,     /* 页面的所有者使用，如果是pagecache页面，文件系统可能使用*/
    PG_arch_1,       /* 与体系架构相关的页面状态位 */
    PG_reserved,     /* 表示该页面不可被换出，防止该page被交换到swap */
    PG_private,     /* 如果page中的private成员非空，则需要设置该标志，如果是pagecache, 包含fs-
private data */
    PG_private_2,       /* 如果是pagecache, 包含fs aux data */
    PG_writeback,       /* 页面正在回写 */
    PG_head,        /* A head page */
    PG_swapcache,       /* 表示该page处于swap cache中 */
    PG_mappedtodisk,    /* 表示page中的数据在后备存储器中有对应 */
    PG_reclaim,     /* 表示该page要被回收，决定要回收某个page后，需要设置该标志 */
    PG_swapbacked,      /* 该page的后备存储器是swap/ram，一般匿名页才可以回写swap分区 */
    PG_unevictable,     /* 该page被锁住，不能回收，并会出现在LRU_UNEVICTABLE链表中，它包括的几种
page：ramdisk或ramfs使用的页、shm_locked、mlock锁定的页 */
#ifdef CONFIG_MMU
    PG_mlocked,     /* 该page在vma中被锁定，一般是通过系统调用mlock()锁定了一段内存 */
#endif
    //......
};
```

接下来主要介绍实验中涉及到的标志位PG_active和PG_referenced。这两个标志位用于内存回收。Linux 中的页面回收是基于 LRU（least recently used，即最近最少使用）算法的。LRU 算法基于局部性原理，即过去一段时间内频繁使用的页面，在不久的将来很可能会被再次访问到。反过来说，已经很久没有访问过的页面在未来较短的时间内也不会被频繁访问到。因此，在物理内存不够用的情况下，很久未被访问的页面成为被换出的最佳候选者。

LRU 算法的基本原理很简单，为每个物理页面绑定一个计数器，用以标识该页面的访问频度。操作系统内核进行页面回收的时候就可以根据页面的计数器的值来确定要回收哪些页面。然而，在硬件上提供这种支持的体系结构很少，Linux 操作系统没有办法依靠这样一种页计数器去跟踪每个页面的访问情况，所以，Linux 在页表项中增加了一个 Accessed 位，当页面被访问到的时候，该位就会被硬件自动置位。该位被置位表示该页面还很年轻，不能被换出去。此后，在系统的运行过程中，该页面的年龄会被操作系统更改。在 Linux 中，相关的操作主要是基于两个 LRU 链表以及两个标识页面状态的标志符。

在 Linux 中，操作系统对 LRU 的实现主要是基于一对双向链表：active 链表和 inactive 链表，这两个链表是Linux 操作系统进行页面回收所依赖的关键数据结构，每个内存区域都存在一对这样的链表。顾名思义，那些经常被访问的处于活跃状态的页面会被放在 active 链表上，而那些虽然可能关联到一个或者多个进程，但是并不经常使用的页面则会被放到 inactive 链表上。页面会在这两个双向链表中移动，操作系统会根据页面的活跃程度来判断应该把页面放到哪个链表上。页面可能会从 active 链表上被转移到 inactive 链表上，也可能从 inactive 链表上被转移到 active 链表上，但是，这种转移并不是每次页面访问都会发生，页面的这种转移发生的间隔有可能比较长。那些最近最少使用的页面会被逐个放到 inactive 链表的尾部。进行页面回收的时候，Linux 操作系统会从 inactive 链表的尾部开始进行回收。简单说，就是针对匿名页和文件页都拆分成一个活跃，一个不活跃的链表。Linux 引入了两个页面标志符 **PG_active** 和 **PG_referenced** 用于标识页面的活跃程度，从而决定如何在两个链表之间移动页面。PG_active 用于表示页面当前是否是活跃的，如果该位被置位，则表示该页面是活跃的。

PG_referenced 用于表示页面最近是否被访问过，每次页面被访问，该位都会被置位。Linux 必须同时使用这两个标志符来判断页面的活跃程度，假如只是用一个标志符，在页面被访问时，置位该标志符，之后该页面一直处于活跃状态，如果操作系统不清除该标志位，那么即使之后很长一段时间内该页面都没有或很少被访问过，该页面也还是处于活跃状态。为了能够有效清除该标志位，需要有定时器的支持以便于在超时时间之后该标志位可以自动被清除。然而，很多 Linux 支持的体系结构并不能提供这样的硬件支持，所以 Linux 中使用两个标志符来判断页面的活跃程度。

