实验三 Linux内核模块

实验目的

- 学习如何替换Linux内核版本：简单的内核版本替换
- 学习如何添加Linux模块：实现一个简单的模块
- 学习如何使用procfs及sysfs
- 学习内存管理中的基本知识：遍历vma、页面冷热识别、页表遍历

实验环境

- vmware/vitural box
- OS: Ubuntu 18.04
- Linux内核版本: 5.9.0

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

- 5.7 实验发布，讲解实验
- 5.14 晚实验课，检查实验
- 5.21 晚实验课，检查实验
- 5.28 晚实验课，实验检查
- 6.4 晚实验课，实验补检查

  > 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。

实验代码提交

> 本次实验以现场检查为主，主要在于考察实验内容、过程思路及确认结果。本实验只需提交代码，**无需写实验报告**。提交代码仅用于检查实验是否为自己独立完成，不另算分。

- 按下面描述的方式组织相关文件

```
- 顶层目录（可自行命名，如EXP2）
  - EXP3.1 子目录1
    - expt.c
    - thread.c
    - Makefile
  - EXP3.2 子目录2
    - kmscan.c
    - Makefile
```

- 将上述文件压缩
  - 格式为 .7z/.rar/.zip

    - 命名格式为 学号_姓名_实验3，如果上传后需要修改，请将文件重新命名为学号_姓名_实验3_修改n (n为修改版本)，以最后修改版本为准。
        - 如PB10001000_张三_实验3.zip ，PB10001000_张三_实验3_修改n.zip
- 提交到bb系统
    - bb系统地址：https://www.bb.ustc.edu.cn/
    - 上传至作业区: 第三次实验
    - 实验提交截止日期：2021-06-05 18:00

> **友情提示**:
>
> 本次实验以实验一、二为基础。一些步骤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、二。
>
> pdf上文本的复制有时候会丢失空格，有时候会增加不必要的空格，有时候还会增加不必要的回车。有些指令一行写不下分成两行了，一些同学就漏了第二行。如果出了bug，建议各位先仔细确认自己输入的指令是否正确。要**逐字符**比对。每次输完指令之后，请观察一下指令的输出，检查一下这个输出是不是报错。**请不要无脑复制。**
>
> 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

第一部分 编译替换内核版本

> 提示:
>
> - **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。如果想问如何生成快照，请自行百度。**
> - 经亲自测试，双系统也可通过本节所述方法替换内核版本。

1.0 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。

- 那为什么不用之前已经配置好的qemu环境?

  这是因为我们使用的qemu环境是精简的内核，很多功能没有，需要在本地环境编译打包进_install目录下，操作会很麻烦。建议本地操作，以本文正文为主。qemu主要操作见附录一。

- 其他说明

  其实，在做系统方面的研究时，更改内核版本是可能会发生的事，ADSL实验室的集群内核版本有时候也会改

1.1 查看本系统内核版本

```
uname -r
```

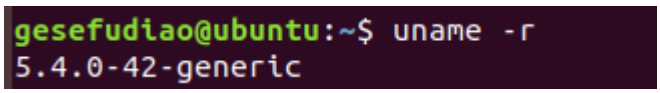


图1

1.2 编译实验所用内核版本

> 注：本过程在实验一中已讲解，此处只是简单的列出操作代码，编译环境安装此处省略

1. 进入/usr/src目录（建议内核代码放在此处）

   `cd /usr/src`

2. 下载实验所需特定内核代码

   `sudo wget 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. 解压

   `sudo tar -Jvxf linux-5.9.tar.xz`

4. 准备安装所需依赖库

   `sudo apt-get install flex bison`

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，可以尝试下面的操作：
   - 精简内核编译配置：将原始内核配置移动到需编译内核源码目录下

     > 注意：`uname -r`两端的是反引号`` ` ``，而不是引号`'`。

     ```
     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。`make -j ?`相当于分别执行`make bzImage`和`make modules`，`-j`选项后面跟的数字是jobsnum，建议设置为 CPU 核心数-1

   > 注意：`nproc`两端的是反引号`` ` ``，而不是引号`'`。这一步你可能需要等待10分钟左右。

   `sudo make -j $((`nproc`-1))`

8. 安装内核模块（这一步很快）

   `sudo make modules_install`

9. 安装内核（这一步也很快）

   `sudo make install`

> **注意**（见图2）：建议检查一下`/boot`目录是否生成了initrd镜像文件。一般通过上述步骤就会在`/boot`目录下生成满足内核启动条件的`vmlinuz`、`initrd.img`（当然`/boot`下附带还有`System.map`、`config`），如果`/boot`下没有`initrd.img`，则需要执行`cd /boot`后通过`mkinitrd -o initrd.img-xxxx`生成`initrd.img`（xxx是版本号）。

```
test@ubuntu:/usr/src$ ll /boot
total 585132
drwxr-xr-x  3 root root      4096 Apr 27 23:19 ./
drwxr-xr-x 24 root root      4096 Apr 26 22:00 ../
-rw-r--r--  1 root root    237786 Jul  9  2020 config-5.4.0-42-generic
-rw-r--r--  1 root root    122667 Apr 27 19:56 config-5.4.114
-rw-r--r--  1 root root    122667 Apr 27 19:29 config-5.4.114.old
-rw-r--r--  1 root root    246820 Apr 27 23:18 config-5.9.0
drwxr-xr-x  5 root root      4096 Apr 27 23:50 grub/
-rw-r--r--  1 root root  41243584 Apr 27 07:14 initrd.img-5.4.0-42-generic
-rw-r--r--  1 root root  11089325 Apr 27 19:56 initrd.img-5.4.114
-rw-r--r--  1 root root 481052051 Apr 27 23:19 initrd.img-5.9.0
-rw-r--r--  1 root root    182704 Jan 28  2016 memtest86+.bin
-rw-r--r--  1 root root    184380 Jan 28  2016 memtest86+.elf
-rw-r--r--  1 root root    184840 Jan 28  2016 memtest86+_multiboot.bin
-rw-------  1 root root   4573787 Jul  9  2020 System.map-5.4.0-42-generic
-rw-r--r--  1 root root   4196181 Apr 27 19:56 System.map-5.4.114
-rw-r--r--  1 root root   4196181 Apr 27 19:29 System.map-5.4.114.old
-rw-r--r--  1 root root   5297540 Apr 27 23:18 System.map-5.9.0
-rw-r--r--  1 root root   9371904 Aug  6  2020 vmlinuz-5.4.0-42-generic
-rw-------  1 root root   9445632 Apr 12 15:15 vmlinuz-5.4.0-72-generic
-rw-r--r--  1 root root   8938368 Apr 27 19:56 vmlinuz-5.4.114
-rw-r--r--  1 root root   8938368 Apr 27 19:29 vmlinuz-5.4.114.old
-rw-r--r--  1 root root   9482656 Apr 27 23:18 vmlinuz-5.9.0
```

图2

1.3 配置GRUB引导程序

> 本过程具体文件作用参见附录一，本处仅提供代码

1. 进入/usr/src/linux-5.9目录下

   `cd /usr/src/linux-5.9`

2. vim 打开/boot/grub/grub.cfg文件查找submenu关键字，查看存在哪些内核启动项(见图3)

   `sudo vim /boot/grub/grub.cfg`

3. 修改/etc/default/grub这个文件中的配置

   `sudo vim /etc/default/grub`

4. 修改GRUB_DEFAULT (见图4)，后面两个步骤主要在于防止现有内核崩溃后，系统无法选择正常内核启动（见图5)

   1. 修改GRUB_DEFAULT值
   2. 注释GRUB_TIMEOUT_STYLE
   3. 修改GRUB_TIMEOUT值

   ```
   GRUB_DEFAULT="Advanced options for Ubuntu>Ubuntu, with Linux 5.9.0"
   # GRUB_TIMEOUT_STYLE=hidden
   GRUB_TIMEOUT=3
   ```

