

# Linux内核编译及添加系统调用（详细版）



Linux内核园

9 人赞同了该文章

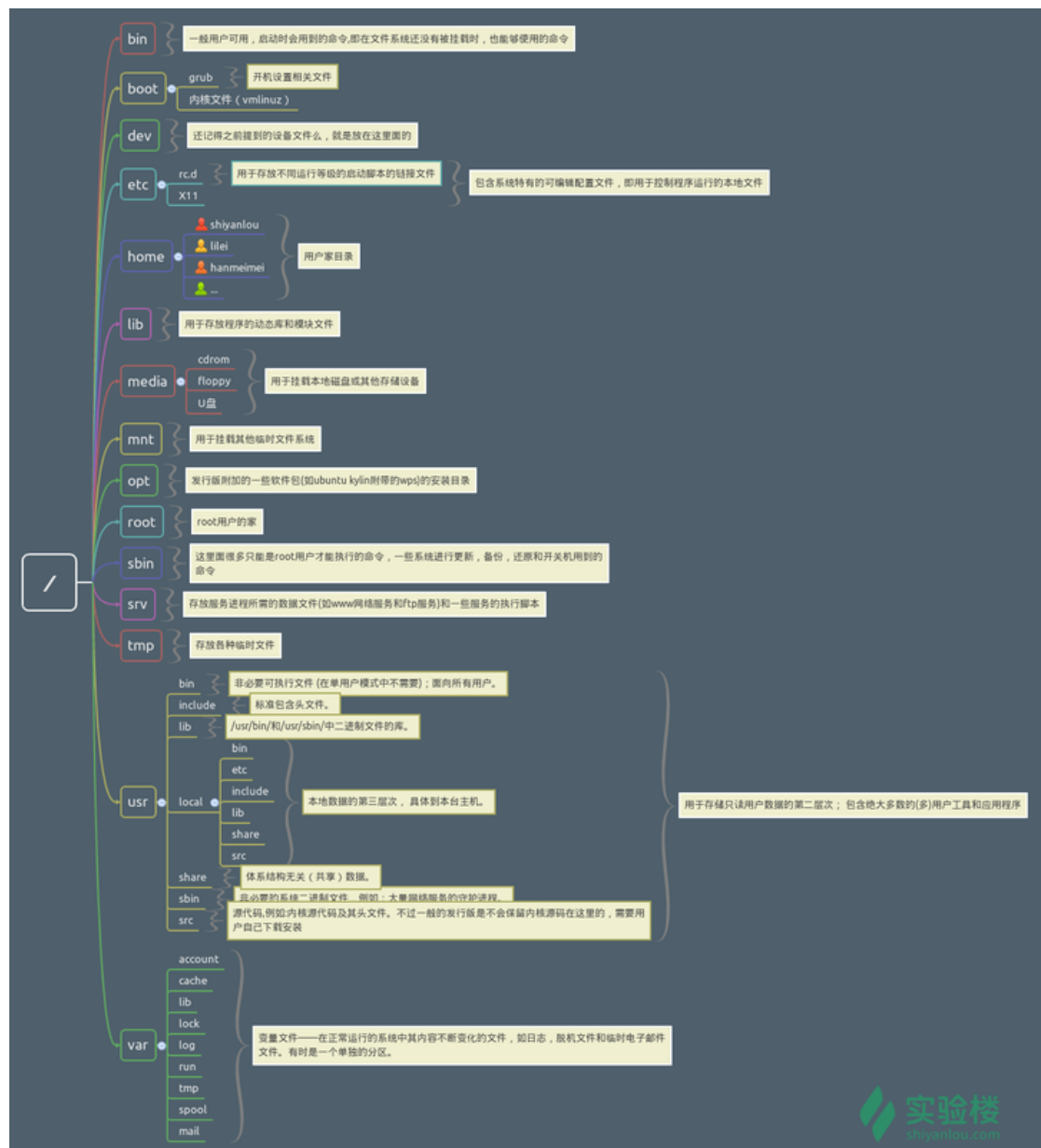
## 实验一：Linux内核编译及添加系统调用（HDU）

花了一上午的时间来写这个，良心制作，发现自己刚学的时候没有找到很详细的，就是泛泛的说了下细节地方也没有，于是自己写了这个，有点长，如果你认真的看完了，也应该是懂了。

### 一、前期准备工作

1. 需要准备虚拟机上安装Ubuntu，笔者安装的是**Ubuntu18.04**，安装的教程自行百度解决，教程很多。有几点需要提一下，就是内存分配至少60G，核分配4个最好，为了在编译的时候别崩溃。

建议去熟悉一下Linux下面的文件目录结构，根目录下每个目录一般会存放什么样的文件



1.，然后常见命令操作也要熟悉一下。

2. 下载Linux内核地址,自行选择版本, 建议选择4.xx版本, 因为版本高出错的概率也大。  
下载好了之后, 会放在自己的Ubuntu中的Downloads目录下, 同时是一个压缩文件, 到时候需要解压到放内核目录文件下。首先进入到该Downloads文件目录下, 查看是否下载好了。

```
$cd ~/Downloads
$ls
linux-4.19.25.tar.xz
```

