Linux 内核编译

准备编译工具 make, gcc,

```
$ sudo apt-get install make
$ sudo apt-get install build-essential
```

在 www.kernel.org 上面，下载对应的 Linux 内核代码。

先解压 tar.xz

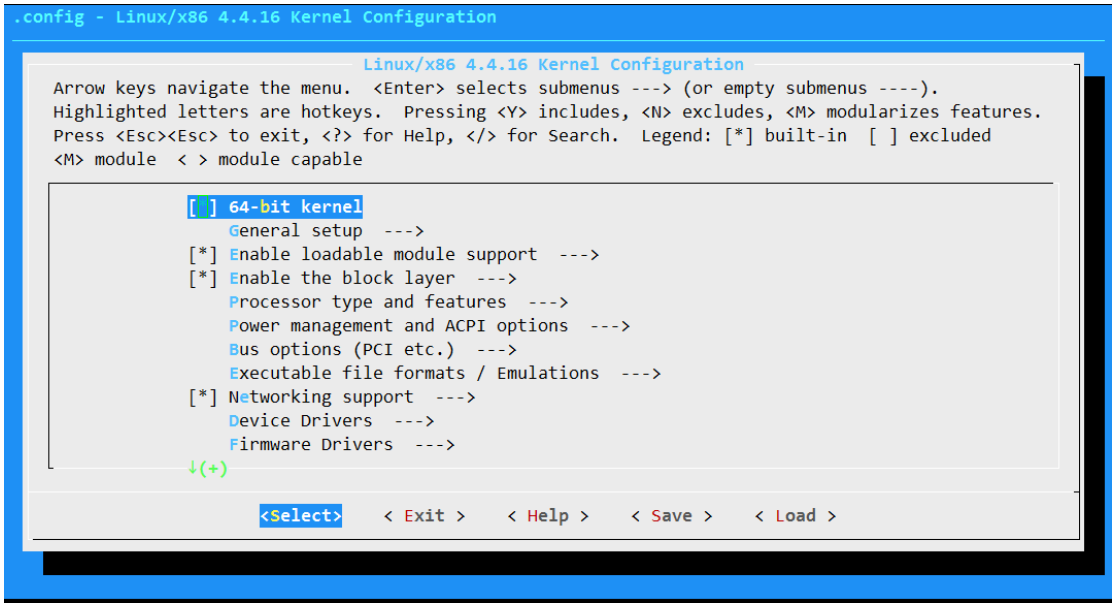
```
$ xz -d linux-4.4.16.tar.xz
$ tar xvf linux-4.4.16.tar
```

```
wangbojing@ubuntu:~/share$ tar xvf linux-4.4.16.tar
```

```
$ cp /boot/config-xxx ./.config
```

```
wangbojing@ubuntu:~/share/linux-4.4.16$ cp /boot/config-4.2.0-27-generic ./.config
```

```
$ make menuconfig
```

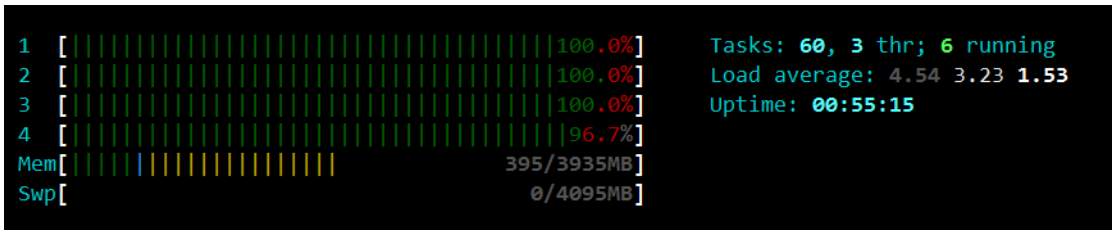


选择 save ，直接退出。

保存退出以后，可以开始编译内核。

```
$ make -j4
```

注：-j4 代表的是 4 个线程， 编译过程中 cpu 与内存参数




```
$ sudo su
```

```
wangbojing@ubuntu:~/share/linux-4.4.16$ sudo su
[sudo] password for wangbojing:
root@ubuntu:/home/wangbojing/share/linux-4.4.16#
```

```
# make modules_install
```

```
wangbojing@ubuntu:~/share/linux-4.4.16$ ls
arch     crypto         include  kernel       modules.builtin  REPORTING-BUGS  System.map  vmlinux-gdb.py
block    Documentation  init     lib          modules.order    samples         tools       vmlinux.o
certs    drivers        ipc      MAINTAINERS  Module.symvers   scripts         usr
COPYING  firmware       Kbuild   Makefile     net              security        virt
CREDITS  fs             Kconfig  mm           README           sound           vmlinux
```

```
root@ubuntu:/home/wangbojing/share/linux-4.4.16# ls /lib/modules/
4.2.0-27-generic  4.4.16
root@ubuntu:/home/wangbojing/share/linux-4.4.16#
```

```
# make bzImage
```

```
root@ubuntu:/home/wangbojing/share/linux-4.4.16# ls arch/x86/boot/
a20.c        compressed  ctype.h                memory.c       regs.c     version.c     video-vesa.o
a20.o        copy.o      early_serial_console.c memory.o       regs.o     version.o     video-vga.c
apm.c        copy.S      early_serial_console.o mkcpustr       setup.bin  vesa.h        video-vga.o
bioscall.o   cpu.c       edd.c                  mkcpustr.c     setup.elf  video-bios.c  vmlinux.bin
bioscall.S   cpucheck.c  edd.o                  mtools.conf.in setup.ld   video-bios.o  voffset.h
bitops.h     cpucheck.o  header.o               pm.c           string.c   video.c       zoffset.h
boot.h       cpuflags.c  header.S               pmjump.o       string.h   video.h
bzImage      cpuflags.h  install.sh             pmjump.S       string.o   video-mode.c
cmdline.c    cpuflags.o  main.c                 pm.o           tools      video-mode.o
cmdline.o    cpu.o       main.o                 printf.c       tty.c      video.o
code16gcc.h  cpustr.h    Makefile               printf.o       tty.o      video-vesa.c
```

安装新编译的内核

```
$ cp arch/x86/boot/bzImage /boot/vmlinuz-4.4.16
$ cp .config /boot/config-4.4.16
$ cd /lib/modules/4.4.16/
$ update-initramfs –c –k 4.4.16
$ update-grub
```

错误解决方案：

```
In file included from scripts/kconfig/mconf.c:23:0:
scripts/kconfig/lxdialog/dialog.h:38:20: fatal error: curses.h: No such file or directory
 #include CURSES_LOC
                    ^
```

安装 ncurses 字符终端处理库， 不然在 make menuconfig 的时候，会提示报错。

```
$ sudo apt-get install libncurses5-dev libncursesw5-dev
```