```
// include/linux/mm_types.h
// 获取page引用计数，不为0意味着活跃
// 注意其中的memcg参数，在不同的内核版本中略有区别：
// 在Linux 3.19至5.10.113版本中，memcg可以用page->mem_cgroup得到
// 而在Linux 5.11及之后版本中需要使用(struct mem_cgroup *)(page->memcg_data)得到
// 各位同学在使用本函数时请注意自己所用的内核版本。
int page_referenced(struct page *, int is_locked,struct mem_cgroup *memcg, unsigned long
*vm_flags);
```

Linux 中实现在 LRU 链表之间移动页面的关键函数如下所示：

- `mark_page_accessed()`：当一个页面被访问时，该函数会被调用以修改 `PG_active` 和 `PG_referenced`。
- `page_referenced()`：当操作系统进行页面回收时，每扫描到一个页面，就会调用该函数设置页面的 `PG_referenced` 位。如果一个页面的 `PG_referenced` 位被置位，但是在一定时间内该页面没有被再次访问，那么该页面的 `PG_referenced` 位会被清除。
- `activate_page()`：该函数将页面放到 `active` 链表上去。
- `shrink_active_list()`：该函数将页面移动到 `inactive` 链表上去。

参考资料：

- **Linux物理内存分配**
- **Linux内存管理篇之页框管理**

1.5 地址转换——页表

进程使用的虚拟地址需要转换成物理地址，并根据此物理地址访问内存数据。用来将虚拟地址转换到物理地址的数据结构称为页表。实现两个地址空间的关联最容易的方式是使用数组：对虚拟地址空间中的每一页，都分配一个数组项，指向与之关联的页框。但这样做会产生很多问题：

1. IA-32体系结构使用4KB大小的页，在虚拟地址空间为4GB的前提下，页表项数可达100万项，其大小为4MB。在64位体系结构下，这个问题会变得更加糟糕。
2. 每个进程都需要一套页表。如果系统的进程数量较多，会导致系统中大量的内存都用来保存页表。
3. 很多情况下进程用不满全部的4GB虚拟内存空间。直接分配一个较大的页表会产生空间浪费。

为此，Linux采用**多级页表**管理进程的内存空间。为同时支持适用于32位和64位的系统，Linux采用了通用的分页模型。Linux在2.6.10版本中采用了三级分页模型，从2.6.11开始采用了四级分页模型，从4.12开始采用了五级分页模型。五级分页模型如下表：

单元	描述
页全局目录	Page Global Directory (PGD)
页上级目录	Page Upper Directory (PUD)
页中间目录	Page Middle Directory (PMD)
页表	Page Table (PT)
页内偏移	Page Offset

在多级页表模型下，页分为页目录表、页表和页三类：

- 页目录表是一个特殊的页，记录该地址对应的子目录或页表的物理内存基址；
- 页表也是一个特殊的页，记录该地址对应的页的物理内存基址。

下图形象地展示了x86-64架构下四级页表的结构。在x86-64架构下，内存地址是64位的，即每个内存地址占8字节。每个4KB大小的页可以存放2^9个地址，所以在虚拟地址中，每级页表都分了9位。虽然每个页表（或页目录表）项长达64位，但因为其记录的是一个页（页表、页目录表本质也是一个页）的基址，而页地址永远4KB对齐，所以这64位地址的低12位永远为0。因为物理地址是64位（8Bytes）对齐的，所以在这低12位里面低3位也永远为0，中间的9位就是页表（或页目录表）项索引（可以通俗地理解为偏移量）。

以下图为例，Cr3寄存器中存放了一个进程PGD表的起始物理地址，该地址加上虚拟地址的PGD部分（PGD部分可以理解成偏移量）就可以得到PUD的物理地址。以此类推，就可以得到PUD表的起始物理地址。

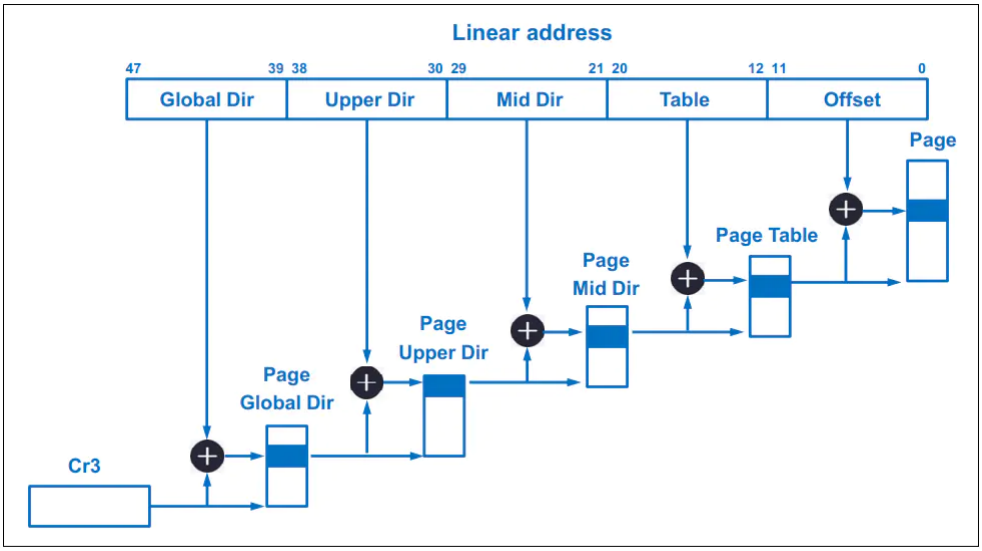


Figure 1　Structure of the 4-level paging

具体而言，如上图，Linux内核通过如下流程完成虚拟地址到物理地址的转换（以64位机器、四级页表为例）：

1. 从CR3寄存器中（也可以从`mm_struct->pgd`）读取PGD（页全局目录）所在物理页面的物理地址（即所谓的页目录基址）。
2. 获取内存地址的PGD（页全局目录）项索引（即：虚拟地址的39~47位），然后查PGD表，找到该地址对应的PGD表项。该表项即为该虚拟地址所属的页上级目录（PUD）的物理基址。
3. 获取内存地址的PUD（页上级目录）项索引（即：虚拟地址的30~38位），然后查上步获取的PUD表，找到该地址对应的PMD表项。该表项即为该虚拟地址所属的页中间目录（PMD）的物理基址。
4. 获取内存地址的PMD（页中间目录）项索引（即：虚拟地址的21~29位），然后查上步获取的PMD表，找到该地址对应的PMD表项。该表项即为该虚拟地址所属的页表（PT）的物理基址。

5. 获取内存地址的页表项（PTE）索引（即：虚拟地址的12~20位），然后查上步获取的页表，找到该物理页面（即页框）的物理基址（即页框编号，PFN）。
6. 获取页面的物理基址后，加上0~11位的offset，就可以得到最终的物理地址，进而访存。

Linux内存管理中涉及页表转换的宏定义如下：

```
/*描述各级页表中的页表项*/
typedef struct { pteval_t pte; } pte_t;
typedef struct { pmdval_t pmd; } pmd_t;
typedef struct { pudval_t pud; } pud_t;
typedef struct { pgdval_t pgd; } pgd_t;

/*  将页表项类型转换成无符号类型 */
#define pte_val(x)  ((x).pte)
#define pmd_val(x)  ((x).pmd)
#define pud_val(x)  ((x).pud)
#define pgd_val(x)  ((x).pgd)

/*  将无符号类型转换成页表项类型 */
#define __pte(x)    ((pte_t) { (x) } )
#define __pmd(x)    ((pmd_t) { (x) } )
#define __pud(x)    ((pud_t) { (x) } )
#define __pgd(x)    ((pgd_t) { (x) } )

/* 获取页表项的索引值 */
#define pgd_index(addr)     (((addr) >> PGDIR_SHIFT) & (PTRS_PER_PGD - 1))
#define pud_index(addr)     (((addr) >> PUD_SHIFT) & (PTRS_PER_PUD - 1))
#define pmd_index(addr)     (((addr) >> PMD_SHIFT) & (PTRS_PER_PMD - 1))
#define pte_index(addr)     (((addr) >> PAGE_SHIFT) & (PTRS_PER_PTE - 1))

/*  获取页表中entry的偏移值 */
#define pgd_offset(mm, addr)    (pgd_offset_raw((mm)->pgd, (addr)))
#define pgd_offset_k(addr)  pgd_offset(&init_mm, addr)
#define pud_offset_phys(dir, addr)  (pgd_page_paddr(*(dir)) + pud_index(addr) *
sizeof(pud_t))
#define pud_offset(dir, addr)       ((pud_t *)__va(pud_offset_phys((dir), (addr))))
#define pmd_offset_phys(dir, addr)  (pud_page_paddr(*(dir)) + pmd_index(addr) *
sizeof(pmd_t))
#define pmd_offset(dir, addr)       ((pmd_t *)__va(pmd_offset_phys((dir), (addr))))
#define pte_offset_phys(dir,addr)   (pmd_page_paddr(READ_ONCE(*(dir))) + pte_index(addr) *
sizeof(pte_t))
#define pte_offset_kernel(dir,addr) ((pte_t *)__va(pte_offset_phys((dir), (addr))))
#define pte_page(pte)   pfn_to_page(pte_pfn(pte))
static inline unsigned long pte_pfn(pte_t pte);