之后开始解压上面的那个压缩文件到存放内核的地方, 就是Linux系统的/**usr/src**目录下, 此目录用来存放内核源码的。从上图也可以了解到。

```
cd ~/Downloads
tar xvjf linux-4.19.25.tar.xz -C /usr/src
```

进入/**usr/src**目录查看是否有, 如果有就可以开始后续工作了。

### 【腾讯文档】Linux内核源码技术学习路线+视频教程代码资料

Linux内核源码技术学习路线+视频教程代码资料  
[docs.qq.com/doc/DWmNMckNQc21ZbENE](https://docs.qq.com/doc/DWmNMckNQc21ZbENE)

2021-12-30 15:  
2021-12-30 15:  
2021-12-30 15:  
2021-12-30 15:  
2021-12-30 15:

## 二、实验要求和内容

### 1. 内容要求:

(1) 添加一个系统调用, 实现对指定进程的nice值的修改或读取功能, 并返回进程最新的nice值及优先级。建议调用原型是int mysetnic(pid\_t pid, int flag, int nicevalue, void\_user\* prio, void\_user\* nice);

参数含义:

pid: 进程ID

flag: 若为0, 则表示读取nice的值; 若为1, 则表示修改nice的值。

nicevalue: 为指定的进程设置新的nice。

prio, nice: 指向进程的优先级和nice值。

返回值: 系统调用成功时返回0; 失败时返回错误码EFAULT。

(2) 写一个简单的应用程序测试 (1) 中添加的系统调用。

(3) 若系统调用了Linux的内核函数, 要求深入阅读相关的源码。

### 2. Linux系统调用的基本概念

实质是指调用内核函数, 于内核态中运行, Linux中的用户通过执行一条访管指令“int \$0x80”来调用系统调用, 该指令会产生一个访管中断, 从而让系统暂停当前的进程执行, 而转去执行系统调用处理程序。通过用户态传入的系统调用号从系统调用表中找到相应的服务例程的入口并执行, 完成后返回。

(1) 系统调用号与系统调用表: Linux内核中设置了一张系统调用表, 用于关联系统调用号及其相对应的服务例程入口地址, 定义在./arch/x86/entry/syscalls/syscall\_64.tbl文件中, 每个系统调用占一个表项, 一旦分配好就不可以有任何变更。

(2) 系统调用服务例程: 每个系统调用都对应一个内核服务例程来实现系统调用的功能, 其命名的格式都是以“sys\_”开头。其代码通常放在./kernel/sys.c中, 服务例程的原型声明则是放

在./include/linux/syscall.h中。如sys\_open,通常格式是asm linkage long sys\_open(int flag.....)。其中的asm linkage是一个必需的限定词,用于通知编译器从堆栈中提取函数的参数,而不是从寄存器中。

在sys.c中编程时,格式是 SYSCALL\_DEFINE5(mysetnice, pid\_t, pid, int, flag, int, nicevalue, void \_\_user \*, prio, void \_\_user \*, nice) **N=5**代表参数的个数。

(3) 系统调用参数传递:在X86中,Linux通过6个寄存器来传入参数,其中一个eax是传递系统调用号,后面的5个传递参数。

(4) 系统调用参数验证

### 三、开始实验

#### 1. 切换到root权限下,防止权限不够,导致出错。

```
$ sudo passwd root
[sudo] password for leslie:
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
$ su root
Password:
```

(1)首先,你安装Linux系统时,它会让你设置一个你的用户名和用户密码,在这里我设置的用户名是leslie,即绿色字体leslie@tp50的前半部分,后半部分tp50是我的主机名。

sudo放在命令首,意思是当前指令以管理员权限运行。

(2)passwd是一条命令,用来修改用户密码,参数root是超级用户名,拥有系统最高的权限。passwd root的意思是修改超级用户的密码,在创建Ubuntu时,默认超级用户是没有密码的(也可能是一个随机数之类的我不记得了),用这条命令重新设定一个密码。

(3)su是一条命令,用来切换当前用户,在第一章你会认识到,Linux是一个多用户多任务的操作系统。参数root是用户名,指示切换到的用户,在这里su root意为切换到root用户,当参数缺省时,默认切换到超级用户。

(4)你会发现,它在提示输入密码时,虽然键盘已经输入了密码,但是终端没有任何响应,不要担心,这正是Unix和Linux的特点,为了确保安全,在输入密码时不显示输入的内容,在输入密码后,直接按下回车就好了。