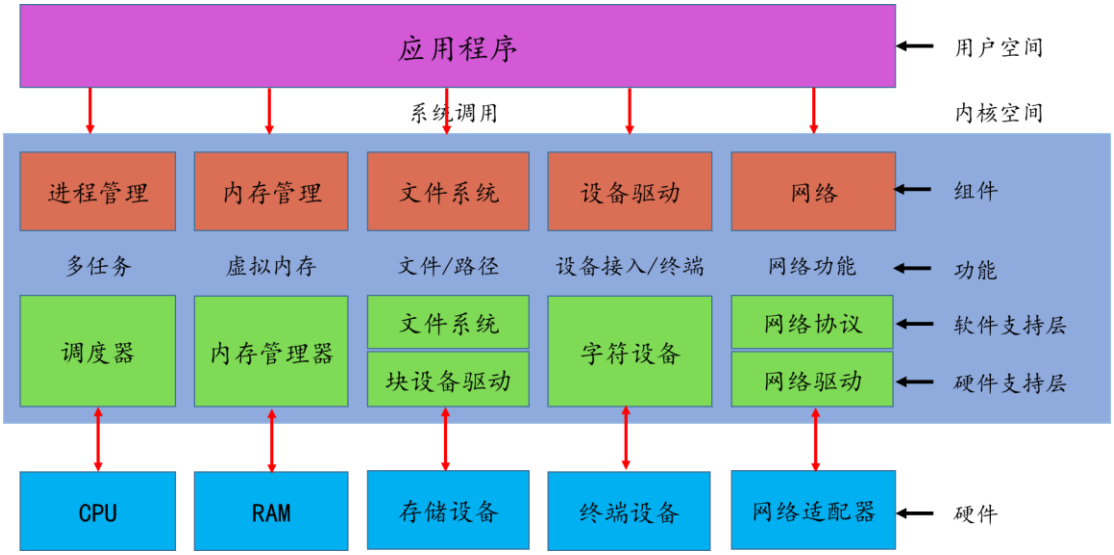
```
  HOSTCC  scripts/sign-file
scripts/sign-file.c:23:30: fatal error: openssl/opensslv.h: No such file or directory
 #include <openssl/opensslv.h>
                              ^
```

安装 ssl 开发库

```
$ sudo apt-get install libssl-dev
```

Linux 内核的整体架构

2.1 整体架构与子系统划分



上图说明了 Linux 内核的整体架构。根据内核的核心功能，Linux 内核提出了 5 个子系统，分别负责如下的功能：

1. Process Scheduler，也称作进程管理、进程调度。负责管理 CPU 资源，以便让各个进程可以以尽量公平的方式访问 CPU。

2. Memory Manager，内存管理。负责管理 Memory（内存）资源，以便让各个进程可以安全地共享机器的内存资源。另外，内存管理会提供虚拟内存的机制，该机制可以让进程使用多于系统可用 Memory 的内存，不用的内存会通过文件系统保存在外部非易失存储器中，需要使用的时候，再取回到内存中。

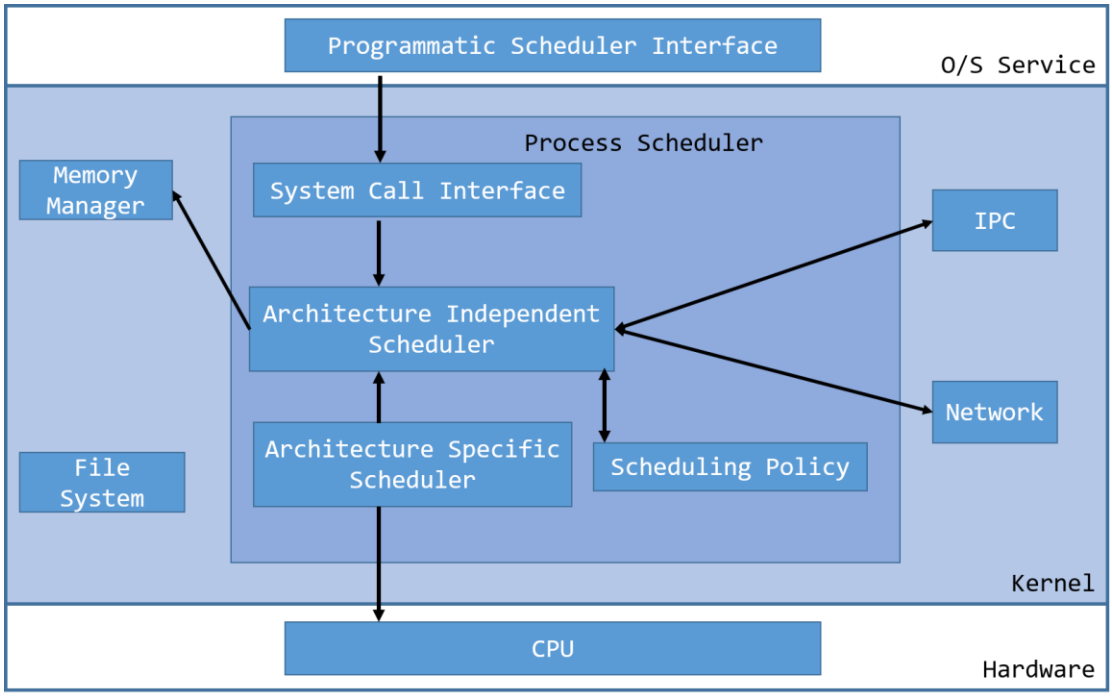
3. VFS（Virtual File System），虚拟文件系统。Linux 内核将不同功能的外部设备，例如 Disk 设备（硬盘、磁盘、NAND Flash、Nor Flash 等）、输入输出设备、显示设备等等，抽象为可以通过统一的文件操作接口（open、close、read、write 等）来访问。这就是 Linux 系统“一切皆是文件”的体现（其实 Linux 做的并不彻底，因为 CPU、内存、网络等还不是文件，如果真的需要一切皆是文件）。

4. 设备驱动，负责管理第三方设备接入/终端。

5. Network，网络子系统。负责管理系统的网络设备，并实现多种多样的网络标准。

2.2 进程调度（Process Scheduler)