/* 检查页（目录）表项是否为空 */
static inline int pgd_none(pgd_t pgd);
static inline int pud_none(pud_t pud);
static inline int pmd_none(pmd_t pmd);
static inline int pte_none(pte_t pte);

/* 检查页（目录）表项是否可用 */
```

```
static inline int pgd_bad(pgd_t pgd);
static inline int pud_bad(pud_t pud);
static inline int pmd_bad(pmd_t pmd);
static inline int pte_bad(pte_t pte);
```

> 提示：Linux在4.12开始采用了五级分页模型，在pgd和pud之间加了一个p4d。因此，`pud_offset`接受的第一个参数类型应当为`p4d_t *`。本试验所用程序实际用不到此级，所有的虚拟地址这段都是0，所以可以将获得的`pgd_t *`类型变量强制转换为`p4d_t *`。

请通过自行阅读源码，或阅读下列参考资料了解各转换宏的使用方法。

参考资料：

- **Linux kernel内存管理**
- **Linux分页机制**
- **Linux中的页表实现**
- **深入理解计算机系统-之-内存寻址（五）-页式存储管理**，详细讲解了传统的页式存储管理机制；
- **深入理解计算机系统-之-内存寻址（六）-linux中的分页机制**，详细的讲解了Linux内核分页机制的实现机制。

第二部分 实验流程及要求

实验的第二部分分为以下几个部分：

- 统计进程VMA数目
- 页表遍历及页面冷热统计
- 进程信息获取

> 注：
>
> 1. 提示：完成提供的`ktest.c`中的代码填空。
> 2. 为了简化代码，本次实验均在系统仅存在小页的环境下完成，需关闭系统透明大页设置。（参考本文档第一部分的开头内容）
> 3. 在测试时，下文的“测试程序”特指使用我们已经提供了的`workload.cc`编译运行产生的进程。

2.1 测试程序、代码框架

2.1.1测试程序

我们提供了workload.cc程序，同学们可以编译并运行（./run.sh）该应用程序，然后通过完成4.1~4.5来监控该进程的相关信息。该测试程序与lab3.1提供的测试程序**不同**：这里直接用`malloc`分配空间，而不是利用第一部分编写的内存分配器。

该程序完整地运行一次大概需要一分多钟。

> 注：编译workload.cc的方式为：`g++ -std=c++11 workload.cc -o [exe file name] -lpthread`

2.1.2 代码框架

本实验需要完成`ktest.c`的代码填空。`ktest.c`是实现的一个Linux内核模块，该模块会周期性的输出用户指定的进程PID的内存信息，即执行一些进程内存信息输出后就睡眠一定时间（睡眠时间也由用户指定）。运行过程中用户可以选择功能执行，同时用户也可以指定是否开启此模块，如果开启则模块会周期性运行，否则将睡眠，直到用户开启此模块。用户还可以指定该内核模块所针对的进程PID，指定PID后，该内核线程将周期性输出该进程的内存信息。`ktest.c`所用的参数是通过sysfs文件系统实现，各位同学可以通过下面的路径进行查看和修改：

文件路径	功能	用户操作权限
/sys/kernel/mm/ktest/func	选择功能	可读可写
/sys/kernel/mm/ktest/sleep_millisecs	扫描时间间隔	可读可写
/sys/kernel/mm/ktest/run	是否开启模块功能	可读可写
/sys/kernel/mm/ktest/pid	选择哪个进程	可读可写
/sys/kernel/mm/ktest/vma	对应进程的虚拟内存区域VMA的总数	只读

本实验已经`ktest.c`中实现了代码框架，各位同学无需修改代码框架。已实现的功能有：

- sysfs的输入与输出。程序会自动地从sysfs读取内容并存放入全局变量内，也会自动地将需要的变量输出到sysfs中。
- 使用内核线程实现了每5秒扫描一次sysfs，读取pid、func和sleep_milliseconds。程序会自动地根据读取到的pid找到进程的`task_struct`结构体并放入`my_task_info`内。
    - 需要指定pid为2.1.1所述的测试程序。注：本试验已经在2.8节提供了一个自动化脚本，该脚本会自动向sysfs内写入测试程序的pid并完成测试。所以这部分无需操作。
    - 需要指定func为下述要完成的实验功能。注：本试验已经在2.8节提供了一个自动化脚本，它会接受参数并自动向sysfs内写入func并完成测试。所以这部分也无需操作。
    - sleep_milliseconds使用模块程序默认的5秒即可。这个参数代表我们的程序每5秒扫描一次并输出一次。
- 代码框架提供了大量注释，如有需要可以阅读。

> 本试验涉及了较多新的知识，比如Linux内核模块、sysfs文件系统、kthread线程等。如果同学希望了解这部分内容，请参考附录3和附录4这两部分。

2.1.3 如何进行简单的编译

1. 解压提供的"lab3-part2.zip"。
2. 编译时文件路径不能带空格。所以请重命名解压缩后的文件夹。文件夹名下称`lab3.2`。
3. 为`run_expr.sh`添加执行权限：`sudo chmod +x run_expr.sh`
4. 执行自动化脚本进行测试运行：`sudo ./run_expr.sh 0`，其中0是参数，选择使用哪个功能func。参数为0表示每5秒`printk`一次helloworld，你可以另开一个终端并通过1.3.1介绍的方法读取。
5. 等待脚本结束即可。无需手动卸载模块，因为脚本在每次加载模块之前都会自动检查模块是否在运行，如有就卸载模块。

> 脚本并不会检查编译是否通过。若编译报错，脚本仍然会运行workload和上次编译输出的模块二进制文件。若遇到“找不到sys/kernel/...”等，建议检查编译输出。

在接下来的2.2至2.6节，将介绍实验要求完成的5个功能。需要各位同学完成 `ktest.c` 中的代码填空以实现这些功能。实现完成后，可以通过运行 `sudo ./run_expr.sh X`，其中 `X` 为数字1至5，以实现功能1至5的自动化运行。

2.2 func = 1：每隔5s统计测试程序的VMA数量

1. 需要的文件

- 修改 /sys/kernel/mm/ktest/func=1：选择功能1运行
- 修改 /sys/kernel/mm/ktest/pid为特定的进程PID
- 输出： /sys/kernel/mm/ktest/vma：对应进程的VMA数量。预期值在39左右。可通过 `cat /sys/kernel/mm/ktest/vma` 获取输出的结果。

2. 任务提示

- 遍历vma需要使用到 `mm_struct` 描述符中的 `mmap` 成员；

2.3 func = 2：每隔5s统计测试程序的页面冷热信息

1. 需要的文件

- 修改 /sys/kernel/mm/ktest/func=2：选择功能2运行
- 修改 /sys/kernel/mm/ktest/pid为特定的进程PID
- 输出：输出页面冷热信息到文件中，方便进行图形绘制工作。**输出要求：输出页面的十进制物理地址，且输出之间用“,”（英文逗号）连接，每扫描一次所有页面执行一次换行，以便直接使用我们提供的画图脚本进行画图。行末不要有逗号，否则会报错**“invalid literal for int() with base 16: '' ”。

2. 任务提示

- 流程：遍历vma，再遍历vma中的虚拟地址（间隔： `PAGE_SIZE` ，即4096），使用 `follow_page` 函数通过虚拟地址获取页面结构体 `struct page` ，然后通过 `page_referenced` 函数判断页面最近是否被引用过。内核态进行文件读写可以参考 **这里** 。我们提供了一个示例输出，其使用方法详见4.8。
- 内核态进行文件读写可以参考 **这里** 。 `kernel_write` 函数的用法：