5. 更新内核目录项

   `sudo update-grub`

> 提示：正常情况下，这一步中，系统应该能找到5.9.0版本的Linux镜像和initrd镜像。

6. 重启

   `sudo reboot`

7. 查看现有系统内核版本

   `uname -r`

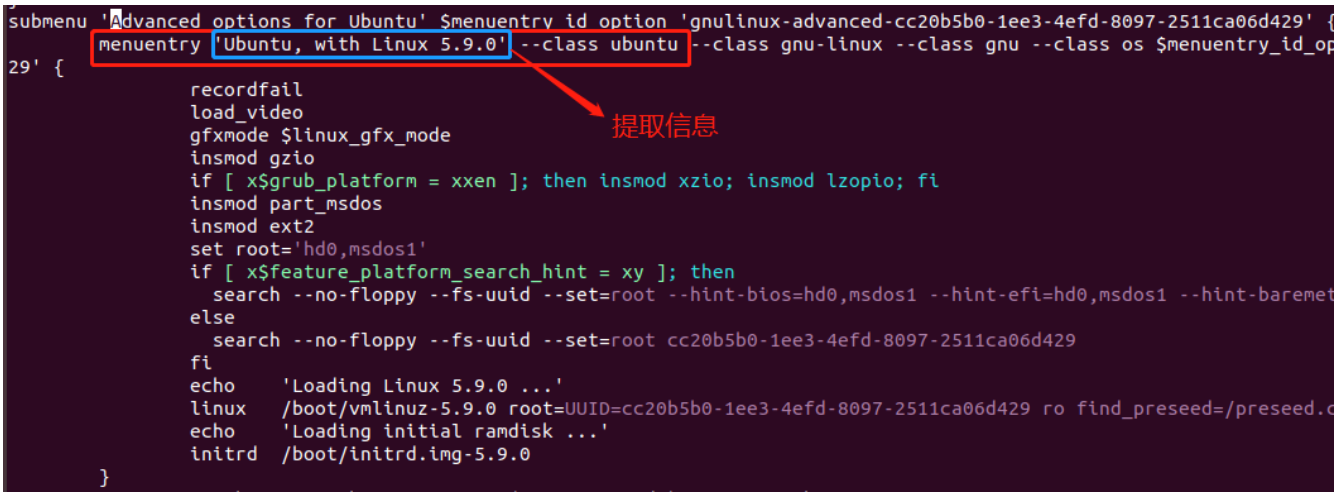


图3

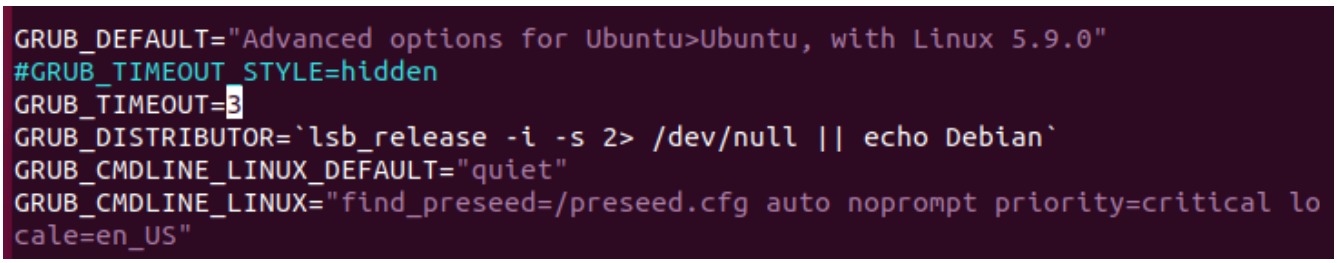


图4

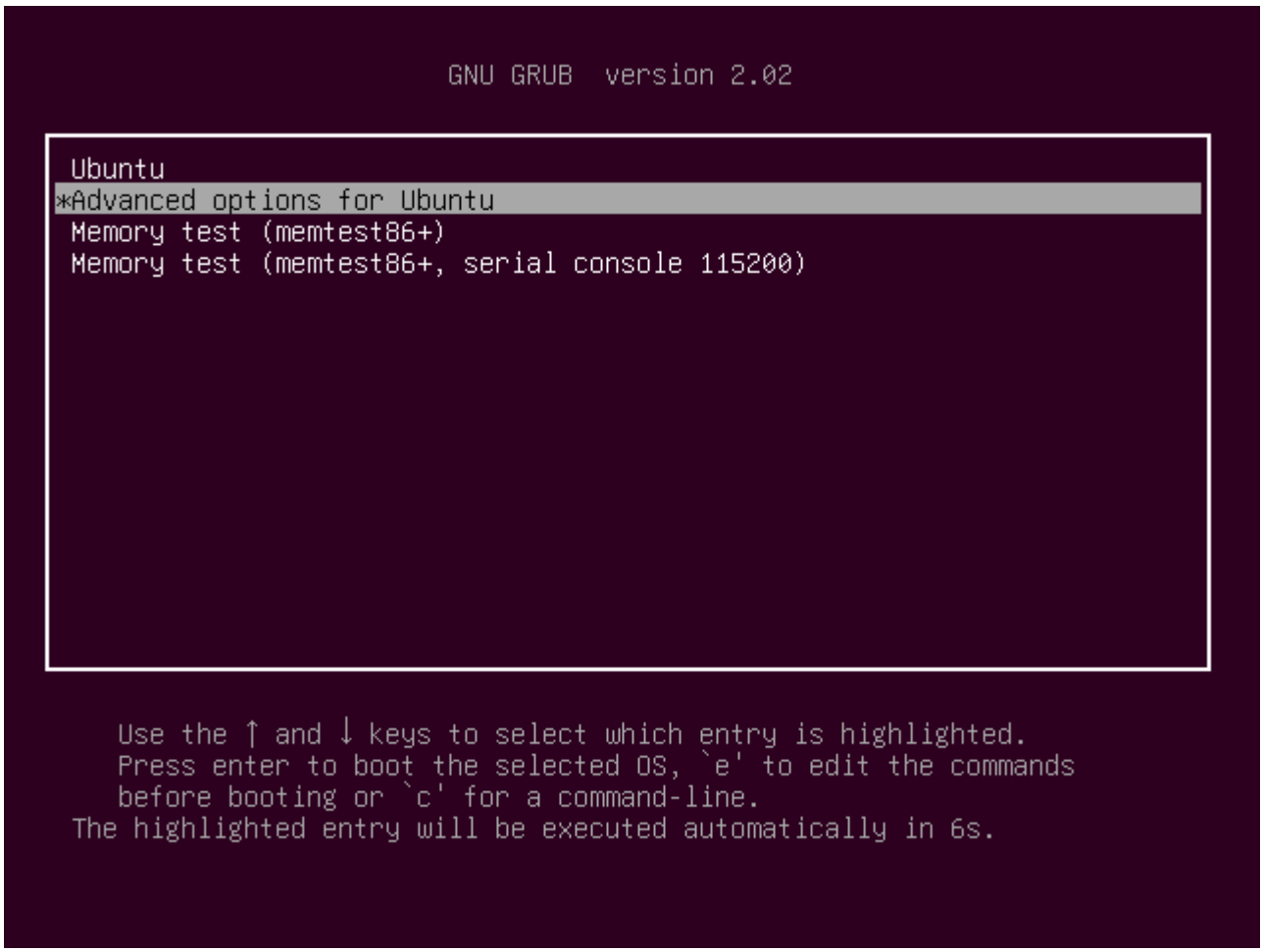


图5

1.4 任务评分规则

满分1分，包括：

- 实验检查：检查学生的linux内核版本是否为5.9。

第二部分 编译和安装内核模块

2.1 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，模块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（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。

而可加载内核模块（LKM）可以根据需要在系统运行动态加载模块，扩充内核的功能，不需要时卸载模块。这样，对应的模块的维护以及系统的使用就简单以及方便。用户可根据需要在Linux内核源码树以外来开发并编译一个模块，修改内核功能，不必重新全部编译整改内核，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。

2.2 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**

2.2.1 编写模块需要引用的基础头文件

> 注：
>
> 1. 本标题的断句方式是"编写模块/需要引用的基础头文件"。不需要自己编写头文件。
> 2. 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.o。

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`。init.h的源代码对两种宏的用法和作用给出了说明：

```
/* These macros are used to mark some functions or
 * initialized data (doesn't apply to uninitialized data)
 * as `initialization' functions. The kernel can take this
 * as hint that the function is used only during the initialization
 * phase and free up used memory resources after
 *
 * Usage:
 * For functions:
 *
 * You should add __init immediately before the function name, like:
 *
 * static void __init initme(int x, int y)    //放在函数返回值类型和函数名之间
 * {
 *    extern int z; z = x * y;
 * }
```

```
 *
*/
```