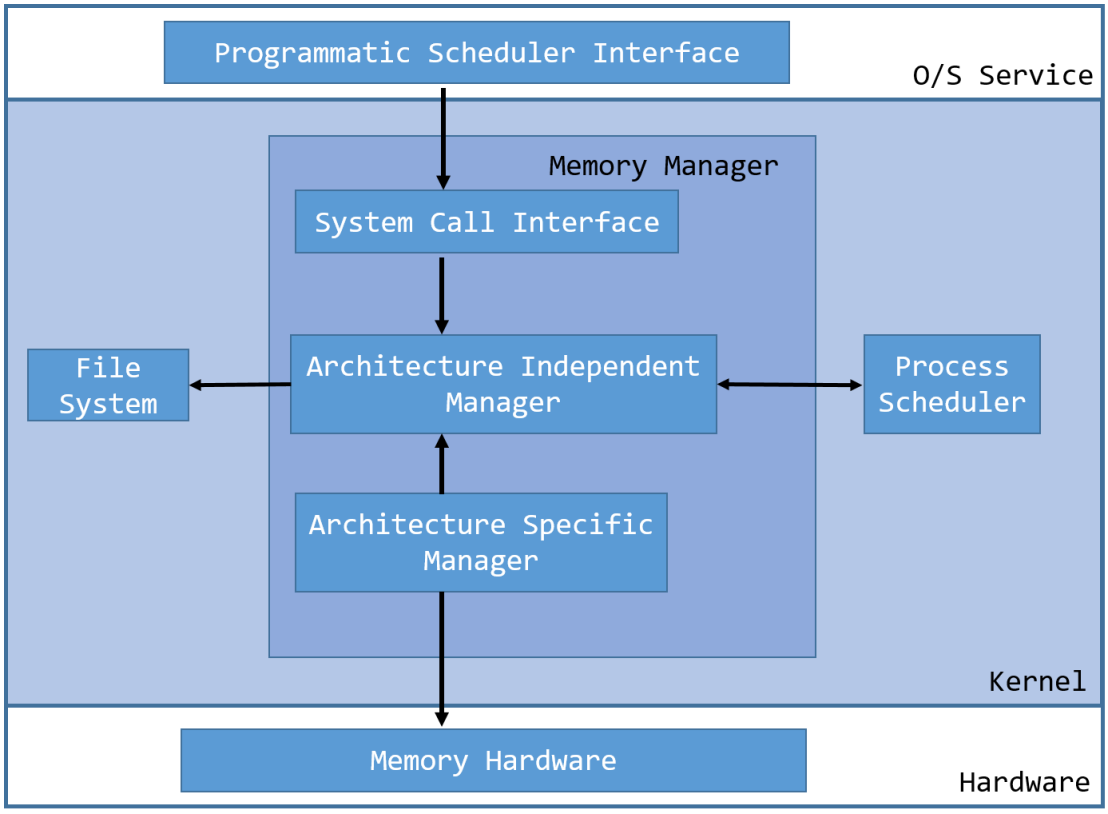
进程调度是 Linux 内核中最重要的子系统，它主要提供对 CPU 的访问控制。因为在计算机中，CPU 资源是有限的，而众多的应用程序都要使用 CPU 资源，所以需要“进程调度子系统”对 CPU 进行调度管理。
进程调度子系统包括 4 个子模块（见下图），它们的功能如下：



1. Scheduling Policy，实现进程调度的策略，它决定哪个（或哪几个）进程将拥有 CPU。
2. Architecture-specific Schedulers，体系结构相关的部分，用于将对不同 CPU 的控制，抽象为统一的接口。这些控制主要在 suspend 和 resume 进程时使用，牵涉到 CPU 的寄存器访问、汇编指令操作等。
3. Architecture-independent Scheduler，体系结构无关的部分。它会和“Scheduling Policy 模块”沟通，决定接下来要执行哪个进程，然后通过“Architecture-specific Schedulers 模块”resume 指定的进程。
4. System Call Interface，系统调用接口。进程调度子系统通过系统调用接口，将需要提供给用户空间的接口开放出去，同时屏蔽掉不需要用户空间程序关心的细节。

2.3 内存管理（Memory Manager, MM)

内存管理同样是 Linux 内核中最重要的子系统，它主要提供对内存资源的访问控制。Linux 系统会在硬件物理内存和进程所使用的内存（称作虚拟内存）之间建立一种映射关系，这种映射是以进程为单位，因而不同的进程可以使用相同的虚拟内存，而这些相同的虚拟内存，可以映射到不同的物理内存上。
内存管理子系统包括 3 个子模块（见下图），它们的功能如下：



1. Architecture Specific Managers，体系结构相关部分。提供用于访问硬件 Memory 的虚拟接口。

2. Architecture Independent Manager，体系结构无关部分。提供所有的内存管理机制，包括：以进程为单位的 memory mapping；虚拟内存的 Swapping。

3. System Call Interface，系统调用接口。通过该接口，向用户空间程序应用程序提供内存的分配、释放，文件的 map 等功能。

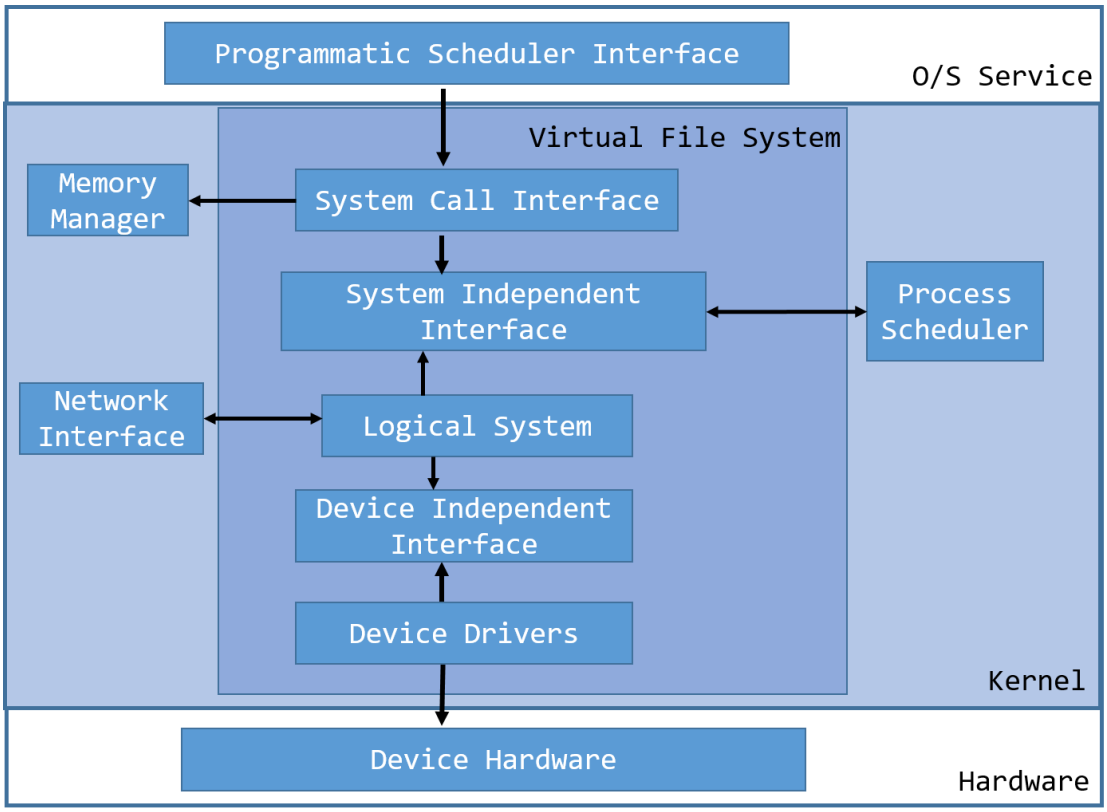
2.4 虚拟文件系统（Virtual Filesystem, VFS）

传统意义上的文件系统，是一种存储和组织计算机数据的方法。它用易懂、人性化的方法（文件和目录结构），抽象计算机磁盘、硬盘等设备上冰冷的数据块，从而使对它们的查找和访问变得容易。因而文件系统的实质，就是“存储和组织数据的方法”，文件系统的表现形式，就是“从某个设备中读取数据和向某个设备写入数据”。