```
// 将 buf[0..count - 1] 里面的内容输出到 file 对应的文件中，
// 其中，file 对应的文件初始偏移是 offset
int kernel_write(struct file *file, char *buf, size_t count, loff_t *offset);
```

- 本func推荐使用接口：本试验已经在代码中对 `kernel_write` 进行了二次封装，提供了 `record_one_data` 接口用于向文件输出。
- 不建议在虚拟机中打开输出的文件，因为文件内容过长，可能会卡死。

3. 按照如下命令来安装画图脚本所用的库：

```
sudo apt install python3
sudo apt install python3-pip
sudo python3 -m pip install matplotlib
```

若按照指令安装matplotlib之后仍报错找不到matplotlib，可能是因为没有在`sudo`下安装matplotlib，建议在指令前加`sudo`。若仍无法解决，建议使用`python3 draw.py`指令手动作图。

2.4 func = 3：遍历测试程序的页表，并打印所有页面物理号

1. 需要的文件

- 修改 /sys/kernel/mm/ktest/func=3：选择功能3运行
- 修改 /sys/kernel/mm/ktest/pid为特定的进程PID
- 输出：输出到文件中，虚拟地址和物理页号对应。

> 注：你需要利用我们介绍的页表转换宏（或函数）**逐级**查找多级页表找到页面的PFN，不准直接用`follow_page`找到页描述符然后用`page_to_pfn(struct page)`转换为PFN。但是你可以将二者对照并比较结果是否正确。
>
> 虽然代码的注释给出的方法是pgd->pud->pmd->pte->page->pfn，但你也可以直接从pte转为pfn。这两种做法在检查时都算正确。

2. 任务提示

- 流程：遍历vma，再遍历vma中的虚拟地址（间隔：PAGE_SIZE -- 4K），通过虚拟地址进行解析，得到对应的物理页号（详见3.6）。
- 本func推荐使用接口：我们在代码中对`kernel_write`进行了二次封装，提供了`record_two_data`接口用于向文件输出。
- 不建议在虚拟机中打开输出的文件，因为文件内容过长，可能会卡死。

2.5 func = 4：dump测试程序的数据段

1. 需要的文件

- 修改 /sys/kernel/mm/ktest/func=4：选择功能4运行
- 修改 /sys/kernel/mm/ktest/pid为特定的进程PID
- 输出：输出到文件中。

2. 任务提示

- 流程：首先根据 pid 获取对应的`mm_struct`，获得里面的`start_data`和`end_data`地址，之后遍历所有地址，找该地址对应的VMA，再找到页面描述符，之后根据代码注释将对应的数据写入到文件中。`mm_struct`中的成员变量已在3.2讲解。
- 进程的各个段的起始地址、终止地址可能不是4K的整数倍；
- 提示一个函数：`find_vma`（位于include/linux/mm.h），其含义请自行阅读源码的注释。