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。

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。

`kernel.h`包含了内核常用的API，比如`printk()`在`kernel.h`中的声明如下：

```
int printk(const char * fmt, ...)
```

2.2.2 模块的初始化函数

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。

2.2.3 模块的清除函数

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。

2.2.4 模块的初始化入口点

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()`——驱动程序初始化入口点。在普通的c开发中，每个程序都有一个`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。

2.2.5模块的初始化出口点

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()`--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。

2.2.6 模块的许可证

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`

内核被污染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("OS2021");
// 该模块的说明
MODULE_DESCRIPTION("test");
```

2.2.6 模块的编译配置

Make是最常用的构建工具，主要用于C语言的项目。但是实际上，任何只要某个文件有变化，就要重新构建的项目，都可以用Make构建。Make的构造规则写在一个叫做Makefile的文件中。

> 如果想系统学习Makefile编写，请移步https://www.w3cschool.cn/mexvtg/

模块的Makefile文件示例：

```
# Makefile
# 内核源代码所在位置。注意：uname -r两侧的是反引号。
KERNEL_DIR = /lib/modules/`uname -r`/build

obj-m += xxx.o

all:
    make -C $(KERNEL_DIR) M=$(PWD) modules

clean:
    rm *.o *.mod.c *.ko *.order *.symvers
```

模块的编译有两种形式，一种是编译成模块，即上面的`obj-m`，另一中是直接编译到内核文件中，此时需要将上面的`obj-m`更改为`obj-y`。

需要注意的是，由于我们是在内核源代码之外编译该模块，所以在编译的时候，需要暂时将编译目录切换到内核源代码中，即上面的`-C $(KERNEL_DIR)`，在Makefile中，可以声明变量，即上面的`KERNEL_DIR = /lib/modules/`uname -r`/build`，使用时，直接`$(KERNEL_DIR)`即可，这里的`$(PWD)`是内核自带变量，所以无需声明，可以直接使用。

编写完之后，直接执行`make`即可完成模块的编译，可以看到目录下生成一个.ko文件，这就是对应的模块了。提示：你们可以直接使用助教提供的Makefile。当需要修改要编译的模块时，只需要修改obj-m后面的名称即可。

2.2.7 模块的参数

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：

```
// 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`

2.2.8 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：

> // linux kernel version : 5.9.0
>
> // 声明 在 /include/linux/mm.h
>
> 函数原型：
>
> ```
> struct page *follow_page(struct vm_area_struct *vma, unsigned long address,
>              unsigned int foll_flags);
> ```

2.3 示例：添加一个内核模块

在本部分中，我们将向各位演示添加一个模块的全部流程。

2.3.1 基本知识

在学习添加内核模块之前我们需要先简单了解一些相关内核知识，具体是Linux内核日志，内核基本输出函数`printk`和`dmesg`命令。Linux系统拥有非常灵活和强大的日志功能，可以保存几乎所有的操作记录，并可以从中检索出我们需要的信息。

> 小知识点：
>
> - 日志默认存放位置：`/var/log/`
> - 日志配置情况：`more /etc/rsyslog.conf`
> - 内核日志及printk结构分析
> - dmesg命令

2.3.2 一个简单的内核模块代码示例

首先，在你的Linux环境下编写一个简单的内核模块代码，该模块实现的是打印若干行“hello world!”，我们在这里命名其为`hello_world.c`,并编写`Makefile`文件。

一个简单的`helloworld.c`代码：实现打印hello world!，共打印n行

> 代码参见附件helloworld.c。

简单的Makefile示例

> 参见附件Makefile。

> 注：
>
> 1. linux内核模块签名
> 2. 你们可以直接使用助教提供的Makefile。当需要修改要编译的模块时，只需要修改obj-m后面的值即可。

2.3.3 模块的安装

直接使用`insmod hello_module.ko`即可。

安装完成之后，可以使用dmesg查看是否有对应的内容输出，如果操作没有问题，则会看到这样的日志：`KERN_INFO init the module.`

> 小知识：一些与内核模块相关的操作:
>
> - lsmod | grep xxxx　**查看模块是否已经安装**
> - modinfo xxxx　**查看模块信息**
> - rmmod xxxx　**卸载模块**

2.3.4 总结：模块的编写、运行、测试

1. 创建mymod目录

   `mkdir mymod`

2. 进入mymod目录

   `cd mymod`

3. 示例:实现一个基本的内核模块，基本功能为打印loop次数的hello world。

   > 提示：该文件的内容为2.3.2所述的`helloworld.c`。如果你不会用vim，可以使用gedit。

   `vim helloworld.c`

4. 打开并编写Makefile

   > 提示：该文件的内容为2.3.2所述的`Makefile`。如果你不会用vim，可以使用gedit。

   `vim Makefile`

5. 编译内核模块(见图xx)，生成xxx.ko

   `make clean`

   `make`

6. 不在shell终端显示信息，清除ring buffer中的内容。

   `sudo dmesg -C`

7. 加载内核模块

   `sudo insmod helloworld.ko loop=3`

8. 查看内核模块是否加载成功

   `lsmod | grep helloworld`

9. 查看内核模块运行信息（见图xxx）

   `sudo dmesg`

10. 卸载模块

    `sudo rmmod helloworld`

```
-rw-rw-r-- 1 gesefudiao gesefudiao  770 Apr 29 04:32 helloworld.c
-rw-r--r-- 1 root       root       4.8K Nov 10 00:55 helloworld.ko
```

```
test@ubuntu:~/mymod$ dmesg
[170358.931055]  module init!
[170358.931058]  hello world!
[170358.931059]  hello world!
[170358.931059]  hello world!
```

2.4 实验要求

在Linux5.9下设计一个带参数的多功能内核模块，参数`func`决定开启哪个功能。功能要求如下：

2.4.1 func = 1（任务一）

1. 目标：当传入参数func = 1时，列出系统中所有内核线程的程序名、PID号和进程状态，参考下述命令的输出：

```
# ps -aux | awk '{if(NR==1 || $11 ~ /\[.*\]/) print $2"\t\t"$8"\t"$11}'

PID             STAT    COMMAND
2               S       [kthreadd]
3               I       [kworker/0:0]
4               I<      [kworker/0:0H]
6               I       [kworker/u96:0]
7               I<      [mm_percpu_wq]
8               S       [ksoftirqd/0]
9               I       [rcu_sched]
10              I       [rcu_bh]
11              S       [migration/0]
12              S       [watchdog/0]
13              S       [cpuhp/0]
14              S       [cpuhp/1]
15              S       [watchdog/1]
16              S       [migration/1]
```

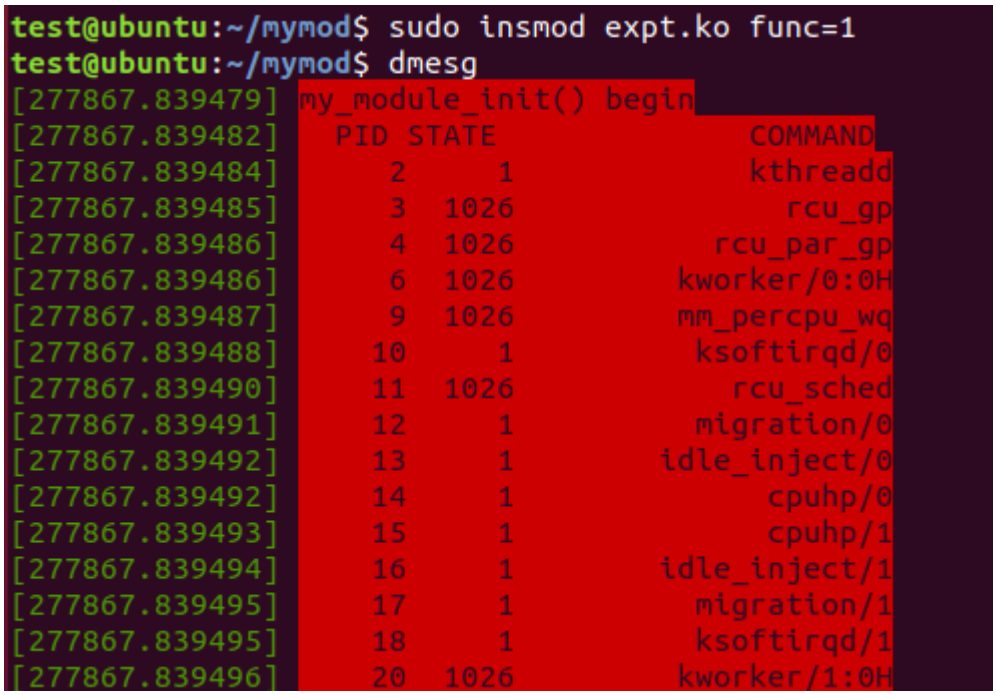
> 注：
>
> 1. ps展示的内核进程COMMAND都是带方括号的。
> 2. 程序所列状态可以是数字，并在实验文档中说明各数字代表的进程状态含义，若有能力，可在程序中转化。