随着计算机技术的进步，存储和组织数据的方法也是在不断进步的，从而导致有多种类型的文件系统，例如 FAT、FAT32、NTFS、EXT2、EXT3 等等。而为了兼容，操作系统或者内核，要以相同的表现形式，同时支持多种类型的文件系统，这就延伸出了虚拟文件系统（VFS）的概念。VFS 的功能就是管理各种各样的文件系统，屏蔽它们的差异，以统一的方式，为用户程序提供访问文件的接口。

我们可以从磁盘、硬盘、NAND Flash 等设备中读取或写入数据，因而最初的文件系统都是构建在这些设备之上的。这个概念也可以推广到其它的硬件设备，例如内存、显示器（LCD）、键盘、串口等等。我们对硬件设备的访问控制，也可以归纳为读取或者写入数据，因而可以用统一的文件操作接口访问。Linux 内核就是这样做的，除了传统的磁盘文件系统之外，它

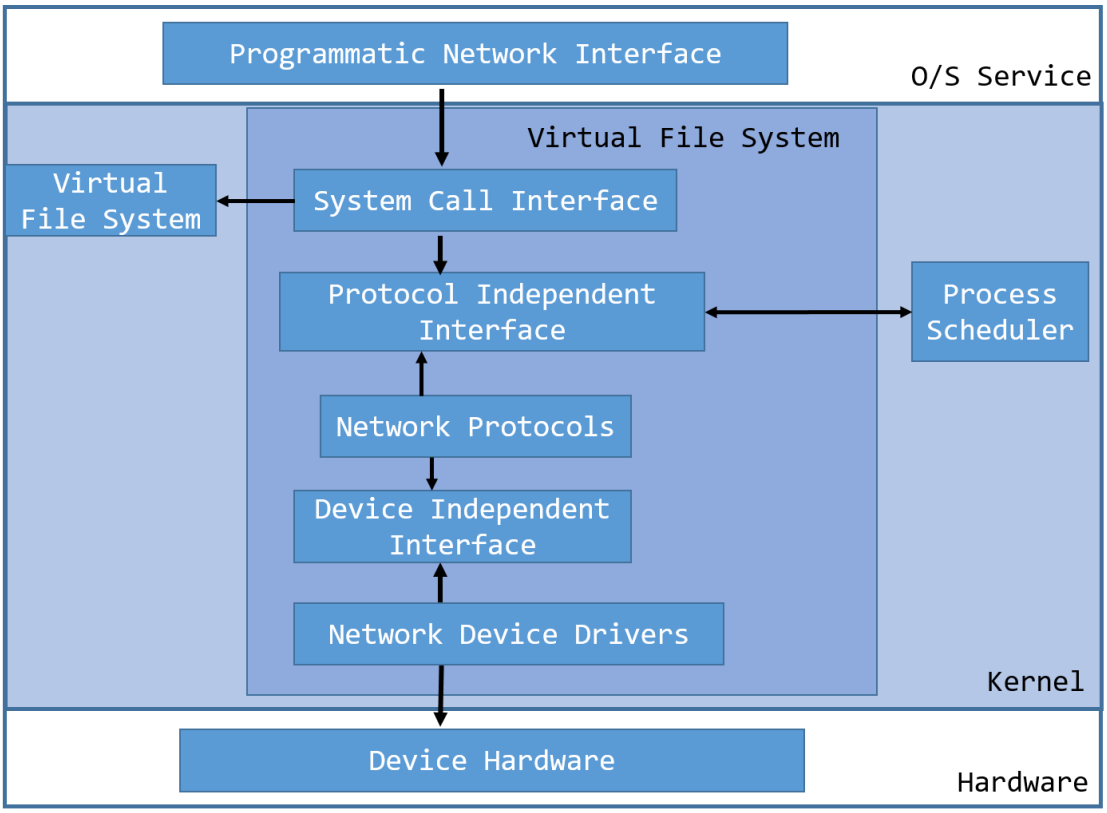
还抽象出了设备文件系统、内存文件系统等等。这些逻辑，都是由 VFS 子系统实现。
VFS 子系统包括 6 个子模块（见下图），它们的功能如下：



1. Device Drivers，设备驱动，用于控制所有的外部设备及控制器。由于存在大量不能相互兼容的硬件设备（特别是嵌入式产品），所以也有非常多的设备驱动。因此，Linux 内核中将近一半的 Source Code 都是设备驱动，大多数的 Linux 底层工程师（特别是国内的企业）都是在编写或者维护设备驱动，而无暇估计其它内容（它们恰恰是 Linux 内核的精髓所在）。
2. Device Independent Interface， 该模块定义了描述硬件设备的统一方式（统一设备模型），所有的设备驱动都遵守这个定义，可以降低开发的难度。同时可以用一致的形势向上提供接口。
3. Logical Systems，每一种文件系统，都会对应一个 Logical System（逻辑文件系统），它会实现具体的文件系统逻辑。
4. System Independent Interface，该模块负责以统一的接口（快设备和字符设备）表示硬件设备和逻辑文件系统，这样上层软件就不再关心具体的硬件形态了。
5. System Call Interface，系统调用接口，向用户空间提供访问文件系统和硬件设备的统一的接口。

2.5 网络子系统（Net）

网络子系统在 Linux 内核中主要负责管理各种网络设备，并实现各种网络协议栈，最终实现通过网络连接其它系统的功能。在 Linux 内核中，网络子系统几乎是自成体系，它包括 5 个子模块（见下图），它们的功能如下：



1. Network Device Drivers，网络设备的驱动，和 VFS 子系统中的设备驱动是一样的。
2. Device Independent Interface，和 VFS 子系统中的是一样的。
3. Network Protocols，实现各种网络传输协议，例如 IP, TCP, UDP 等等。
4. Protocol Independent Interface，屏蔽不同的硬件设备和网络协议，以相同的格式提供接口（socket)。
5. System Call interface，系统调用接口，向用户空间提供访问网络设备的统一的接口。

Linux 内核源代码的目录结构

Linux 内核源代码包括三个主要部分：
1. 内核核心代码，包括第 3 章所描述的各个子系统和子模块，以及其它的支撑子系统，例如电源管理、Linux 初始化等
2. 其它非核心代码，例如库文件（因为 Linux 内核是一个自包含的内核，即内核不依赖其它的任何软件，自己就可以编译通过）、固件集合、KVM（虚拟机技术）等
3. 编译脚本、配置文件、帮助文档、版权说明等辅助性文件