```
extern struct vm_area_struct * find_vma(struct mm_struct * mm, unsigned long addr);
```

- 本func推荐使用接口：我们在代码中对`kernel_write`进行了二次封装，可以先将数据写入全局变量`buf`中，再调用`flush_buf`函数将其写入文件。不调用`flush_buf`会导致文件没有输出。你们可以自由选择使用我们提供的二次封装或直接使用`kernel_write`。

- 输出的内容中有一些乱码是正常的：这是因为数据段并不只有`hamlet_scene_1`等能显示出来的字符串，还有一些其他的全局变量，而这些东西往往是二进制的，我们无法用肉眼看出结果。此外，目前发现在一些机器上运行正确的代码在某些机器上只能打印出`hamlet_scene_1`字符串的一半，后面的不能打印。对此，我们的解决方案是：检查时只需要能够看到workload.cc中`hamlet_scene_1`、`gloal_data`两个字符串的内容即可，不要求完整性。
- 不要求输出`.rodata`段的"Malloc failed at getRandomString"等字符串。是否输出都无所谓。

2.6 func = 5： dump测试程序的代码段

1. 需要的文件

- 修改 /sys/kernel/mm/ktest/func=5：选择功能5运行
- 修改 /sys/kernel/mm/ktest/pid为特定的进程PID
- 输出：输出到文件中。

2. 任务提示

- 流程：首先根据 pid 获取对应的`mm_struct`，获得里面的`start_code`和`end_code`地址，之后将对应的数据通过`kernel_write`写入到文件中即可。
- dump出来的东西是二进制文件。因为func=4和func=5的原理相同，所以本部分主要通过观察代码进行检查。

2.7 任务评分规则

满分6分，包括：

1. 每个任务(func)完成，得1分。总共5个func。
2. 代码讲解，得1分。

> 注：实验检查时会要求先确认运行结果，再简单解释一下如何实现。

2.8 自动化运行实验脚本

为了方便学生开展实验，我们提供了自动化运行代码的脚本。其工作过程如下：

1. 禁用大页。
2. 更新代码中涉及`follow_page`和`page_referenced`地址的相应行，这样即使上述两个函数的地址发生了变化，也能及时的进行地址修改从而避免出错。
3. 加载编写的内核模块。
4. 编译workload.cc。
5. 运行workload。
6. 运行编写的内核模块。

7. 等待workload运行结束。运行结束后，如果是func=2，脚本会将expr_result.txt重命名为active_page_log.txt。其他func下脚本会将expr_result.txt重命名，在其文件名后加上时间。
8. 画图或者保持生成的数据段/代码段等结果。

自动化运行实验脚本代码保存在压缩包内 `run_expr.sh` 。如有兴趣可以阅读脚本代码并学习其中的语法。

> 脚本并不会检查编译是否通过。若编译报错，脚本仍然会运行workload和上次编译输出的模块二进制文件。若遇到“找不到sys/kernel/...”等，建议检查编译输出。

2.9 实验材料的目录结构

- lab3-part.zip
  - trace目录：测试程序
  - helloworld目录：附录3.3.2描述的一个简易的打印helloworld的模块代码。
  - sysfs目录：附4.2.3描述的一个简易的sysfs模块代码。
  - run_expr.sh：2.8描述的自动化运行脚本。
  - ktest.c：本实验的代码框架。
  - draw.py：2.3的画图脚本。
  - active_page_log.txt：画图脚本的参考标准输入。你们需要能够跑出自己的active_page_log.txt。受采样时间的影响，自己跑出log的可以和示例不一致，但画出的图应该与示例接近（这里的接近指输出在时间上具备一定的周期性，如，每2~3个点较密集的横线中夹杂一条点较稀疏的横线）。直接运行 `python3 draw.py` 就可以读取log并输出图像。
  - Makefile。

附1 实验第一部分选做

在实验第一部分中，我们实现了一个64位内存分配器，并且对其内存使用效率进行了测试与分析。我们会发现在调用过free后，内存的使用率会降到一个很低的水平，这是因为虽然用户调用free还给了内存分配器，但是内存分配器并没有还给内核，依然处于占用状态。在实际的分配器中，对**空闲内存的回收**是提高内存使用率、在必要时将空闲内存归还给内核的有效手段。因此，本试验给出一个相关的选做题目如下：

实现分配器的内存回收机制，具体要求：

- 该回收机制必须支持按页回收分配器中空闲内存
  - 回收可以发生在任何空闲页上（而非仅堆尾的空闲页），因此用 `sbrk()` 构造的连续地址空间堆无法满足该要求。你需要将 `sbrk()` 获取内存的方式改为使用 `mmap()` ，然后使用 `munmap()` 以4KB页为单位对齐回收（参考第一部分文档中1.1节）；
  - 回收的粒度是4KB页，也即只有地址按4KB对齐的整块页全部空闲时，你才可以将这块页归还给操作系统。也可以简单理解为， `mmap()` 返回地址开始，每4KB是一个页，然后以页为单位去判断是否可以回收。
- 该回收机制必须支持异步回收：

    - 最终需要呈现出一个内存回收调用接口（比如一个名为 `garbage_collection()` 的函数），这个接口会去扫描空闲链表，查找可回收页并通过适当的系统调用函数进行回收。之后修改负责分配器初始化的 `mm_init()` 函数，在其中额外启动一个内存回收守护线程，该线程每隔固定时间调用一次上述内存回收接口；
    - 该内存回收守护线程在异步执行回收任务的过程中，可能与 `mm_malloc()` 或 `mm_free()` 产生竞争。你需要通过实现页粒度的锁管理，确保分配器的申请/释放功能，与内存回收功能可以在保证一定效率的前提下并发执行（你不能对整个空闲链表全局加锁，因为这样产生的效果与同步回收几乎没有区别）
- 该回收机制需要额外测试：
    - 修改原先内存使用率的统计方法，以确保实现回收机制后，仍能统计到正确的内存使用率；
    - 定量分析出加入回收机制后内存使用率的提升；
    - 展示内存分配器在时间性能上的变化（给出测试结果即可）。

实现上述全部功能和要求，并在实验检查时，向助教流畅讲述代码逻辑及展示测试效果，**可以得到额外的1'加分**。本选做题目较为开放，因此无具体评分细则，由助教在检查时酌情判断。

附2 编译替换内核版本

若本机的Linux版本不能满足实验需要，可以考虑更改Linux内核版本（**这在本次实验中并不是必须的**）。下面是修改内核版本的流程：

注意，在替换内核版本之前，请做好文件备份。虚拟机可以通过生成系统快照的方式进行快捷备份。

附2.1 为什么要替换内核版本

- 为什么要替换内核版本?

模块在编译时会从你的系统中寻找linux头文件，而不同版本的内核头文件不一定相同。模块在运行时会从你的系统中寻找依赖库，而不同版本的内核依赖库也不一定相同。所以，在一个版本下编译运行成功的模块不一定能在其他版本下正常运行。所以，为了方便检查，我们将本次实验的内核版本设置为 5.9。当然，你也可以在其他内核下进行实验，不过不保证助教给的代码可以在其他内核上正常运行。

- 那为什么不用之前已经配置好的qemu环境?

这是因为需要在本地环境编译打包进_install目录下，操作会很麻烦。所以建议本地操作。

附2.2 查看本系统内核版本

```
$ uname -r
5.4.0-42-generic
```

附2.3 编译本实验所用内核版本

该部分过程已经在实验一中讲解，此处只是简单列出操作，编译环境安装步骤在此省略。

1. 进入 `/usr/src` 目录（建议内核代码放在此处）
2. 使用 `wget` 下载实验所需特定内核代码。内核代码地址链接：
   https://cdn.kernel.org/pub/linux/kernel/v5.x/linux-5.9.tar.xz

> 校内镜像链接： https://mirrors.ustc.edu.cn/kernel.org/linux/kernel/v5.x/linux-5.9.tar.xz

3. 使用 `tar` 指令解压。解压 `.xz` 文件的参数为 `-Jxvf` 。
4. 安装依赖库：使用包管理器安装 `flex` 和 `bison` 两个包。
5. 进入源代码根目录（ `cd /usr/src/linux-5.9` ）
6. 编译配置选择

如果直接 `sudo make menuconfig` 不修改内核编译配置，直接save，然后exit（具体配置含义见附录），会导致编译时间过长（兼容各种环境），且生成的的内核代码占用磁盘空间将近20G。为避免此情况的发生，可以将原始内核配置移动到需编译内核源码目录下，操作如下：

```
sudo cp /boot/config-`uname -r` /usr/src/linux-5.9/.config
sudo make clean
sudo make oldconfig // 出现提示一直Enter就可以了
sudo make localmodconfig // 出现提示一直Enter就可以了
```

7. 编译内核：sudo make -j $((`nproc` -1))
8. 安装内核模块： `sudo make modules_install`
9. 安装内核： `sudo make install`

执行完成后，可以检查一下 `/boot` 目录下是否生成了以下文件：

- config-5.9.0
- initrd.img-5.9.0
- System.map-5.9.0
- vmlinuz-5.9.0

若没有生成以上文件，请检查 `make` 操作后输出的相关信息并重新编译。

附2.4 配置 GRUB 引导文件

1. 修改默认的 grub 配置文件