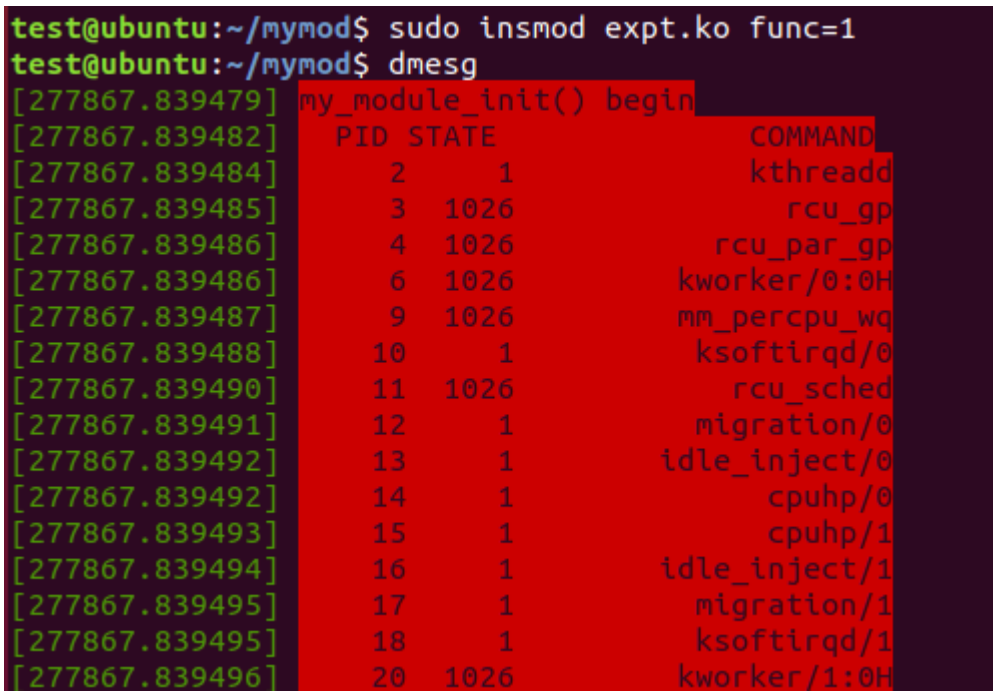
2. 输出的信息必须包括：
   - 进程的PID
   - 进程的状态
   - 进程的COMMAND（进程名）
3. 任务提示
   - 内核程序与用户程序的区别？(如何从结构体内部成员的角度出发判断某个进程为内核线程)
   - `for_each_process`获取到的`task_struct`结构体保存有对应进程的很多信息，如PID、启动时间、运行时间、进程名等(kernel version: 5.9.0)
     - `for_each_process`宏定义在`include/linux/sched/signal.h`中。
     - `task_struct`结构体定义在`include/linux/sched.h`中，为了使用它的成员变量，你可能要去参考一下。
     - 可能会用到的`task_struct`结构体成员：

成员变量	含义
`struct mm_struct *mm`	进程的内存描述符
`char comm[TASK_COMM_LEN]`	进程名
`pid_t pid`	进程的PID
`volatile long state`	进程的状态指标

4. 示例

```
sudo insmod expt.ko func=1
```





2.4.2 func = 2（任务二）

每隔五秒统计系统所有内核进程的个数。

1. 目标：当传入参数func = 2时，列出系统中所有内核进程的个数，参考下述命令的输出：

   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。

```
$ ps -aux | awk '{if(NR==1 || $11 ~ /\[.*\]/) print $2"\t\t"$8"\t"$11}' | wc -l
205
```

2. 任务提示：

- 内核线程和用户线程的区别？（见任务一）
- 重要名词
  - HZ：系统定时器频率HZ用来定义系统定时器每隔1秒产生多少个时钟中断。
  - Tick：HZ的倒数，系统定时器两次时钟中断的时间间隔。
  - Xtime：记录Wall time值，也就是UTC时间，是一个`struct timeval`结构，在用户空间通过`gettimeofday`读取。
  - Jiffies：全局变量`jiffies`用来记录自系统启动以来产生的节拍的总数，定义为`unsigned long volatile __jiffy_data jiffies;`
  - RTC：实时时钟，是一个硬件时钟，用来持久存放系统时间。

> 注：全局变量`jiffies`用来记录自系统启动以来产生的节拍的总数。启动时，内核将该变量初始化为0，此后，每次时钟中断处理程序都会增加该变量的值。一秒内时钟中断的次数等于Hz，所以jiffies一秒内增加的值也就是Hz。

- 延迟函数
  - `ndelay(unsigned long nsecs)`
  - `udelay(unsigned long usecs)`
  - `mdelay(unsigned long msecs)`
  - `msleep()`是让当前进程休眠，让出CPU给其它进程使用，等到时间到了之后再唤醒。由此看来，`msleep()`不能用于中断上下文中。
  - `mdelay()`是一个让CPU空转，一直等待到给定的时间后才退出。

> 一般而言内核建议使用msleep。

- Linux Kernel提供了内核定时器机制，其核心是由硬件产生中断来追踪时间的流逝情况。
  - 基础结构：**struct timer_list**（对应一个定时器）
  - 头文件目录：`/include/linux/timer.h` (kernel version: 5.9.0)
  - Jiffies/HZ：系统开机到现在系统运行了多少秒

使用定时器的基本流程

1. 定义一个定时器，可以把它放在你的设备结构中

   `struct timer_list my_timer_list;`

2. 初始化一个定时器

   `init_timer(&my_timer_list);`

3. 为定时器设定其服务函数

`my_timer_list.function = timer_function;`

4. 设定定时器1s后运行服务程序

   `my_timer_list.expire = jiffies + HZ;`

5. 添加定时器

   `add_timer(&my_timer_list);`

6. 写定时器服务函数

```
void timer_function(struct timer_list* t)
{
    // your code
}
```

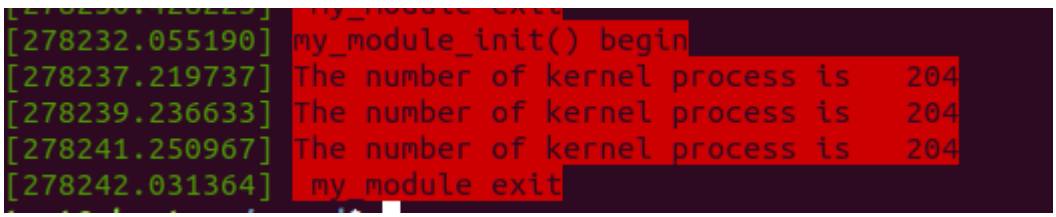
7. 当定时器不再需要时删除定时器

```
del_timer(&my_timer_list);          //当定时器不再需要时删除定时器
del_timer_sync(&my_timer_list);     //基本和del_timer一样，比较适合在多核处理器使
                                    //用，一般推荐使用del_timer_sync
```

> 注意:
> 在不同内核版本中，各个步骤对应的函数可能不一定存在，但基本工作流程如上所述。在内核版本5.9.0中，`/include/linux/timer.h`提供`timer_setup`函数合并第2、3步,同时需注意函数`timer_function`的参数类型。

3. 示例输出：

```
$ sudo insmod expt.ko func=2
```



2.4.3 func = 3 pid=xx（任务三）

传入第二参数pid，参数为某个进程的PID号，功能是列出该进程家族的家族信息，包括父进程、兄弟进程和子进程及线程的程序名、PID号。

1. 一个能产生子进程的测试程序

   测试程序thread.c

   > 代码参见附件thread.c

   结合之前的进程、子进程、线程的知识可组合不同的可能性，给出自己的解答并实验验证。

   > 注：
   >
   > 1. 优秀分：（1）如何实现顺序输出 0 ... 29 -- 0.5分 (2) 子进程开启多线程，内核模块打印相关信息 --0.5分