下图示使用 ls 命令看到的内核源代码的顶层目录结构，具体描述如下。

```
wangbojing@ubuntu:~/share/linux-4.4.16$ ls
arch      CREDITS        firmware  ipc       lib          modules.builtin  README           security  usr          vmlinux.o
block     crypto         fs        Kbuild    MAINTAINERS  modules.order    REPORTING-BUGS   sound     virt
certs     Documentation  include   Kconfig   Makefile     Module.symvers   samples          System.map  vmlinux
COPYING   drivers        init      kernel    mm           net              scripts          tools     vmlinux-gdb.py
wangbojing@ubuntu:~/share/linux-4.4.16$
```

include/ ---- 内核头文件，需要提供给外部模块（例如用户空间代码）使用。
kernel/ ---- Linux 内核的核心代码，包含了 2.2 小节所描述的进程调度子系统，以及和进程调度相关的模块。
mm/ ---- 内存管理子系统（2.3 小节）。

fs/ ---- VFS 子系统（2.4 小节）。
net/ ---- 不包括网络设备驱动的网络子系统（2.5 小节）。
ipc/ ---- IPC（进程间通信）子系统。
arch// ---- 体系结构相关的代码，例如 arm, x86 等等。
arch//mach- ---- 具体的 machine/board 相关的代码。
arch//include/asm ---- 体系结构相关的头文件。
arch//boot/dts ---- 设备树（Device Tree）文件。
init/ ---- Linux 系统启动初始化相关的代码。
block/ ---- 提供块设备的层次。
sound/ ---- 音频相关的驱动及子系统，可以看作“音频子系统”。
drivers/ ---- 设备驱动
lib/ ---- 实现需要在内核中使用的库函数，例如 CRC、FIFO、list、MD5 等。
crypto/ ----- 加密、解密相关的库函数。
security/ ---- 提供安全特性（SELinux）。
virt/ ---- 提供虚拟机技术（KVM 等）的支持。
usr/ ---- 用于生成 initramfs 的代码。
firmware/ ---- 保存用于驱动第三方设备的固件。
samples/ ---- 一些示例代码。
tools/ ---- 一些常用工具，如性能剖析、自测试等。
Kconfig, Kbuild, Makefile, scripts/ ---- 用于内核编译的配置文件、脚本等。
COPYING ---- 版权声明。
MAINTAINERS ----维护者名单。
CREDITS ---- Linux 主要的贡献者名单。
REPORTING-BUGS ---- Bug 上报的指南。
Documentation, README ---- 帮助、说明文档。

Linux 内核 常用数据结构

Linux 内核代码中广泛使用了数据结构和算法，其中最常用的两个是链表和红黑树。

链表

Linux 内核代码大量使用了链表这种数据结构。链表是在解决数组不能动态扩展这个缺陷而产生的一种数据结构。链表所包含的元素可以动态创建并插入和删除。链表的每个元素都是离散存放的，因此不需要占用连续的内存。链表通常由若干节点组成，每个节点的结构都是一样的，由有效数据区和指针区两部分组成。有效数据区用来存储有效数据信息，而指针区用来指向链表的前继节点或者后继节点。因此，链表就是利用指针将各个节点串联起来的一种存储结构。

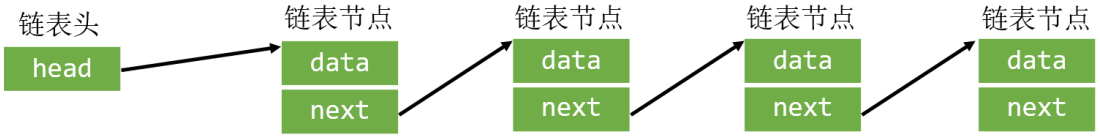
（1）单向链表

单向链表的指针区只包含一个指向下一个节点的指针，因此会形成一个单一方向的链表，如

下代码所示。

```
struct list {
    int data;   /*有效数据*/
    struct list *next; /*指向下一个元素的指针*/
};
```

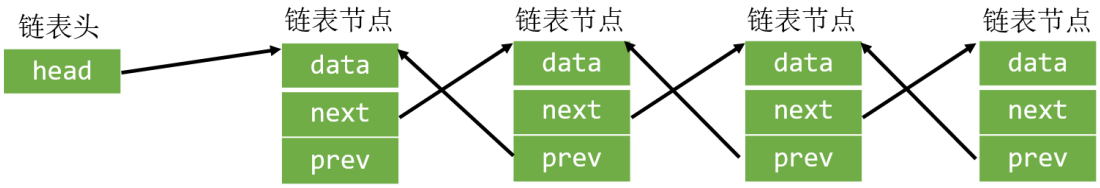
如图所示，单向链表具有单向移动性，也就是只能访问当前的节点的后继节点，而无法访问当前节点的前继节点，因此在实际项目中运用得比较少。



（2）双向链表

如图所示，双向链表和单向链表的区别是指针区包含了两个指针，一个指向前继节点，另一个指向后继节点，如下代码所示。

```
struct list {
    int data;   /*有效数据*/
    struct list *next; /*指向下一个元素的指针*/
    struct list *prev; /*指向上一个元素的指针*/
};
```



（3）Linux 内核链表实现

单向链表和双向链表在实际使用中有一些局限性，如数据区必须是固定数据，而实际需求是多种多样的。这种方法无法构建一套通用的链表，因为每个不同的数据区需要一套链表。为此，Linux 内核把所有链表操作方法的共同部分提取出来，把不同的部分留给代码编程者自己去处理。Linux 内核实现了一套纯链表的封装，链表节点数据结构只有指针区而没有数据区，另外还封装了各种操作函数，如创建节点函数、插入节点函数、删除节点函数、遍历节点函数等。

Linux 内核链表使用 struct list_head 数据结构来描述。

```
<include/linux/types.h>

struct list_head {
    struct list_head *next, *prev;
};
```

struct list_head 数据结构不包含链表节点的数据区，通常是嵌入其他数据结构，如 struct page 数据结构中嵌入了一个 lru 链表节点，通常是把 page 数据结构挂入 LRU 链表。

```
<include/linux/mm_types.h>

struct page {
    ...
    struct list_head lru;
    ...
}
```

链表头的初始化有两种方法，一种是静态初始化，另一种动态初始化。把 next 和 prev 指针都初始化并指向自己，这样便初始化了一个带头节点的空链表。

```
<include/linux/list.h>

/*静态初始化*/
#define LIST_HEAD_INIT(name) { &(name), &(name) }

#define LIST_HEAD(name) \
    struct list_head name = LIST_HEAD_INIT(name)

/*动态初始化*/
static inline void INIT_LIST_HEAD(struct list_head *list)
{
    list->next = list;
    list->prev = list;
}
```

添加节点到一个链表中，内核提供了几个接口函数，如 list_add()是把一个节点添加到表头，list_add_tail()是插入表尾。

```
<include/linux/list.h>

void list_add(struct list_head *new, struct list_head *head)
list_add_tail(struct list_head *new, struct list_head *head)
```

遍历节点的接口函数。

```
#define list_for_each(pos, head) \
for (pos = (head)->next; pos != (head); pos = pos->next)
```

这个宏只是遍历一个一个节点的当前位置，那么如何获取节点本身的数据结构呢？这里还需要使用 list_entry()宏。

```
#define list_entry(ptr, type, member) \
    container_of(ptr, type, member)
```

container_of()宏的定义在 kernel.h 头文件中。

```
#define container_of(ptr, type, member) ({           \
    const typeof( ((type *)0)->member ) *__mptr = (ptr);    \
    (type *)( (char *)__mptr - offsetof(type,member) );})

#define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)
```

其中 offsetof()宏是通过把 0 地址转换为 type 类型的指针，然后去获取该结构体中 member 成员的指针，也就是获取了 member 在 type 结构体中的偏移量。最后用指针 ptr 减去 offset，就得到 type 结构体的真实地址了。