```
sudo vim /etc/default/grub
```

2. 找到对应的两行配置文件并且修改为如下内容：

```
# GRUB_TIMEOUT_STYLE=hidden
GRUB_TIMEOUT=3
```

3. 更新内核目录项

```
sudo update-grub
```

4. 重启系统后选择 `Advanced options for Ubuntu`，并且选择 5.9 内核启动，系统启动成功后，请使用 `uname -r` 命令检查一下是否成功切换到 5.9 内核。

附录3 内核模块介绍

附3.1 什么是Linux内核模块

> 课程中我们学习到内核按照种类来划分主要分为**宏内核**、**微内核**和**混合内核**，具体定义和解释参考课程PPT。Linux 内核模块作为 Linux 内核的扩展手段，可以在运行时动态加载和卸载。

Linux是宏内核结构，所有内容都集成在一起， 效率很高，但可扩展性和可维护性较差，而模块化机制可弥补这一缺陷。所谓的模块化，就是各个部分以模块的形式进行组织，可以根据需要对指定的模块进行编译，然后安装到内核中即可。该种方式的优势是不需要预先将无用功能都编译进内核，尤其是各种驱动（通常来说，不同型号的硬件，对应的驱动不同）。如果为了顾全所有的硬件，而把所有的驱动都编译进内核，内核的体积会变得非常庞大，同时，在需要添加新硬件或者升级设备驱动时，必须重新构建内核。

可加载内核模块（Loadable Kernel Module，LKM）可以根据需要在系统运行时动态加载模块，扩充内核的功能，也可以在不需要时卸载模块。这样，对于模块的维护以及系统的使用就更加简单方便。用户可根据需要在Linux内核源码树之外开发并编译新的模块，进而修改内核功能，不必重新全部编译整个内核，只需要编译相应模块即可。同时，模块目标代码一旦被加载重定位到内核，其作用域和静态链接的代码完全等价。目前内核模块存在两种加载方式：

- 静态加载：在内核启动过程中加载，例如：**KSM模块**
- 动态加载：在内核运行的过程中随时加载。例如：各种驱动代码。

附3.2 内核模块的组成

> 基本结构：
> **头文件 -> 初始化函数 -> 清除函数 -> 引导内核的模块入口 -> 引导内核的模块出口 -> 模块许可证 -> 编译安装**

附3.2.1 需要引用的基础头文件

> 注：因为内核编程和用户层编程所用的库函数不一样，故头文件也不同。
>
> - 内核头文件的位置：`/usr/src/linux-5.9/include/`。引用方法是 `#include <linux/xxx.h>`。
> - 用户层头文件的位置：`/usr/include/`。引用方法是 `#include <xxx.h>`。

编写内核代码所用到的头文件包含在内核代码树的 `include/` 及其子目录中。`module.h`，`kernel.h`，`init.h`,这三个头文件全部包含在 `/include/linux/` 中。这三个头文件以预处理指令的形式写在模块源代码的首部：

```
// 必备头函数
#include <linux/module.h>
#include <linux/init.h>
#include <linux/kernel.h>
```

在编译模块源文件之前，由预处理程序对预处理指令进行预处理，然后由这些头文件的内容和其他部分一起组成一个完整的，可以用来编译的最后的源程序，然后由编译程序对该源程序正式进行编译，才得到目标程序。内核模块代码编译后得到目标文件后缀为.ko。

头文件`module.h`包含了对模块的结构定义以及模块的版本控制，可装载模块需要的大量符号和函数定义。

头文件`init.h`包含了两个非常重要的宏`__init`和`__exit`。

- 宏`__init`用于将一些函数标记为“初始化”函数。内核可以将此作为一个提示，即该函数仅在初始化阶段使用，并在初始化阶段之后释放使用的内存资源。**模块被装载之后**，模块装载器就会将初始化函数扔掉，这样可将该函数占用的资源释放出来。其使用方法详见2.2.2.
- 宏`__exit`的用法和`__init`一样，它的作用是标记该段代码仅用于模块卸载（编译器将把该函数放在特殊的ELF段中）。即被标记为`__exit`的函数只能在模块被卸载时调用。其使用方法详见2.2.3.

头文件`kernel.h`包含了内核常用的API，如，`printk()`就在`kernel.h`中声明。

附3.2.2 初始化函数

初始化函数的定义如下：

```
static int __init name_function(void)
{
    /* 模块要实现的功能 */
    return 0;
}
module_init(name_function);
```

模块功能函数是在模块被装入内核后调用的，也就是在模块的代码被装入内核内存后，才调用模块功能函数。

> **注意：** `__init`标记只是一个可选项，并不是写所有模块代码都要加`__init`。但是在测试我们自己写的模块时，最好加上`__init`。因为我们在写一个模块功能函数的时候，可能这个函数里面有定义的变量，当调用这个函数的时候，就要为变量分配内存空间，但注意，此时分配给变量的内存，是在内核空间分配的，也就是说分配的是内核内存。所以说如果只是想要测试一下模块的功能，并不需要让模块常驻内核内存，那就应该在执行完函数后，将当初分配给变量的内存释放。为了达到这个效果，只需要把这个函数标记为`__init`属性。

附3.2.3 清除函数

清除函数的定义如下：

```
static void __exit name_function(void)
{
    /* 这里是清除代码*/
}
module_exit(name_function);
```

`__exit`标记该段代码仅用于模块卸载，被标记为`__exit`的函数只能在模块被卸载或者系统关闭时调用。如果一个模块未定义清除函数，则内核不允许卸载该模块。

附3.2.4 初始化入口点

源码定义如下：

```
/**
 * module_init() - driver initialization entry point
 * @x: function to be run at kernel boot time or module insertion
 *
 * module_init() will either be called during do_initcalls (if
 * builtin) or at module insertion time (if a module).  There can only
 * be one per module.
 */
#define module_init(x)  __initcall(x);
```

`module_init()` 是驱动程序初始化入口点。每一个普通的c程序都有一个 `main` 函数作为入口。而在内核中，则是 `module_init()` 来负责。在内核引导时运行的函数，或者在 `do_initcalls` 期间调用 `module_init()` ，或者在模块插入时(如果是模块)调用 `module_init()` 。每个模块只能有一个。

附3.2.5 初始化出口点

源码定义如下：

```
/**
 * module_exit() - driver exit entry point
 * @x: function to be run when driver is removed
 *
 * module_exit() will wrap the driver clean-up code
 * with cleanup_module() when used with rmmod when
 * the driver is a module.  If the driver is statically
 * compiled into the kernel, module_exit() has no effect.
 * There can only be one per module.
 */
 # define module_exit(x)    __exitcall(x);
```

`module_exit()` 是驱动程序出口点。当驱动程序被删除时运行的函数。当驱动程序是一个模块时，`module_exit()` 将使用 `cleanup_module()` 包装驱动程序清理代码。如果驱动程序被静态编译到内核中，则 `module_exit()` 没有作用。每个模块只能有一个。

附3.2.6 许可证

编写内核模块，需要添加模块许可证。如果没有添加模块许可证，会收到内核被污染的警告：`module license unspecified taints kernel` ，可能会导致驱动程序的一些系统调用无法使用。

```
// 该模块的LICENSE
MODULE_LICENSE("GPL");
// 该模块的作者
MODULE_AUTHOR("OS2022");
// 该模块的说明
MODULE_DESCRIPTION("test");
```

若添加模块许可证之后仍然报错，可能是因为编译时系统时间早于模块.c文件的修改时间。

附3.2.7 参数

有时在安装模块的时候需要传递一些信息给模块，可以使用以下方式传递：