#### 2. 分配系统调用号,修改系统调用表

##### (1) 查看系统调用表,并修改

```
gedit /usr/src/linux-4.19.25/arch/x86/entry/syscalls/syscall_64.tbl
```

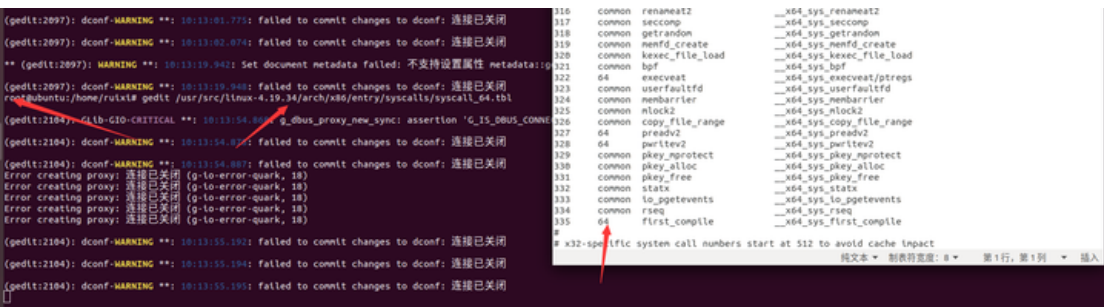
你只需要将**linux-4.19.25**换成你自己下载好的版本即可。

你会看见这个格式

| 名称            | 作用       |
|---------------|----------|
| <number>      | 系统调用号    |
| <abi>         | 应用二进制接口  |
| <name>        | 系统调用名    |
| <entry point> | 服务例程入口地址 |

应用二进制接口分为三种：64、x32和common，即三种不同的调用约定，这里不需考虑太多，三种任意选择一种即可，按照上述格式编写新的系统调用表表项如下：

```
335      64      first_compile      __x64_sys_first_compile
```