下面是遍历链表的一个例子。

```
<drivers/block/osdblk.c>

static ssize_t class_osdblk_list(struct class *c,
                struct class_attribute *attr,
                char *data)
{
    int n = 0;
    struct list_head *tmp;

    list_for_each(tmp, &osdblkdev_list) {
        struct osdblk_device *osdev;

        osdev = list_entry(tmp, struct osdblk_device, node);

        n += sprintf(data+n, "%d %d %llu %llu %s\n",
            osdev->id,
            osdev->major,
            osdev->obj.partition,
            osdev->obj.id,
            osdev->osd_path);
    }
    return n;
}
```

红黑树

红黑树（Red Black Tree）被广泛应用在内核的内存管理和进程调度中，用于将排序的元素组织到树中。红黑树被广泛应用在计算机科学的各个领域中，它在速度和实现复杂度之间提供一个很好的平衡。

红黑树是具有以下特征的二叉树。

- 每个节点或红或黑。
- 每个叶节点是黑色的。
- 如果结点都是红色，那么两个子结点都是黑色。
- 从一个内部结点到叶结点的简单路径上，对所有叶节点来说，黑色结点的数目都是相同的。

红黑树的一个优点是，所有重要的操作（例如插入、删除、搜索）都可以在 $O(\log n)$时间内完成，n 为树中元素的数目。经典的算法教科书都会讲解红黑树的实现，这里只是列出一个内核中使用红黑树的例子，供读者在实际的驱动和内核编程中参考。这个例子可以在内核代码的 documentation/Rbtree.txt 文件中找到。

```
#include <linux/init.h>
#include <linux/list.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/slab.h>
#include <linux/mm.h>
#include <linux/rbtree.h>

MODULE_AUTHOR("figo.zhang");
MODULE_DESCRIPTION(" ");
MODULE_LICENSE("GPL");

  struct mytype {
     struct rb_node node;
     int key;
};

/*红黑树根节点*/
 struct rb_root mytree = RB_ROOT;
/*根据 key 来查找节点*/
struct mytype *my_search(struct rb_root *root, int new)
  {
     struct rb_node *node = root->rb_node;

     while (node) {
          struct mytype *data = container_of(node, struct
mytype, node);

          if (data->key > new)
               node = node->rb_left;
          else if (data->key < new)
               node = node->rb_right;
          else
               return data;
```

```
    }
    return NULL;
  }

/*插入一个元素到红黑树中*/
  int my_insert(struct rb_root *root, struct mytype *data)
  {
    struct rb_node **new = &(root->rb_node), *parent=NULL;

    /* 寻找可以添加新节点的地方 */
    while (*new) {
        struct mytype *this = container_of(*new, struct
mytype, node);

        parent = *new;
        if (this->key > data->key)
            new = &((*new)->rb_left);
        else if (this->key < data->key) {
            new = &((*new)->rb_right);
        } else
            return -1;
    }

    /* 添加一个新节点 */
    rb_link_node(&data->node, parent, new);
    rb_insert_color(&data->node, root);

    return 0;
  }

static int __init my_init(void)
{
    int i;
    struct mytype *data;
    struct rb_node *node;

    /*插入元素*/
    for (i =0; i < 20; i+=2) {
        data = kmalloc(sizeof(struct mytype), GFP_KERNEL);
        data->key = i;
        my_insert(&mytree, data);
    }

    /*遍历红黑树，打印所有节点的 key 值*/
```

```
     for (node = rb_first(&mytree); node; node =
rb_next(node))
          printk("key=%d\n", rb_entry(node, struct mytype,
node)->key);

     return 0;
}

static void __exit my_exit(void)
{
     struct mytype *data;
     struct rb_node *node;
     for (node = rb_first(&mytree); node; node = rb_next(node))
{
          data = rb_entry(node, struct mytype, node);
          if (data) {
               rb_erase(&data->node, &mytree);
               kfree(data);
          }
     }
}
module_init(my_init);
module_exit(my_exit);
```

mytree 是红黑树的根节点，my_insert()实现插入一个元素到红黑树中，my_search()根据 key 来查找节点。内核大量使用红黑树。

CFS 调度器

CFS（完全公平调度器）实现的主要思想是维护为任务提供处理器时间方面的平衡（公平性），这意味着应给进程分配相当数量的处理器。分给某个任务的时间失去平衡时（意味着一个或多个任务相对于其他任务而言未被给予相当数量的时间），应给失去平衡的任务分配时间，让其执行。

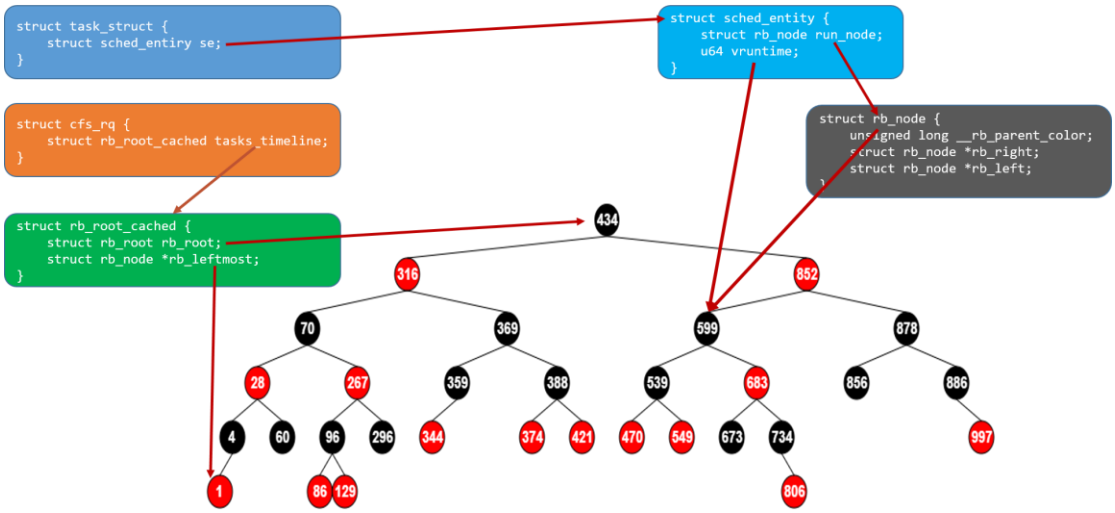
CFS 通过虚拟运行时间（vruntime）来实现平衡，维护提供给某个任务的时间量。进程的虚拟时间是指实际运行时间相对于权重为 0 的进程的比例值。在 CFS 调度器中有一个计算虚拟时间的核心函数 calc_delta_fair(),它的计算公式为：

vruntime = 实际运行时间*1024 / 进程权重

因此，进程按照各自不同的速率在物理时钟节拍内前进，优先级高则权重大，其虚拟时钟比真实时钟跑得慢，但获得比较多的运行时间；反之，优先级低则权重小，其虚拟时钟比真实时钟跑得快，反而获得比较少的运行时间。CFS 调度器总是选择虚拟时钟跑得慢的进程来运行，从而让每个调度实体（sche_entity）的虚拟运行时间互相追赶，进而实现进程调度上的平衡。