> 2. 实验检查可能会提问多进程、多线程组合相关知识（与第二次作业相关）。

2. 如何检查自己的输出是否正确

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。

```
# 编译
$ gcc thread.c -o thread.o -pthread
# 后台运行一次
$ ./thread.o > /dev/null &
# 结果（具体进程编号可能有差异）
[2] 24596
# 运行
$ ./thread.o
# 若干输出，且占据终端
```

另一个终端中：

```
# 在另一个终端里查找进程的pid（24604 和 24596 互为兄弟进程，具体进程编号可能有差异）
# 思考：这样使用ps真的是显示兄弟进程吗?

$ ps aux | grep ./thread.o | grep -v grep
test      24596  0.0  0.0  96828   800 pts/1    Sl   07:26   0:00 ./thread.o
test      24604  0.0  0.0  96828   832 pts/1    Sl+  07:26   0:00 ./thread.o

# 用树状图显示pid所拥有的进程（具体进程编号可能有差异，编号请看上面的输出）
# 24604 子进程：24605；24604 线程组：24604、24606、24607、24608
$ pstree -p 24604
thread.o(24604)─┬─echo(24605)
                ├─{thread.o}(24606)
                ├─{thread.o}(24607)
                └─{thread.o}(24608)
```

> pstree的基本用法:
>
> 功能 : 以树状图的方式展现进程之间的派生关系。

参数	含义
-a	显示每个程序的完整指令，包含路径，参数或是常驻服务的标示。
-c	不使用精简标示法。
-G	使用VT100终端机的列绘图字符。
-h	列出树状图时，特别标明现在执行的程序。
-H <程序识别码>	此参数的效果和指定"-h"参数类似，但特别标明指定的程序。
-l	采用长列格式显示树状图。
-n	用程序识别码排序。预设是以程序名称来排序。
-p	显示程序识别码。
-u	显示用户名称。
-U	使用UTF-8列绘图字符。
-V	显示版本信息。

2、输出的信息必须包括：

- 进程的PID
- 进程的状态
- 进程的COMMAND（进程名）

> 输出格式不需要完全照搬ps和pstree。后面给出了一个输出示例。

3、任务提示

- 了解内核链表的实现，使用方式与我们常有的链表用法有何不同(提问)。内核通用链表中定义了一个`list_head`，以及定义了一些相关的函数，如`list_for_each`， `list_entry`，`find_get_pid`，`pid_task`。(kernel: 5.9.0)
  - `find_get_pid()`: 获得对应的pid结构指针，能够成功获取对应进程号pid的进程描述符信息
  - `pid_task()`: 获取在指定的pid命名空间的进程描述符task_struct;
  - `list_for_each(pos, head)`: 宏定义，依次遍历某个链表
  - `list_entry(ptr, type, member)`: 通过已知的指向member子项的指针，获得整个结构体的指针
- 遍历某个进程的线程，需要用到宏定义`while_each_thread`，参见`include/linux/sched/signal.h`。

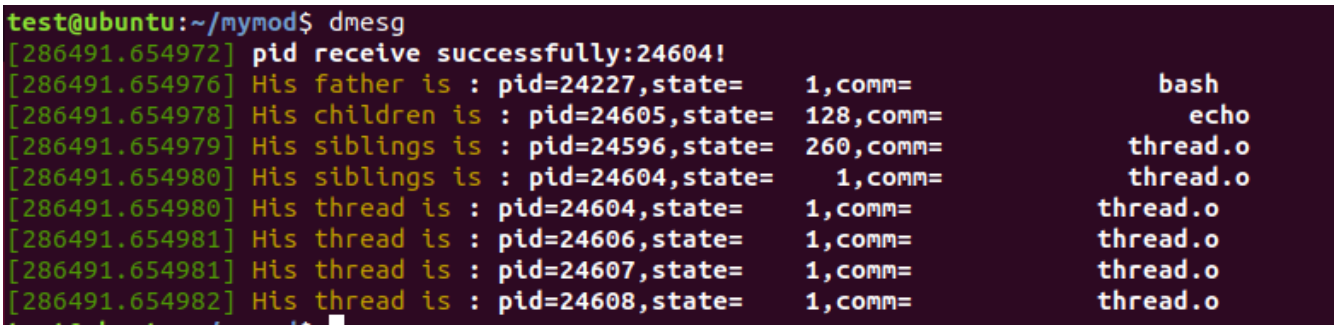
  `task_struct`结构体定义在“ include/linux/sched.h ”中，为了使用它的成员变量，你可能要去参考一下。可能会到的结构体成员：

成员变量	含义
`char comm[TASK_COMM_LEN]`	进程名
`pid_t pid`	进程的PID
`volatile long state`	进程的状态指标
`struct task_struct* parent`	进程的父进程，进程退出时，发送给`parent`进程
`struct list_head children`	所有子进程列表。实际上，children是该链表的表头。
`struct list_head sibling`	链接在父进程的`children`链表成员也链接在`sibling`成员上。

> 检查要求：
>
> 1. 了解内核通用链表实现的基本原理及掌握list.h的基本函数
> 2. 内核中的命名空间是什么？为何需要命名空间？

4、示例

```
sudo insmod expt.ko func=3 pid=24604
```



2.5 任务评分规则

满分5分，包括：

- 第一部分(func = 1)实验1分。
- 第二部分(func = 2)实验2分。
- 第三部分(func = 3 pid = xxx)实验2分.
- 总共可得5分。

> 注：实验检查的流程为先确认运行结果，再简单解释一下实现思路。

第三部分 实现一个自动化模块

本次实验在于帮助同学了解进程、内存基础知识，并使用sysfs文件系统控制内核模块，procfs文件系统导出内核模块信息，实现内核模块功能自动化。模块功能主要涉及进程vma管理、内存管理中的页面冷热识别及进程页表的遍历。

> 注意：
>
> - 本部分的主要目的是锻炼大家如何**使用** procfs、sysfs文件系统，本部分涉及上述实验部分内容。

> - 本部分主要是通过代码示例，然后学生通过自己的理解完成任务要求。

3.1 procfs文件系统

> 建议大家在自己的Linux系统中尝试运行本次实验。

3.1.1 基本介绍

- procfs是Linux内核信息的抽象文件接口，大量内核中的信息以及可调参数都被作为常规文件映射到一个目录树中，我们就可以简单直接的通过`echo`或`cat`这样的文件操作命令对系统信息进行查取和调整。
- procfs作为一个伪文件系统(pseudo-filesystem)，在linux系统中挂载在`/proc`目录，该目录中的所有文件都不会消耗磁盘空间。
- 最初procfs是为了提供有关系统中进程的信息，后来内核中的很多模块也开始使用它来报告信息，或启用动态运行时配置。procfs也提供了一个**用户空间和内核空间通信的接口**，使得我们自己的内核模块或用户态程序可以通过procfs进行参数的传递。例如：可以用类似`cat /proc/meminfo`读取内核信息，也可以使用类似`echo 1 >>/proc/sys/net/ipv4/ip_forward`向内核写消息。
- 在当今的Linux系统中，大量的系统工具也通过procfs获取内核参数，例如`ps`、`lspci`等等，没有procfs，它们将不能正常工作。
- /proc目录下常见的文件介绍：

文件	用途
/proc/apm	高级电源管理（APM）版本信息及电池相关状态信息，通常由apm命令使用；
/proc/buddyinfo	用于诊断内存碎片问题的相关信息文件；
/proc/cmdline	在启动时传递至内核的相关参数信息，这些信息通常由lilo或grub等启动管理工具进行传递；
/proc/cpuinfo	处理器的相关信息的文件；
/proc/devices	系统已经加载的所有块设备和字符设备的信息，包含主设备号和设备组（与主设备号对应的设备类型）名；
/proc/diskstats	每块磁盘设备的磁盘I/O统计信息列表；
/proc/iomem	每个物理设备上的记忆体（RAM或者ROM）在系统内存中的映射信息；
/proc/kallsyms	模块管理工具用来动态链接或绑定可装载模块的符号定义，由内核输出；
/proc/kmsg	此文件用来保存由内核输出的信息，通常由/sbin/klogd或/bin/dmsg等程序使用，不要试图使用查看命令打开此文件；
/proc/modules	当前装入内核的所有模块名称列表，可以由lsmod命令使用，也可以直接查看；
/proc/xxxx(pid)/xxxx	进程相关子目录 ，系统当前运行进程的proc抽象，包含了一些文件，用于显示进程相关的信息。