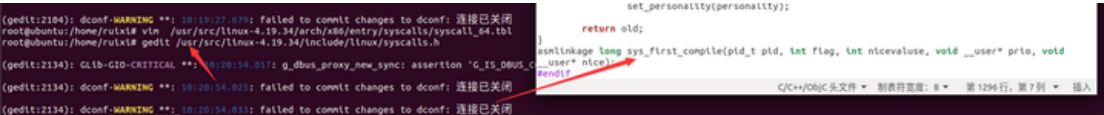


(2) 声明系统调用服务例程

查看系统调用头文件

```
gedit /usr/src/linux-4.19.25/include/linux/syscalls.h
```

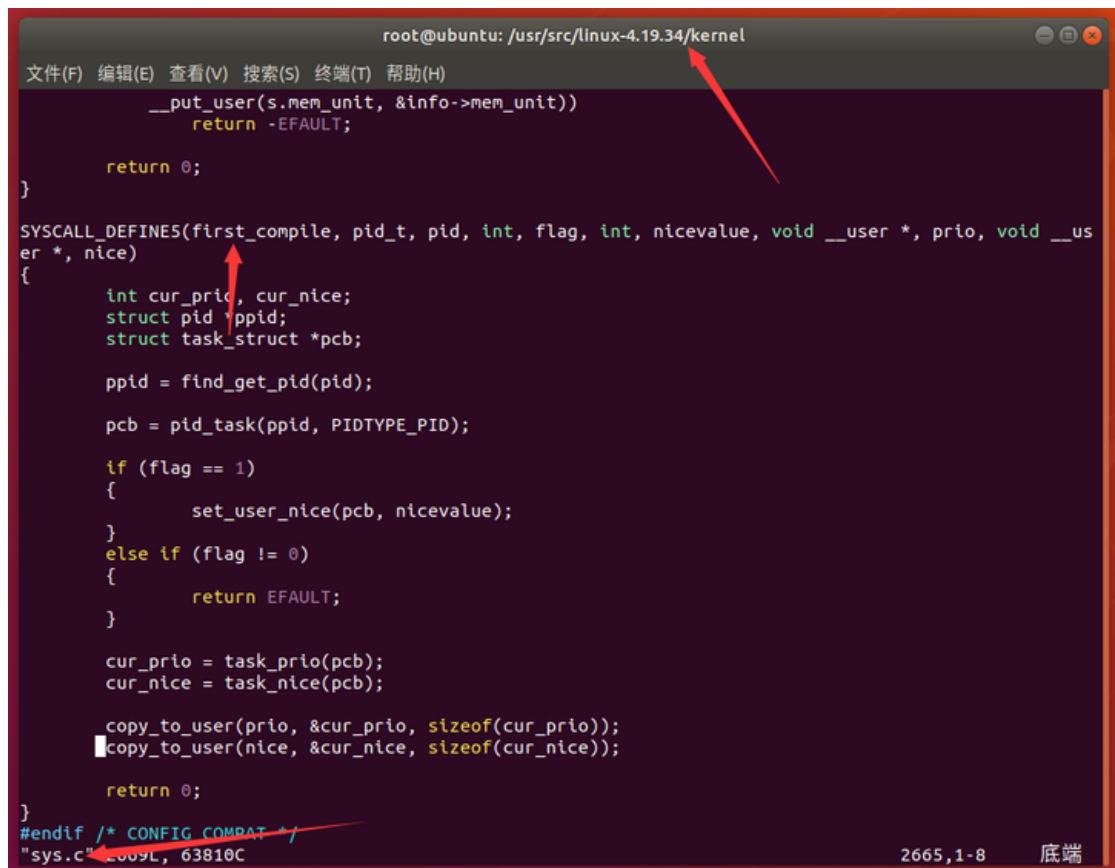
同样将linux-4.19.25换成你自己的版本即可。



(3) 实现自己的系统调用服务例程

首先进入解压后的文件目录，就是开始解压放入的目录，如下图：

```
$cd /usr/src/linux-4.19.34(换成自己的版本即可)/kernel
vim sys.c
```



```
root@ubuntu: /usr/src/linux-4.19.34/kernel
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
__put_user(s.mem_unit, &info->mem_unit))
    return -EFAULT;

    return 0;
}

SYSCALL_DEFINE5(first_compile, pid_t, pid, int, flag, int, nicevalue, void __user *, prio, void __user *, nice)
{
    int cur_prio, cur_nice;
    struct pid *ppid;
    struct task_struct *pcb;

    ppid = find_get_pid(pid);

    pcb = pid_task(ppid, PIDTYPE_PID);

    if (flag == 1)
    {
        set_user_nice(pcb, nicevalue);
    }
    else if (flag != 0)
    {
        return -EFAULT;
    }

    cur_prio = task_prio(pcb);
    cur_nice = task_nice(pcb);

    copy_to_user(prio, &cur_prio, sizeof(cur_prio));
    copy_to_user(nice, &cur_nice, sizeof(cur_nice));

    return 0;
}
#endif /* CONFIG_COMPAT */
"sys.c" 2665,1-8 底端
```

函数说明：

这一步与上一步的关系，就是C语言中头文件与实现文件的关系，上一步我们对函数进行了声明，这里给函数一个具体的实现。

首先要明确，我们要实现一个什么样的功能，根据内容要求可知，这个系统调用需要具备对指定进程的nice值的修改及读取的功能，同时返回进程最新的nice值及优先级prio。

把功能分拆成一个一个小块，我们需要做到的有以下几点：

根据进程号pid找到相应的进程控制块PCB（因为进程控制块中记录了用于描述进程情况及控制进程运行所需要的全部信息，nice值和优先级正是其中的一部分）；

根据PCB读取它的nice值和优先级prio；

根据PCB对相应进程的nice值进行修改；

将得到的nice值和优先级prio进行返回。

```
SYSCALL_DEFINE5(first_compile, pid_t, pid, int, flag, int, nicevalue, void __us
{
    int cur_prio, cur_nice;
    struct pid *ppid;
    struct task_struct *pcb;

    ppid = find_get_pid(pid);

    pcb = pid_task(ppid, PIDTYPE_PID);

    if (flag == 1)
    {
        set_user_nice(pcb, nicevalue);
    }
    else if (flag != 0)
    {
```

```
        return EFAULT;
    }

    cur_prio = task_prio(pcb);
    cur_nice = task_nice(pcb);

    copy_to_user(&prio, &cur_prio, sizeof(cur_prio));
    copy_to_user(&nice, &cur_nice, sizeof(cur_nice));

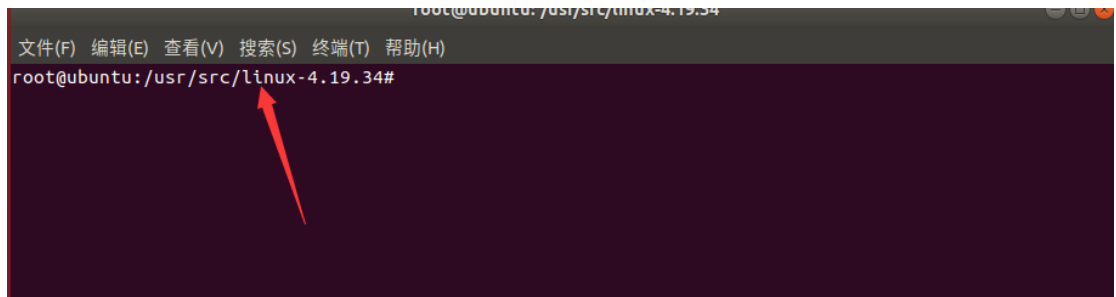
    return 0;
}
```

#### (4) 开始编译内核

首先，用下面这条命令查漏补缺，很有用处，用来它，我编译是一次通过的，没有遇见什么其他麻烦。

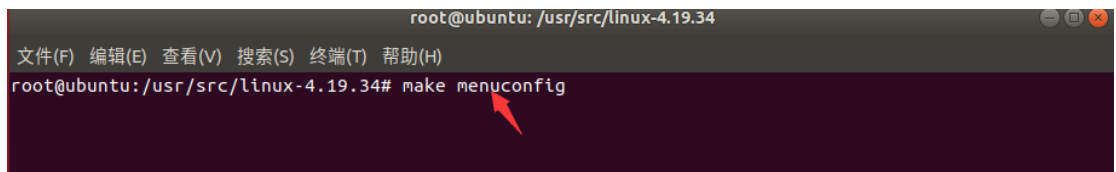
```
sudo apt-get install libncurses5-dev make openssl libssl-dev bison flex
```