CFS 调度器没有将进程维护在运行队列中，而是维护了一个以虚拟运行时间为顺序的红黑树。红黑树的主要特点有：

1. 自平衡，树上没有一条路径会比其他路径长出俩倍。
2. O(log n) 时间复杂度，能够在树上进行快速高效地插入或删除进程。



Linux 内的所有任务都由称为 task_struct 的任务结构表示，它位于调度的最顶端。该结构（在./linux/include/linux/sched.h）完整地描述了任务并包括了任务的当前状态、其堆栈、进程标识、优先级（静态和动态）等等。

```
struct task_struct{
    ...
    volatile long state;
    void *stack;
    unsigned int flags;
    int prio;
    int static_prio;
    int normal_prio;
    struct sche_entity se;
    ...
};
```

但是，由于不是所有任务都是可运行的，所以在 task_struct 中不会发现任何与 CFS 相关的字段。因此，需要通过一个名为 sched_entity 的新结构来跟踪调度信息。

```
struct sched_entity{
    ...
    struct load_weight load;
    struct rb_node run_node;
    struct list_head group_node;

    u64 vruntime;
    ...
};
```

sched_entity 包含负载权重、各种统计数据以及 vruntime（任务运行的虚拟时间量，并作为红黑树的索引）。同时，sched_entity 还包含红黑树的节点 rb_node。

```
struct rb_node{
```

```
    unsigned long __rb_parent_color;
    struct rb_node *rb_right;
    struct rb_node *rb_left;
};
```

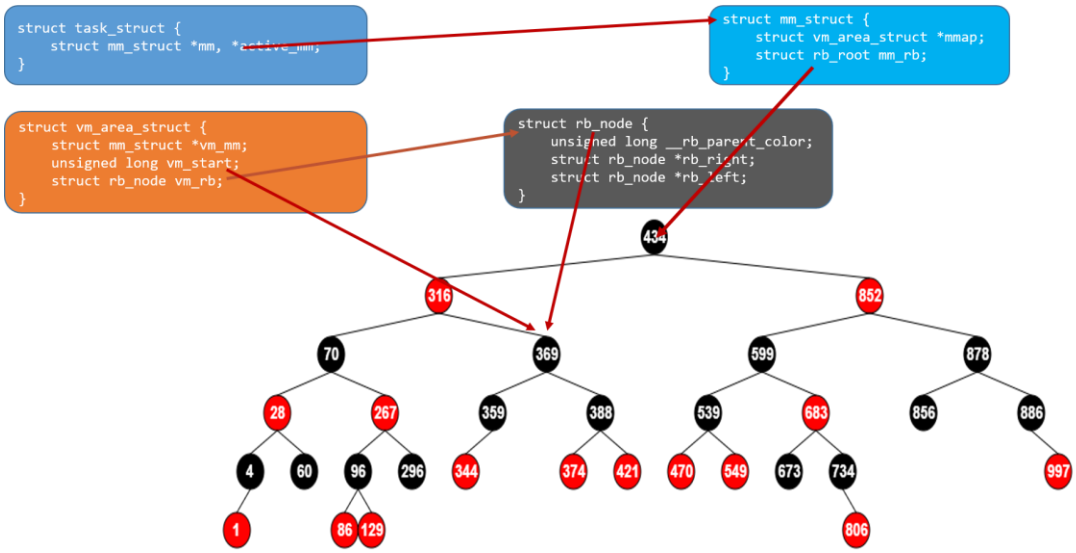
红黑树的每个节点都由 rb_node 表示，它只包含子引用和父对象的颜色。红黑树的叶子不包含信息，但是内部节点代表一个或多个可运行的任务。红黑树的根通过 rb_root_cached 结构中的 rb_root 引用，而该结构同时包含了红黑树的最左节点 rb_leftmost 的指针。

```
struct rb_root_cached{
    struct rb_root rb_root;
    struct rb_node *rb_leftmost;
};
```

在运行过程中，__schedule()（在./kernel/sched/core.c 中）是 CFS 调度器的核心函数，其作用是让调度器选择和切换到一个合适的进程运行。
在时钟周期开始时，调度器调用__schedule()函数来开始调度的运行。
然后，__schedule()函数调用 pick_next_task()让进程调度器从就绪队列中选择一个最合适的进程 next，即红黑树最左边的节点。
接着，通过 context_switch()切换到新的地址空间，从而保证 next 进程运行。
在时钟周期结束时，调度器调用 entity_tick()函数来更新进程负载、进程状态以及 vruntime（当前 vruntime + 该时钟周期内运行的时间）。
最后，将该进程的虚拟时间与就绪队列红黑树中最左边的调度实体的虚拟时间做比较，如果小于坐左边的时间，则不用触发调度，继续调度当前调度实体。否则，则表明最左边的调度实体更需要调度。因此，调度器将当前调度实体放回红黑树，并选择红黑树中最左边的调度实体作为 next 在下一个时钟周期进行调度。
通过以上的结构和调度方式，Linux 内核保证了操作系统中进程调度的公平性。

虚拟内存管理

在内存管理中，在内核管理某个进程的内存时使用了红黑树，见下面数据结构（只保留和内存管理相关的成员），每个进程都有一个 active_mm 的成员用于管理该进程的虚拟内存空间。struct mm_struct 中的成员 mm_rb 是红黑树的根，该进程的所有虚拟空间块（虚拟地址不连续）都以起始虚拟地址为 key 值挂在该红黑树上。该进程新申请的虚拟内存区间会插入到这棵树上，当然插入过程中可能会合并相邻的虚拟区域。删除时会从该树上摘除相应的 node。



在 32 位的系统上，线性地址空间可达到 4GB，这 4GB 一般按照 3:1 的比例进行分配，也就是说用户进程享有前 3GB 线性地址空间，而内核独享最后 1GB 线性地址空间。由于虚拟内存的引入，每个进程都可拥有 3GB 的虚拟内存，并且用户进程之间的地址空间是互不可见、互不影响的，也就是说即使两个进程对同一个地址进行操作，也不会产生问题。在前面介绍的一些分配内存的途径中，无论是伙伴系统中分配页的函数，还是 slab 分配器中分配对象的函数，它们都会尽量快速地响应内核的分配请求，将相应的内存提交给内核使用，而内核对待用户空间显然不能如此。用户空间动态申请内存时往往只是获得一块线性地址的使用权，而并没有将这块线性地址区域与实际的物理内存对应上，只有当用户空间真正操作申请的内存时，才会触发一次缺页异常，这时内核才会分配实际的物理内存给用户空间。

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，一个合法的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。

从进程的角度来讲，VMA 其实是虚拟空间的内存块，一个进程的所有资源由多个内存块组成，所以，一个进程的描述结构 task_struct 中首先包含 Linux 的内存描述符 mm_struct 结构。

```
struct task_struct {
.......
    struct mm_struct *mm;
.......
}
```