3.1.2 proc文件系统常用函数介绍

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

```
# linux kernel version : 5.9.0

struct proc_dir_entry *proc_mkdir(const char *name, struct proc_dir_entry *parent);
//功能：在proc中创建一个文件夹

//参数1：创建的文件夹名称

//参数2：创建的文件夹路径，就是在哪个文件夹中创建，如果是proc根目录，此参数为NULL

//返回值：创建的文件夹路径

# 不同版本参数可能有所不同，需注意，主要是proc_ops结构体可能不同
static inline struct proc_dir_entry *proc_create(const char *name, umode_t mode, struct
proc_dir_entry *parent, const struct proc_ops *proc_ops, void *);
//功能：在proc中创建一个文件

//参数1：创建的文件的名称

//参数2：文件的读写权限

//参数3：创建的文件路径，即在哪个文件夹中创建，如果是proc根目录，此参数为NULL

//参数4：此文件的操作函数proc_ops

//返回值：创建的文件路径


void proc_remove(struct proc_dir_entry *de);
//功能：删除proc中的文件或文件夹

//参数：删除的文件或文件夹路径

//返回：无
```

3.1.3 proc文件系统简单示例

示例目的： 传入一个pid，在proc根目录下创建目录test，目录test下有一个文件pid

> 代码参见附件procfs_test.c

3.1.4 模块挂载示例

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。

```
# 插入模块
$ sudo make
$ sudo insmod procfs_test.ko pid=4

# 查看proc目录
$ ll /proc/test

#结果
total 0
```

```
dr-xr-xr-x   3 root root 0 May  3 05:53 ./
dr-xr-xr-x 404 root root 0 May  3 05:52 ../
-rw-rw-r--   1 root root 0 May  3 05:54 pid

# 查看pid内容
$ cat /proc/test/pid

#结果
4
```

3.2 sysfs文件系统

3.2.1 基本介绍

- sysfs 是 Linux 内核中一种新的虚拟的基于内存的文件系统，用于向用户空间导出内核对象并且能对其进行读写。它的作用和 proc 类似，但除了同样具有查看(cat) 和 设定(echo) 内核参数功能之外，也可以用来管理 Linux 统一设备模型。
- sysfs 比 proc 相比有很多优点，最重要的就是设计上很清晰。比如，一个**proc 虚拟文件**有可能有内部格式，`/proc/scsi/scsi`它是可读可写的，并且读写格式不一样，代表不同的操作。但是，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。

3.2.2 sysfs文件系统常用函数介绍

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

```
// include/linux/kobject.h
// 每一个attribute会对应自己的show/store函数,意味每个文件拥有自己的读、写方法
struct kobj_attribute {
    struct attribute attr;
    ssize_t (*show)(struct kobject *kobj, struct kobj_attribute *attr,
            char *buf);
    ssize_t (*store)(struct kobject *kobj, struct kobj_attribute *attr,
             const char *buf, size_t count);
};
```

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

```
//  include/linux/sysfs.h
#define __ATTR(_name, _mode, _show, _store) {               \
    .attr = {.name = __stringify(_name),                \
         .mode = VERIFY_OCTAL_PERMISSIONS(_mode) },     \
    .show   = _show,                            \
    .store  = _store,                           \
}

#define __ATTR_RO(_name) {                      \
    .attr   = { .name = __stringify(_name), .mode = 0444 },     \
    .show   = _name##_show,                     \
}

// kobj_attribute是内核提供给我们的一种更加灵活的处理attribute的方式，但是它还不够。只有当我们
```

```
// 使用kobject_create来创建kobject时，使用kobj_attribute才比较方便，但大部分情况下，我们是把
// kobject内嵌到自己的结构里，此时就无法直接使用内核提供的dynamic_kobj_ktype，因此，我们需要创
// 建自己的kobj_attribute。

#define XXXX_ATTR_RO(_name) \
        static struct kobj_attribute _name##_attr = __ATTR_RO(_name)

#define XXXX_ATTR(_name)                            \
        static struct kobj_attribute _name##_attr = \
                __ATTR(_name, 0644, _name##_show, _name##_store)


// 表示读方法,当我们cat 这个节点时，调用此方法
aaa_show()

// 表示写方法，当我们echo值到这个节点时，调用此方法。
aaa_store()

XXXX__ATTR_RO(aaa) / XXXX__ATTR(aaa)

// 我们这里只有一个属性结构体数组只有一个成员，可以有多个，比如：
static struct attribute* bbb_attrs[] = {
    // 扫描进程的扫描间隔 默认为30秒
    &aaa_attr.attr,
    &bbb_attr.attr,
    NULL,
};

// 创建一个属性集合
static struct attribute_group aaa_attr_group = {
    .attrs = aaa_attrs,
    .name = "aaa",
};

// 在kobj目录下创建一个属性集合，并显示集合中的属性文件。如果文件已存在，会报错。
int sysfs_create_group(struct kobject *kobj,
               const struct attribute_group *grp)

// 在kobj目录下删除一个属性集合，并删除集合中的属性文件
void sysfs_remove_group(struct kobject *kobj,
            const struct attribute_group *grp)

// 创建sysfs接口后，就可以在shell 终端查看到和操作接口了。当我们将数据 echo 到接口中时，在用户
// 空间完成了一次 write 操作，对应到 kernel ，调用了驱动中的”store”。当我们cat一个接口时则会调
// 用”show” 。
```

3.2.3 sysfs文件系统简单示例

示例目的： 通过sysfs文件系统控制模块输入，此示例主要控制指标：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

示例重要知识点： sysfs文件系统、内核线程

> 代码参见附件sysfs_test.c

3.2.4 模块挂载示例

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 May  3 20:27 ./
drwxr-xr-x 8 root root    0 May  3 05:52 ../
-rw-r--r-- 1 root root 4096 May  3 20:27 cycle
-rw-r--r-- 1 root root 4096 May  3 20:27 func
-rw-r--r-- 1 root root 4096 May  3 20:27 run
-rw-r--r-- 1 root root 4096 May  3 20:27 sleep_millisecs

# 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!

# 换成print_hi
$ sudo dmesg -C
$ sudo sh -c "echo 2 > /sys/kernel/mm/sysfs_test/func"
$ sudo sh -c "echo 1 > /sys/kernel/mm/sysfs_test/run"

#查看结果
$ dmesg
[52883.239254] hi world!
[52883.239257] hi world!
[52883.239258] hi world!
[52883.239259] hi 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/
```

> 注： 自己可以试试改变其他参数看看自动控制的效果。

3.3 内存管理基础知识

> 本部分目的在于帮助想要进一步理解真实系统内存管理模块的同学，希望可以为同学们带来进一步探索的兴趣。

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"
> ```

3.3.1 VMA

用户进程的虚拟地址空间包含了若干区域，这些区域的分布方式是特定于体系结构的，不过所有的方式都包含下列成分：

- 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。

由此可以看到进程的虚拟内存空间会被分成不同的若干区域，每个区域都有其相关的属性和用途，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。在linux内核中，这样的区域被称之为虚拟内存区域(virtual memory areas),简称vma。一个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：

```
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;

    pgprot_t vm_page_prot;      /* vma的访问权限 */
    unsigned long vm_flags;    /* 标识集 */

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

    /*
     * For areas with an address space and backing store,
     * linkage into the address_space->i_mmap prio tree, or
     * linkage to the list of like vmas hanging off its node, or
     * linkage of vma in the address_space->i_mmap_nonlinear list.
     */
    /* shared联合体用于和address space关联 */
    union {
        struct {
            struct list_head list;/* 用于链入非线性映射的链表 */
            void *parent;   /* aligns with prio_tree_node parent */
            struct vm_area_struct *head;
        } vm_set;

        struct raw_prio_tree_node prio_tree_node;/*线性映射则链入i_mmap优先树*/
    } shared;