然后定位到，源码在的目录，也就是解压后放的目录。

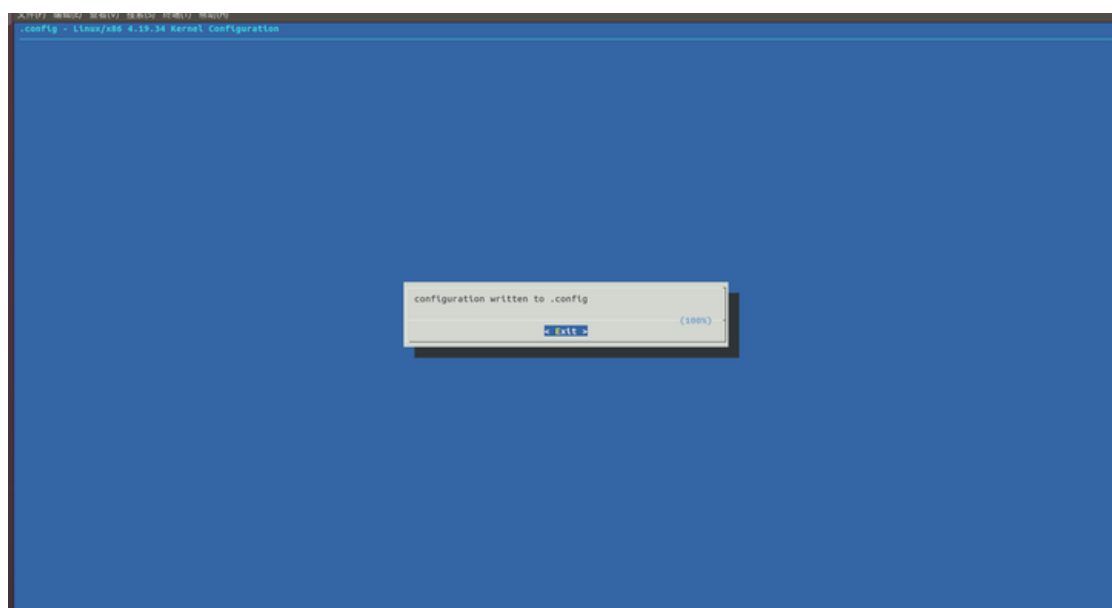
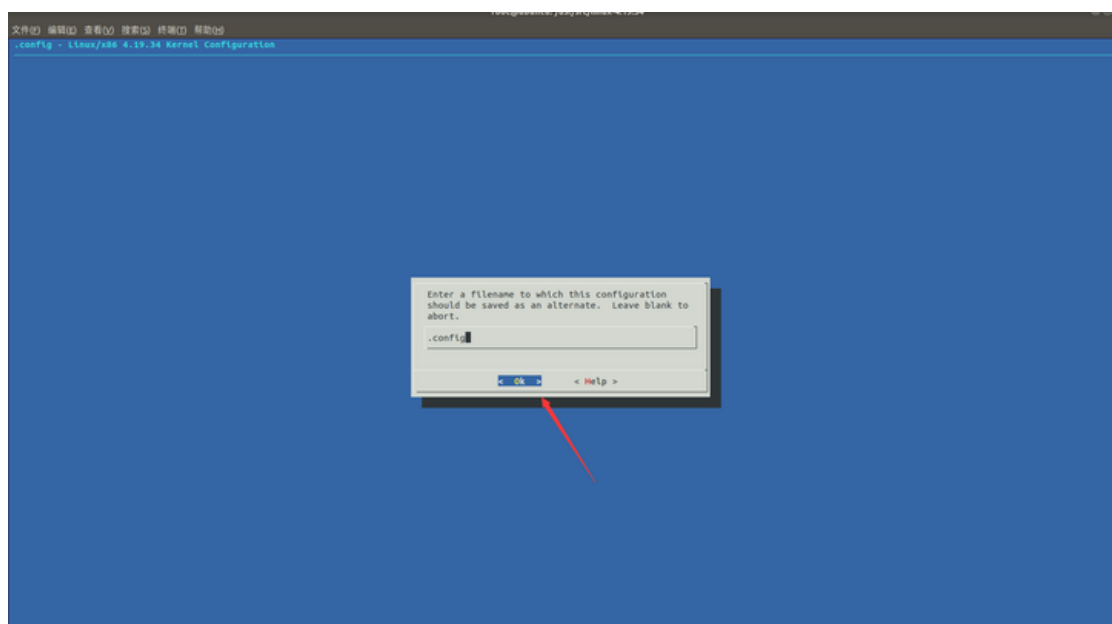
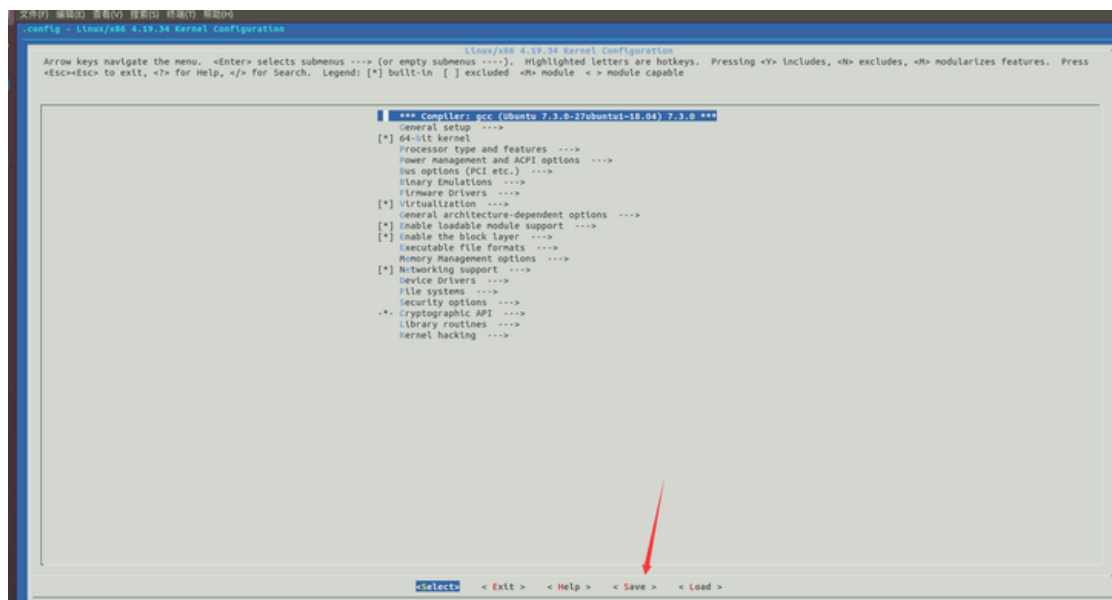