在 mm_struct 中进而包含了 vm_area_struct :

```
struct mm_struct {
          struct vm_area_struct * mmap;       /* list of VMAs */
          struct rb_root mm_rb;
          struct vm_area_struct * mmap_cache;      /* last find_vma result
*/
.......
}
```

一个进程的每个 VMA 块都会链接到中的链表和红黑树：

1. mmap 形成一个单链表，一个进程的所有 VMA 都链接到这个链表，链表头是 mm->mmap
2. mm_rb 是红黑树节点，每个进程都一个 VMA 红黑树

VMA 按照起始地址递增的方式，插入到 mm_struct->mmap 链表。当进程拥有大量的 VMA 的时候，搜索效率比较低，所以哟娜那个到红黑树来加快查找。

接下来看看这次的主角 vm_area_struct：

```
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
```

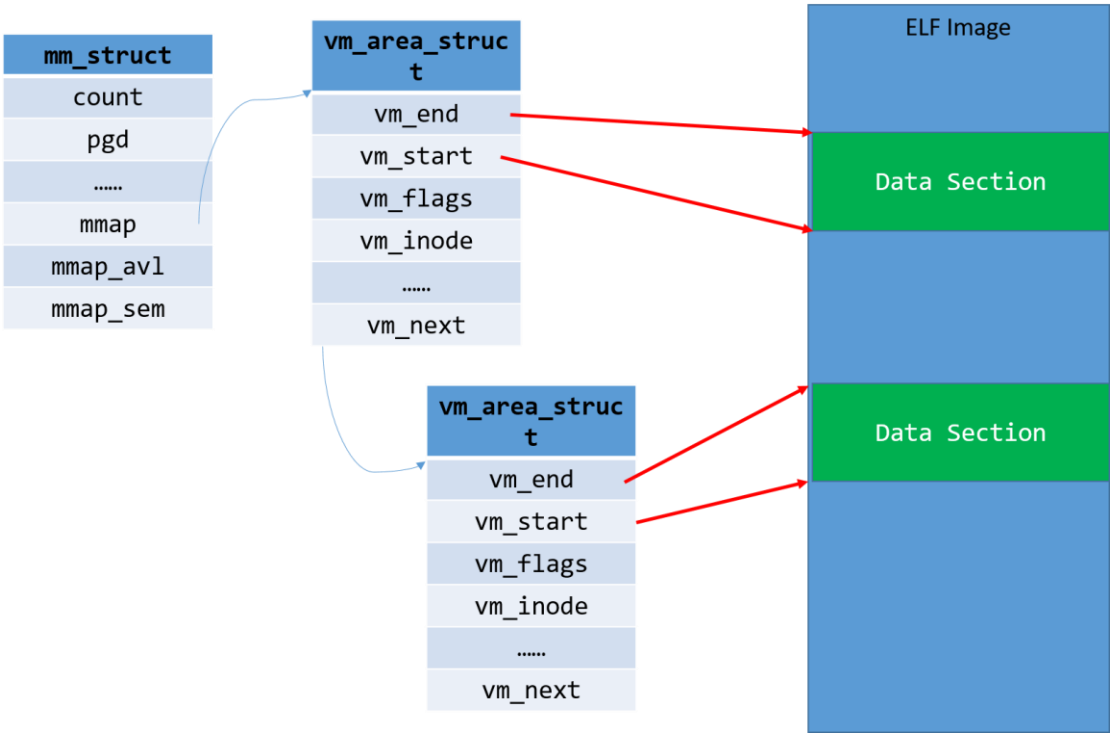
先树*/
    } shared;

    /*
     * 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
};
所以进程的 VMA 的组织为：



无锁环形缓冲区

生产者和消费者模型是计算机编程中最常见的一种模型。生产者产生数据，而消费者消耗数据，如一个网络设备，硬件设备接收网络包，然后应用程序读取网络包。环形缓冲区是实现生产者和消费者模型的经典算法。环形缓冲区通常有一个读指针和一个写指针。读指针指向环形缓冲区中可读的数据，写指针指向环形缓冲区可写的数据。通过移动读指针和写指针实现缓冲区数据的读取和写入。

在 Linux 内核中，KFIFO 是采用无锁环形缓冲区的实现。FIFO 的全称是“First In First Out”，即先进先出的数据结构，它采用环形缓冲区的方法来实现，并提供一个无边界的字节流服务。采用环形缓冲区的好处是，当一个数据元素被消耗之后，其余数据元素不需要移动其存储位置，从而减少复制，提高效率。

（1）创建 KFIFO

在使用 KFIFO 之前需要进行初始化，这里有静态初始化和动态初始化两种方式。

```
<include/linux/kfifo.h>

int kfifo_alloc(fifo, size, gfp_mask)
```

该函数创建并分配一个大小为 size 的 KFIFO 环形缓冲区。第一个参数 fifo 是指向该环形缓冲区的 struct kfifo 数据结构；第二个参数 size 是指定缓冲区元素的数量；第三个参数 gfp_mask 表示分配 KFIFO 元素使用的分配掩码。

静态分配可以使用如下的宏。

```
#define DEFINE_KFIFO(fifo, type, size)
#define INIT_KFIFO(fifo)
```

（2）入列

把数据写入 KFIFO 环形缓冲区可以使用 kfifo_in()函数接口。

```
int kfifo_in(fifo, buf, n)
```

该函数把 buf 指针指向的 *n* 个数据复制到 KFIFO 环形缓冲区中。第一个参数 fifo 指的是 KFIFO 环形缓冲区；第二个参数 buf 指向要复制的数据的 buffer；第三个数据是要复制数据元素的数量。

（3）出列

从 KFIFO 环形缓冲区中列出或者摘取数据可以使用 kfifo_out()函数接口。

```
#define    kfifo_out(fifo, buf, n)
```

该函数是从 fifo 指向的环形缓冲区中复制 *n* 个数据元素到 buf 指向的缓冲区中。如果 KFIFO 环形缓冲区的数据元素小于 *n* 个，那么复制出去的数据元素小于 *n* 个。

（4）获取缓冲区大小

KFIFO 提供了几个接口函数来查询环形缓冲区的状态。

```
#define kfifo_size(fifo)
#define kfifo_len(fifo)
#define    kfifo_is_empty(fifo)
#define    kfifo_is_full(fifo)
```

kfifo_size()用来获取环形缓冲区的大小，也就是最大可以容纳多少个数据元素。kfifo_len()用来获取当前环形缓冲区中有多少个有效数据元素。kfifo_is_empty()判断环形缓冲区是否为空。kfifo_is_full()判断环形缓冲区是否为满。

（5）与用户空间数据交互

KFIFO 还封装了两个函数与用户空间数据交互。

```
#define    kfifo_from_user(fifo, from, len, copied)
#define    kfifo_to_user(fifo, to, len, copied)
```

kfifo_from_user()是把 from 指向的用户空间的 len 个数据元素复制到 KFIFO 中，最后一个参数 copied 表示成功复制了几个数据元素。kfifo_to_user()则相反，把 KFIFO 的数据元素复制到用户空间。这两个宏结合了 copy_to_user()、copy_from_user()以及 KFIFO 的机制，给驱动开发者提供了方便。