```
// 需要加上该头文件
#include <linux/moduleparam.h>
module_param(name, type, param);
// name为安装以及使用时的参数名字，type为类型，param为对应的sysfs的权限
module_param_string(name, string, len, param);
// name为外部名字，string为内部名字
module_param_array(name, type, nump, param)
// nump用于存放数组项数
```

使用的方式为，在安装模块的指定对应的参数及其值即可，如 `sudo insmod xxx.ko name=xxx`

附3.2.8 如何使用内核未导出的变量

参考： **如何使用Linux内核中没有被导出的变量或函数?**
建议使用第三种方法，不需要重新编译内核，在后面的实验中，很多函数不能直接使用，如：`page_referenced`， `follow_page` 等。这里以 `follow_page` 函数为例：

```
// 声明在   /include/linux/mm.h
struct page *follow_page(struct vm_area_struct *vma, unsigned long address,
             unsigned int foll_flags);
```

使用方法：

```
// 全局为follow_page定义别名
typedef typeof(follow_page)* my_follow_page;

// 全局定义导出的mfollow_page
// 在命令行中使用 sudo cat /proc/kallsyms | grep follow_page 以获取follow_page的内核地址
// 这里获得的地址为ffffffffa34739a0，但不同机器的地址不同，内核重启地址可能发生变化。
my_follow_page mfollow_page = (my_follow_page)0xffffffffa34739a0;
```

若输出的地址为0，是因为系统对内核地址做了保护。解决方案请参考 **链接1** 和 **链接2** 。

> 警告：每次重启系统之后，都要重新获取函数的内核地址，并在源码中修改地址，再重新编译。若不这么做，会产生段错误(Segmentation Fault)，这会导致模块无法卸载，需要重启系统。
>
> 注意：在不同的内核版本中不一定可以查询到follow_page函数的地址，Linux 3.9至4.19.241的版本只提供了follow_page_mask的地址。follow_page_mask是follow_page的具体实现，参数与follow_page略有差别，在实验代码中我们也提供了相关的调用方法。同学们在遇到无法找到follow_page函数的地址时，可以尝试使用follow_page_mask函数。

附3.3 编译、加载内核模块

在本部分中，我们将向各位演示加载一个模块的全部流程。作为示例，我们使用附件提供的 `hello_world.c` 。该模块的功能是接收一个名为 `loop` 的参数，然后向内核日志中打印 `loop` 次"hello world!"。

附3.3.1 基本知识

1. Linux内核日志

使用`printk`打印出的Linux内核日志存放在`/var/log/`中。使用`dmesg`打印日志，使用`sudo dmesg -C`清空日志。

> 提示：
>
> 1. 每条内核日志的左侧记录有日志的记录时间。
> 2. 内核日志是行缓冲的。若打印的输出末尾没有\n，有可能打印不出来。 **参考链接**
> 3. 若执行`dmesg`之后系统卡死，可能是因为写出死循环，导致内核日志过长，输出时内存爆满，建议重启。

2. 模块相关指令

指令	含义
`insmod [模块文件][参数名称=参数值]`	加载模块
`lsmod`	显示已加载的模块
`modinfo 模块名`	查看模块信息
`rmmod 模块名`	卸载模块

参考资料：

- **内核日志及printk结构分析**
- **dmesg命令**

附3.3.2 编辑模块代码、编译模块

操作流程：

1. 创建一个名为mymod的目录，并进入该目录。
2. 编写模块代码与Makefile。作为示例，这里使用我们提供的`helloworld.c`和`Makefile`。
3. 使用`make`编译（参考Lab1的2.3.17）。编译出的模块二进制文件名默认扩展名为`.ko`。在示例中，文件名是`helloworld.ko`。

> 注：
>
> 1. **linux内核模块签名**
> 2. 本次实验不涉及自己编写Makefile，可以直接使用助教提供的Makefile编译自己写的模块。Makefile中各指令的含义详见Makefile文件内的注释。当需要修改要编译的模块时，只需要修改obj-m后面的值。如，若要编译`test.c`，可将其改为`obj-m += test.o`，得到的模块二进制文件为`test.ko`。
> 3. 若不使用我们给的Makefile而直接用`gcc helloworld.c`，会报诸如“无法读取modules.order”的错。

附3.3.3 加载模块

使用指令 `sudo insmod 模块二进制文件名 参数=值` 加载模块。在示例中，若想打印"hello world!"三次，加载指令应当为 `sudo insmod helloworld.ko loop=3` 。

> 提示：
>
> 1. 若 `insmod` 时报错"loading out-of-tree module taints kernel"的warning，可以忽略。
> 2. 若 `insmod` 时报错"insmod:ERROR:could not insert module xxx.ko.File existed"，说明该模块已被加载了，请先卸载模块再加载模块。**若无法卸载模块，请重启Linux。目前助教尚未获知除重启外方便地强制卸载模块的方法。**
> 3. 若双系统在 `insmod` 时报错"Operation not permitted"，可能是因为BIOS设置问题，请进入BIOS并将secure mode设置为disable。
> 4. 若模块加载后死机，可能是因为写出了死循环，建议重启。

附3.3.4 检查模块是否正常工作

1. 使用 `lsmod | grep helloworld` 检查模块 `helloworld` 是否挂载。
2. 使用 `sudo dmesg` 查看内核日志，检查"hello world!"是否在内核日志中被打印了三次。

> 若 `lsmod` 时报错kmod_module_get_holders:could not open.....: no such file or directory ,建议在家目录下创建一个与（不含 `.c` 扩展名的）源码文件名相同的文件夹，将源码文件移动到该处目录下编译运行。

附3.3.5 卸载模块

使用指令 `sudo rmmod helloworld` 卸载模块。

> 注意： 模块和终端不是绑定的，关掉一个终端不会卸载模块。如果卸载模块时卡死，即使你关掉终端，这个模块还是没有卸载成功。

附4 内核功能与文件系统

> 本部分所涉及内容已在代码框架中实现，各位无需更改。若不感兴趣，可以不了解其实现，只了解如何向sysfs内读写数据即可。

附4.1 内核线程kthread

使用下列函数（宏）需要引入 `#include <kthread.h>` 。

附4.1.1 创建并运行内核线程

```
kthread_run(threadfn, data, namefmt, ...)
```

- 参数 `threadfn` ：线程要执行的函数。其类型为 `int (*threadfn)(void *data)` ，即：该函数接受一个 `void *` 指针作为接受的参数，返回值为 `int` ；
- 参数 `data` ：线程函数接受的参数，其类型为 `void *` ；
- 参数 `namefmt` ：线程名。
- 返回值：线程对应的 `task_struct` 指针。

附4.1.2 停止内核线程