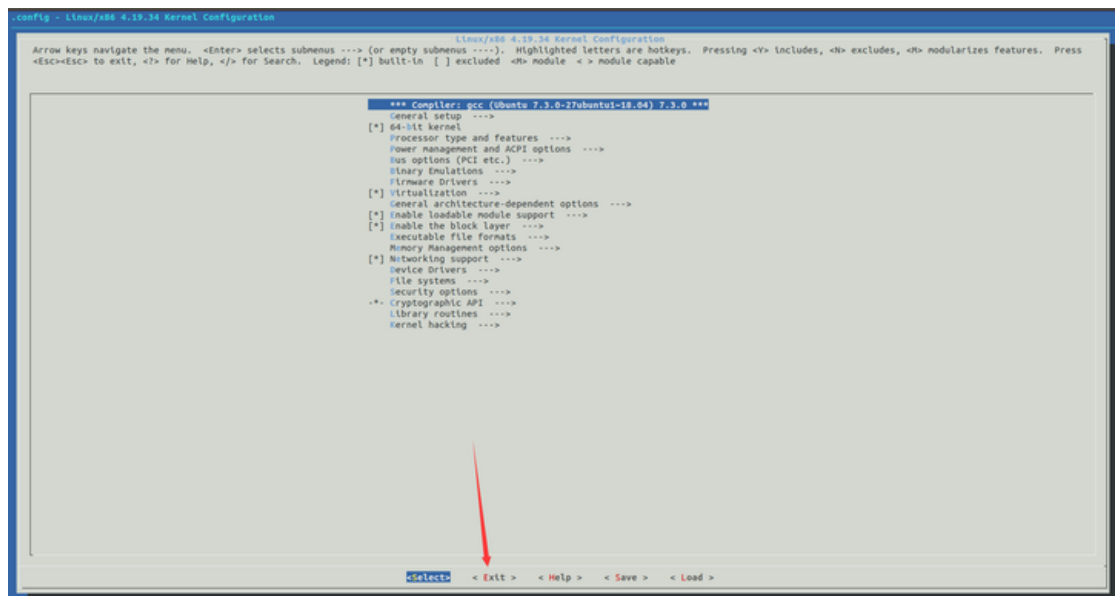
然后运行命令

```
make menuconfig
```



然后会出现以下界面，根据下面的图片所指的按钮来，通过左右键来确定光标停在选的按钮上，enter键是确定键





准备工作做好后，开始编译，耗时最长

```
sudo make -j4 2> error.log
```

-j4表示使用四线程进行编译，这个过程大概持续一个小时，后面的重定向将错误信息输出到了error.log这个文件里面，方便我们之后进行错误排查，不至于一两个小时坐在电脑面前盯着信息输出生怕出现一个错误而自己错过了，之后修改只能靠两眼排查，相信我，那不是一种好的体验。开始等待吧，结束后就可以安装内核了。

【文章福利】小编推荐自己的Linux内核技术交流群: **【1143996416】** 整理了一些个人觉得比较好的学习书籍、视频资料共享在群文件里面，有需要的可以自行添加哦!!!（含视频教程、电子书、实战项目及代码）