    /*
     * A file's MAP_PRIVATE vma can be in both i_mmap tree and anon_vma
     * list, after a COW of one of the file pages.  A MAP_SHARED vma
     * can only be in the i_mmap tree.  An anonymous MAP_PRIVATE, stack
     * or brk vma (with NULL file) can only be in an anon_vma list.
     */
    /*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 */

    /* Function pointers to deal with this struct. */
    /*该vma上的各种标准操作函数指针集*/
    const struct vm_operations_struct *vm_ops;

    /* Information about our backing store: */
    unsigned long vm_pgoff;     /* 映射文件的偏移量，以PAGE_SIZE为单位 */
    struct file * vm_file;          /* 映射的文件，没有则为NULL */
    void * vm_private_data;     /* was vm_pte (shared mem) */
    unsigned long vm_truncate_count;/* truncate_count or restart_addr */

#ifndef CONFIG_MMU
    struct vm_region *vm_region;    /* NOMMU mapping region */
#endif
#ifdef CONFIG_NUMA
    struct mempolicy *vm_policy;    /* NUMA policy for the VMA */
```

```
#endif
};
```

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。

3.3.2 页框管理

页框管理是Linux系统的基本功能,主要负责维护RAM资源，完成系统对RAM资源请求的分配。 Linux 把 RAM 每4KB划分为一个页框，这样正好与页大小相等或为页框的整数倍，便于请求页框。
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。 虽然如此，但是实际上系统使用内存时还是倾向于分配连续的内存块，因为分配连续内存时，页表不需要更改，因此能降低TLB的刷新率。为了能够在保存连续页的同时，又能满足对启动内存需求的分配， 页框在分配上使用了**伙伴系统。**

在内核中，每一个物理页框对应一个**页描述符(struct page)**, 所有的页描述符存放在一个**mem_map**线性数组之中。 每一个页框的页框号(`address >> PAGE_SHIFT`)为页描述符在 mem_map 数组的位置(下标)。

> *关于页描述符，在`include/linux/mm.h`可以找到其定义:*

```
// 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 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 */
}
```

页面回收算法： 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 算法基于这样一个事实，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，这样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。

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，则调用该函数相应地修改 `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。

> 建议阅读文献：Linux物理内存分配

3.3.3 页表

用来将虚拟地址映射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. 该数组指向与之关联的页帧, 但这会引发一个问题, 例如, 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, 则需要包含100万项的页表. 这个问题在64位体系结构下, 情况会更加糟糕. 而每个进程都需要自身的页表, 这会导致系统中大量的所有内存都用来保存页表。为减少页表的大小并容许忽略不需要的区域, 计算机体系结构的涉及会将虚拟地址分成多个部分. 同时虚拟地址空间的大部分们区域都没有使用, 因而页没有关联到页帧, 那么就可以使用功能相同但内存用量少的多的模型: **多级页表**。

Linux中的分页：为了同时支持适用于32位和64位的系统, Linux采用了通用的分页模型. 在Linux-2.6.10版本中, Linux采用了三级分页模型. 而从2.6.11开始普遍采用了四级分页模型。目前的内核的内存管理总是指定使用四级页表, 而不管底层处理器是否如此。

单元	描述
页全局目录	Page GlobalDirectory
页上级目录	Page Upper Directory
页中间目录	Page Middle Directory
页表	Page Table
页内偏移	Page Offset

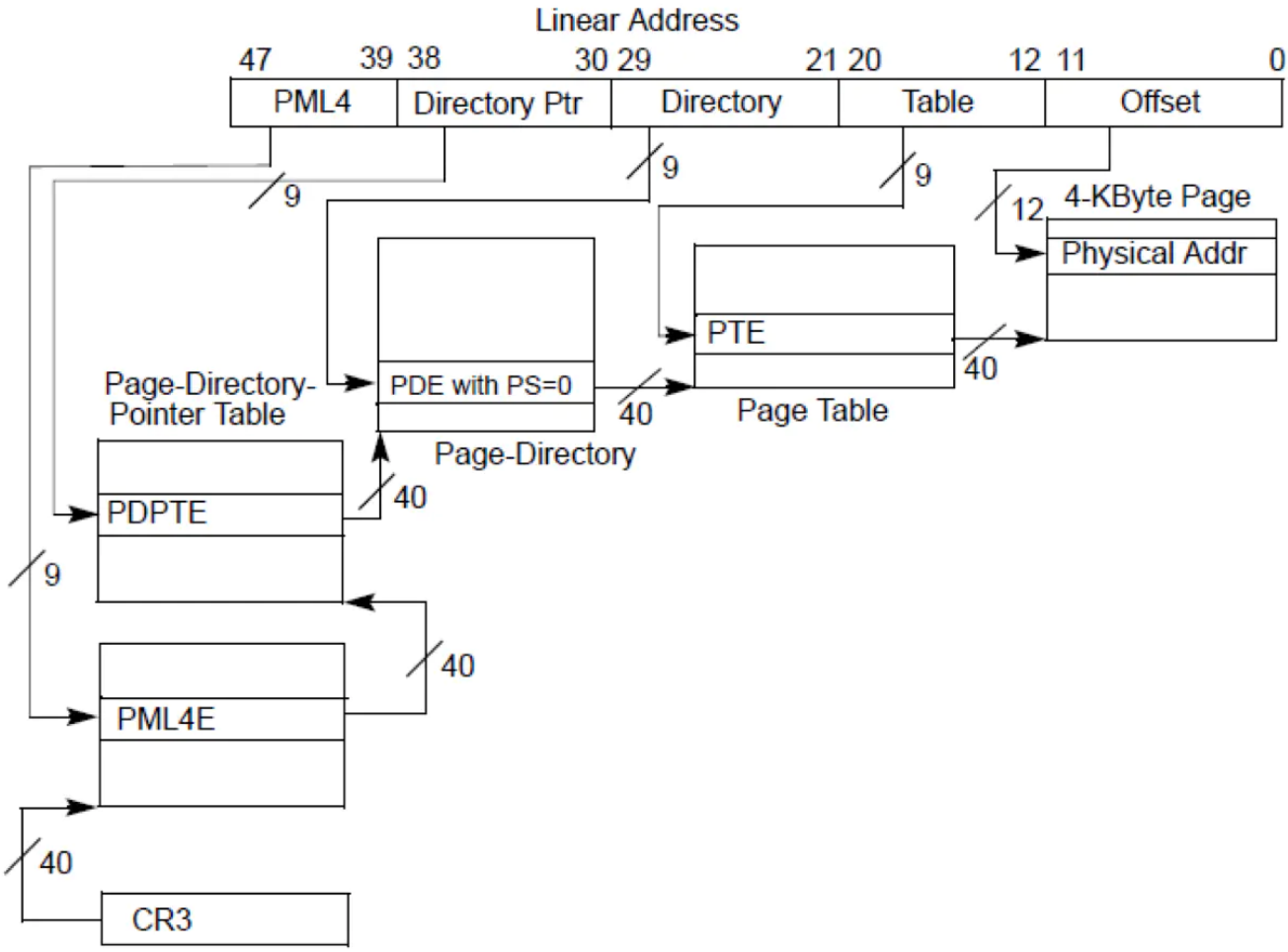
Linux不同于其他的操作系统, 它把计算机分成独立层(体系结构无关)/依赖层(体系结构相关)两个层次。对于页面的映射和管理也是如此。页表管理分为两个部分, 第一个部分依赖于体系结构, 第二个部分是体系结构无关的. 所有数据结构几乎都定义在特定体系结构的文件中。这些数据结构的定义可以在头文件`arch/对应体系/include/asm/page.h`和`arch/对应体系/include/asm/pgtable.h`中找到。但是对于AMD64和IA-32已经统一为

一个体系结构。但是在处理页表方面仍然有很多的区别, 因为相关的定义分为两个不同的文件 `arch/x86/include/asm/page_32.h` 和 `arch/x86/include/asm/page_64.h`, 类似的也有 `pgtable_xx.h`。

其他相关内容参考：

1. 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
2. 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

> 小评：其实很多设计的根源或者说原因都来自于CPU的设计, OS很多时候都是辅助CPU。Linux的四级页表就是依据CPU的四级页表来设计的。这里主要说的就是Intel x64页面大小为4KB的情况, 如图所示:



3.4 实验要求

实验的第三部分分为三小部分，分别是遍历进程VMA，页面冷热统计以及页表软件遍历。本次实验需要使用procfs、sysfs文件系统实现自动化模块kmscan，主要是指：

通过sysfs完成模块参数的输入。要求：

文件	功能
/sys/kernel/mm/kmscan/func	选择功能
/sys/kernel/mm/kmscan/sleep_millisecs	扫描时间间隔
/sys/kernel/mm/kmscan/run	是否开启模块功能
/sys/kernel/mm/kmscan/pid	选择哪个进程

通过procfs实现模块的输出。要求：

文件	功能
/proc/kmscan/xxx(pid)/file	进程xxx的文件页总数
/proc/kmscan/xxx(pid)/active_file	进程xxx的活跃文件页总数
/proc/kmscan/xxx(pid)/anon	进程xxx的匿名页总数
/proc/kmscan/xxx(pid)/active_anon	进程xxx的活跃匿名页总数
/proc/kmscan/xxx(pid)/vma_count	进程xxx的VMA数量

> 注：
>
> 0. 提示：你们需要综合我们提供的`sysfs_test.c`和`procfs_test.c`，自己编写一个能实现实验要求的模块。
> 1. 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3.3的开头部分）
> 2. 在测试时，下文的“某进程”特指使用`thread.c`(就是本文档2.4.3.1提到的那个)运行产生的进程。

3.4.1 func = 1：每隔5s统计某进程的vma数量

1. 需要的文件

- 控制： /sys/kernel/mm/kmscan/func ： 选择功能
- 输出： /proc/kmscan/xxx(pid)/vma_count： 进程xxx VMA数量

2. 任务提示

- 通过pid获取`task_struct`描述符，参考实验2.4；
- 遍历vma需要使用到`mm_struct`描述符中的mmap成员；

3.4.2 func = 2：每隔5s统计某进程的匿名页、文件页非活跃数量和总量

1. 需要的文件

- 控制： /sys/kernel/mm/kmscan/func ： 选择功能
- 输出：

文件	功能
/proc/kmscan/xxx(pid)/file	进程xxx的文件页总数
/proc/kmscan/xxx(pid)/active_file	进程xxx的活跃文件页总数
/proc/kmscan/xxx(pid)/anon	进程xxx的匿名页总数
/proc/kmscan/xxx(pid)/active_anon	进程xxx的活跃匿名页总数

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。
- 需要了解的函数：