```
int kthread_stop(struct task_struct *k);
```

- 参数 `k` ：内核线程的 `task_struct` 结构体。
- 返回值：内核函数的结束状态。

该函数会设置一个标志位，内核线程可以通过调用 `kthread_should_stop()` 函数查看自己是否应当停止

附4.1.3 延时

将内核线程挂起等待，涉及等待队列、 `wait_event` 系列函数。可以参考： **等待队列**

在我们提供的代码中，为了将内核线程停止一段时间（类似于sleep），使用了 `schedule_timeout_interruptible` 函数。其用法参见： **内核线程** 。

附4.1.4 系统中与时间相关的变量（宏）

变量（宏）名	含义
`HZ`	Linux内核每隔固定周期都会发生时钟中断， HZ 代表系统在1s中发生时钟中断的次数。
`TICK_NSEC` , `TICK_NSEC`	`HZ` 的倒数，表示两次时钟中断的时间间隔（单位分别为纳秒、微秒）
`jiffies`	记录自系统启动以来发生的时钟中断总数。因为一秒内时钟中断的次数等于 `HZ` ，所以 `jiffies` 一秒内增加的值就是 `HZ` 。

Linux还提供了 `msecs_to_jiffies` 等函数实现不同时间单位的转换。使用上述变量和函数需要 `#include<linux/jiffies.h>` 。

附4.2 sysfs文件系统

附4.2.1 基本介绍

- sysfs 是 Linux 内核中一种基于内存的虚拟文件系统，用于向用户空间导出内核对象并且能对其进行读写。
- 一个**sysfs的设计原则是：一个属性文件只做一件事情**，sysfs 属性文件一般只有一个值，直接读取或者写入。
- sysfs被映射在 `/sys/` 目录下。

附4.2.2 使用sysfs

下面假定我们要在 `/sys/kernel/mm/` 下创建一个名为 `ktest` 的属性集合，该属性集下有 `pid` 和 `func` 两个属性。这样，我们可以通过读取或修改 `/sys/kernel/mm/ktest/pid` 、 `/sys/kernel/mm/ktest/func` 两个文件来控制模块。

> 提示：使用下列函数需要 `#include <linux/sysfs.h>` 和 `#include <kobject.h>` 。

1. 设置属性的读方法(show)。每一个属性文件都要设置。当我们cat 这个属性文件时，会自动调用此方法。

参数 `kobj` 和 `attr` 表示挂载点（后面会提到）和属性列表， `buf` 表示映射到sysfs的内存地址。向这个内存地址内写入字符串，就能从sysfs内读出相同的字符串。下面的示例代码表示将一个全局变量 `pid` 打印到 `buf` 内：

```
static ssize_t pid_show(struct kobject *kobj, struct kobj_attribute *attr,
                        char *buf)
{
    return sprintf(buf, "%u\n", pid);
}
```

2. 设置属性的写方法(store)。每一个属性文件都要设置。当我们echo值到这个节点时，调用此方法。

```
static ssize_t pid_store(struct kobject *kobj,
                         struct kobj_attribute *attr,
                         const char *buf, size_t count){
    // do something
}
```

3. 创建属性。每一个属性都要执行相同操作。在提供的代码里，我们使用了一个 `KPAP_ATTR` 宏简化此步骤。

```
static struct kobj_attribute pid_attr = __ATTR(_name, 0644, pid_show, pid_store)
```

4. 设置属性集

```
// 创建属性集内的属性列表。
static struct attribute* ktest_attrs[] = {
    &pid_attr.attr,
    &func_attr.attr,
    NULL,
};

// 设置属性集名称及其中的属性。
static struct attribute_group ktest_attr_group = {
    .attrs = ktest_attrs,
    .name = "ktest",
};
```

5. 创建属性集

```
// 创建属性集，如果文件已存在，会报错。
// 前一个参数可以简单理解为挂载位置，后一个参数是属性集结构体。
int sysfs_create_group(struct kobject *kobj,
                const struct attribute_group *grp)
// 用例：mm_kobj的挂载点是/sys/kernel/mm/。
sysfs_create_group(mm_kobj, &ktest_attr_group)
```

6. 删除属性集

```
// 删除属性集，并删除属性文件。
// 前一个参数可以简单理解为挂载位置，后一个参数是属性集结构体。
void sysfs_remove_group(struct kobject *kobj,
            const struct attribute_group *grp)
// 用例：mm_kobj的挂载点是/sys/kernel/mm/。
sysfs_remove_group(mm_kobj, &ktest_attr_group)
```

附4.2.3 sysfs文件系统简单示例

我们提供了 `sysfs_test.c` 作为示例文件，该示例通过sysfs文件系统控制模块输入，主要控制指标包括：模块是否开启、模块启动函数、模块扫描周期数及模块扫描时间间隔。

下面给出了示例代码的操作与输出示例：

> 在这一节的代码中，每行开头为$的，是输入的shell命令，剩下的行是shell的输出结果。

```
# 清空缓存
$ sudo dmesg -C
# 编译并加载模块
$ sudo make
$ sudo insmod sysfs_test.ko

# 查看模块是否加载成功
$ lsmod | grep sysfs_test

# 结果
sysfs_test                  16384  0

# 查看sysfs文件是否创建成功
$ ll /sys/kernel/mm/sysfs_test/

#结果
total 0
drwxr-xr-x 2 root root    0 4月  29 14:10 ./
drwxr-xr-x 8 root root    0 4月  29 14:10 ../
-rw-r--r-- 1 root root 4096 4月  29 14:10 cycle
-rw-r--r-- 1 root root 4096 4月  29 14:10 func
-rw-r--r-- 1 root root 4096 4月  29 14:10 run
-rw-r--r-- 1 root root 4096 4月  29 14:10 sleep_millisec

# 测试
# 开启print_hello
$ sudo sh -c "echo 1 > /sys/kernel/mm/sysfs_test/func"
$ sudo sh -c "echo 4 > /sys/kernel/mm/sysfs_test/cycle"
$ sudo sh -c "echo 1 > /sys/kernel/mm/sysfs_test/run"

#查看结果
$ dmesg
[52801.318328] hello world!
[52801.318342] hello world!
[52801.318343] hello world!
[52801.318344] hello world!
```

```
# 停止
$ sudo sh -c "echo 0 > /sys/kernel/mm/sysfs_test/run"

# 卸载模块并查看sysfs是否正常卸载
$ sudo rmmod sysfs_test
$ ll /sys/kernel/mm/

#结果
total 0
drwxr-xr-x  7 root root 0 May  3 05:52 ./
drwxr-xr-x 14 root root 0 May  3 05:52 ../
drwxr-xr-x  4 root root 0 May  3 05:52 hugepages/
drwxr-xr-x  2 root root 0 May  3 05:52 ksm/
drwxr-xr-x  2 root root 0 May  3 06:51 page_idle/
drwxr-xr-x  2 root root 0 May  3 06:51 swap/
drwxr-xr-x  3 root root 0 May  3 06:51 transparent_hugepage/
```