- |                                  |                               |  |
|----------------------------------|-------------------------------|--|
| 202101021 深入理解Nginx模块与架构解析       | 20210519 剖析Linux内核CFS调度器      | 20210908 剖析linux内核protocolsockets_k_buff |
| 202101027 高性能服务器《IO复用技术》详解       | 20210522 剖析Linux内核SMP负载均衡     | 20210911 剖析linux内核--内核互斥技术精髓             |
| 202101028 服务器通讯必备绝招详解            | 20210601 剖析Linux内核ARM异常处理     | 20210915 剖析Linux内核虚拟文件系统架构               |
| 202101107 深度解析《大厂经典面试题》          | 20210606 剖析Linux内核物理内存管理      | 20210916 剖析Linux内核源码IPC机制                |
| 202101108 高性能稳定服务器《两招绝杀》         | 20210612 剖析Linux内核页高速缓存及回写    | 20210917 剖析Linux内核高速缓存精髓                 |
| 202101112 高性能计算服务器: libevent和定时器 | 20210622 剖析Linux内核IPv4协议      | 20210918 剖析Linux内核内存分配与回收                |
| 202101121 面试大厂必考《预处理及内存管理》       | 20210624 Linux高性能服务器模型选择      | 20210920 剖析Linux内核设备驱动架构                 |
| 20210310 剖析Linux内核进程调度与切换        | 20210630 剖析Linux内核内存管理        | 20210921 剖析Linux内核Ext2_3文件系统             |
| 20210312 剖析Linux内核进程调度策略         | 20210703 剖析Linux内核分页机制        | 20210922 剖析Linux内核入门必修篇                  |
| 20210318 剖析Linux内核锁及进程间通信        | 20210708 剖析Linux内核高级路由选择      | 20210923 剖析Linux内核缓存及中断处理                |
| 20210326 剖析Linux内核缓存和刷新机制        | 20210722 剖析Linux内核蓝牙子系统架构     | 20210925 剖析Linux内核异常与中断处理                |
| 20210331 剖析Linux内核时钟机制及调度算法      | 20210809 剖析Linux内核CPU缓存技术     | 20210927 剖析Linux内核中断后半部处理机制              |
| 20210407 剖析Linux内核地址映射机制         | 20210815 剖析Linux内核内存回收        | 20210930 剖析Qt开发入门必备第一讲                   |
| 20210413 剖析Linux内核源码数据同步         | 20210818 剖析Linux内核页表缓存        | 20211004 剖析Linux内核后半部处理机制                |
| 20210417 剖析Linux内核四大核心框架         | 20210820 剖析Linux内核MMU机制       | 20211006 剖析Linux内核进程调度与切换                |
| 20210507 剖析Linux内核源码中断机制--       | 20210823 90分钟搞定DPDK技术精髓       | 20211009 剖析Qt跨平台GUI原理机制                  |
| 20210514 剖析Linux内核并发与同步          | 20210901 剖析Linux内核网络协议栈       | 课程配套资料下载地址                               |
| 20210517 剖析Linux内核MMU机制          | 20210905 剖析容器舵手Kubernetes设计架构 |  |

## (5) 安装内核

此时还是在你原来的目录路径下  
安装模块：

```
sudo make modules_install
```

使用这一行命令进行模块的安装，模块的安装持续时间大概在十几分钟左右，视你分配的资源多寡这个时间会适当地增加或减少。



结束后

安装内核：

```
sudo make install
```

使用这一行命令进行内核的安装，内核的安装持续时间大概是几分钟，视你分配的资源多寡这个时间会适当地增加或减少。

这一步完成且没有任何错误后，恭喜你，你已经完成了绝大多数的工作了，剩下的都是一些简单且容易调试的内容，重启你的电脑/虚拟机。

这个时候可以查看你的/**lib/module**目录下有无安装好的内核了。

```
root@ubuntu:/lib# cd modules/  
root@ubuntu:/lib/modules# ls  
4.18.0-15-generic 4.18.0-16-generic 4.19.34  
root@ubuntu:/lib/modules#
```