```
// include/linux/mm.h
// 根据rmap机制获取页面结构体struct page
struct page *follow_page(struct vm_area_struct *vma, unsigned long address,
             unsigned int foll_flags);
//  判断页面是否为匿名页
static inline int PageAnon(struct page *page)
{
    return ((unsigned long)page->mapping & PAGE_MAPPING_ANON) != 0;
    /* #define PAGE_MAPPING_ANON    1  此函数非常easy，就是判断page的mapping成员的末位是不是
1，是的话返回1，不是的话返回0*/

}
// 获取page引用计数,不为0意味着活跃
extern int page_referenced(struct page *page,
            int is_locked,
            struct mem_cgroup *memcg,
            unsigned long *vm_flags)
```

> `follow_page`，`page_referenced`的使用参考2.2.8

3.4.3 软件遍历某进程的页表，并打印所有页面物理号（可printk输出到内核日志中）

本部分实验为进阶实验，可选做。

页表处理的基本函数（宏）：(这些函数（宏）的含义请参考：linux内核四级分页理解)

```
/*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))))
```

参考文献：

- Linux虚拟内存三级页表
- linux内核四级分页理解

3.5 任务评分规则

满分4分，加分2分，包括：

1. 第一任务(func = 1)完成，得2分。
2. 第二任务(func = 2)完成，得2分。
3. 加分表现：检查回答问题优秀者可加分，加分最高为1分。
4. 加分表现：第三任务(func = 3)完成，可得2分。主要根据第三次实验的表现而来，主要在于优化输出（考虑不在内核日志中输出）。
5. 本次实验加分按照第4条和第5条加分项取最高分加分。第三部分最高可得6分，不会溢出。

> 注：实验检查时会要求先确认运行结果（请**自行准备**可以验证相应功能支持的测试样例），再简单解释一下如何实现。

参考文献：

1. linux进程地址空间--vma的基本操作
2. Linux内存管理篇之页框管理

附录 在QEMU环境下进行实验

注：该部分内容描述的做法可以不需要替换实际运行环境的内核

a. 下载并编译内核

注：该步骤与实验一的步骤相同，区别只是换了内核版本

1. 下载linux-5.9.tar.xz，解压缩得到目录linux-5.9，不妨称之为Linux源代码根目录(以下简称源码根目录)

```
mkdir ~/oslab
cd ~/oslab
wget https://cdn.kernel.org/pub/linux/kernel/v5.x/linux-5.9.tar.xz
tar -Jvxf linux-5.9.tar.xz
```

2. 准备安装所需依赖库（我们已在实验一中安装，可跳过）

```
sudo apt-get install git build-essential bc libelf-dev xz-utils libssl-dev libncurses5-dev libncursesw5-dev
```

3. 进入源代码根目录，并编译配置选择

```
cd ~/oslab/linux-5.9
make menuconfig
```

4. 执行编译指令

```
make -j $((`nproc`-1)) # 此处为使用(你的CPU核心数-1)个线程进行编译，如果虚拟机分配的cpu数只有1(如Hyper-V默认只分配1核)则需先调整虚拟机分配的核心数.
```

b. 配置QEMU

该部分与实验一的步骤基本完全相同，唯一的区别是用busybox准备根文件系统时，需要修改init文件：

```
cd ~/oslab/busybox-1.32.1/_install
sudo vim init # 或 sudo gedit init
```

在实验一的基础上需要增加两行，使得init文件的内容变成如下：

```
#!/bin/sh
echo "INIT SCRIPT"
mkdir /proc
mkdir /sys
mount -t proc none /proc
mount -t sysfs none /sys
mkdir /tmp
mount -t tmpfs none /tmp
mknod -m 666 /dev/ttyS0 c 4 64      #增加的内容
setsid cttyhack sh                  #增加的内容
echo -e "\nThis boot took $(cut -d' ' -f1 /proc/uptime) seconds\n"
exec /bin/sh
```

c. 编译模块

新建一个文件夹存放自己编写的模块代码和Makefile文件：

```
cd ~/oslab
mkdir lab3mod
cd lab3mod
vim helloworld.c     # 编写自己的内核模块代码
vim Makefile         # 编写Makefile
```

其中Makefile的内容与本文档正文中描述的有所差异，只需要有如下的内容即可：(有多个模块需要编译，可以仿照着增加多个行)

```
obj-m += helloworld.o
```

然后进入到内核源码目录执行模块编译执行：

```
cd ~/oslab/linux-5.9
make modules M=~/oslab/lab3mod    # 替换为自己的模块所在目录
```

d. 重新制作根文件系统

将编译好的内核模块复制到busybox的用于制作根文件系统的目录下，重新打包镜像文件：

```
sudo cp ~/oslab/lab3mod/helloworld.ko ~/oslab/busybox-1.32.1/_install
cd ~/oslab/busybox-1.32.1/_install
find . -print0 | cpio --null -ov --format=newc | gzip -9 > ~/oslab/initramfs-busybox-
x64.cpio.gz
```

注：每次重新编译模块后都需要重新打包镜像

e. 运行QEMU并且加载模块

```
qemu-system-x86_64 -s -kernel ~/oslab/linux-4.9.263/arch/x86_64/boot/bzImage
-initrd ~/oslab/initramfs-busybox-x64.cpio.gz --append "nokaslr
root=/dev/ram init=/init console=ttyS0 " -nographic

# qemu shell
insmod helloworld.ko
rmmod helloworld.ko
```

f. 静态编译二进制可执行文件到QEMU下运行

正文中需要用到thread.c作为测试程序，以及pstree工具，可以在宿主机的环境中将相应的代码进行静态编译出二进制文件，然后通过步骤d将其打包到根文件镜像系统中

```
gcc thread.c -static -o thread.o -pthread

wget https://raw.githubusercontent.com/ZacharyLiu-CS/USTC_OS/master/term2021/lab3/pstree-
src.tar.gz
tar zxf pstree-src.tar.gz
cd pstree-src

gcc pstree.c -DVERSION="\"12\"" -lreadline -ltinfo -static -o pstree
# 将 thread.o 和 pstree 打包到根文件系统镜像文件中
```