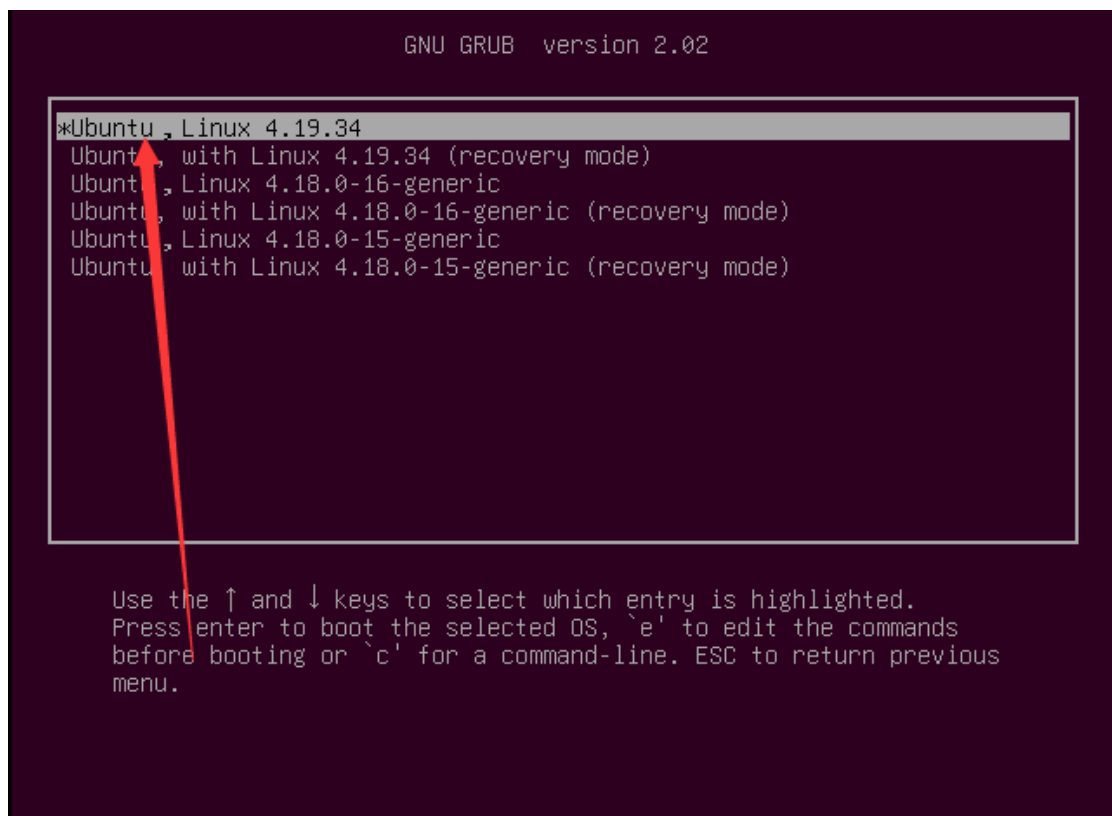
### (6) 重启系统

查看内核版本的命令，如下，你可以看自己现在的版本是否是新安装的内核

```
uname -a
```

```
test test.c  
root@ubuntu:/home/ts_17011328/Lab1# vim test.c  
root@ubuntu:/home/ts_17011328/Lab1# uname -a  
Linux ubuntu 4.19.34 #1 SMP Mon Apr 15 11:22:17 CST 2019 x86_64 x86_64 x86_64 GNU/Linux  
root@ubuntu:/home/ts_17011328/Lab1#
```

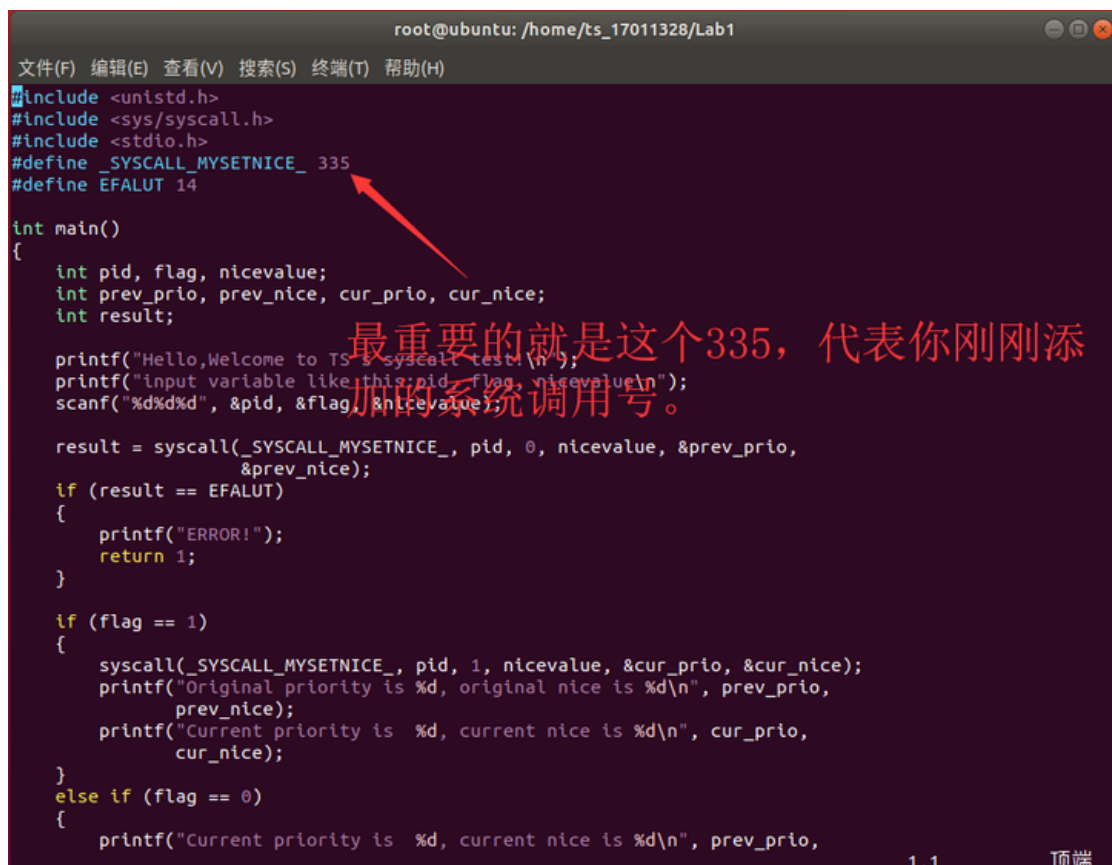
之后可能会弹出这个界面，选择你自己刚刚编译好的内核即可



### (7) 测试

自己选择一个文件夹，存放自己的测试代码,创建.c文件

```
vim test.c
```



```
#include <unistd.h>
#include <sys/syscall.h>
#include <stdio.h>
```

```

#define _SYSCALL_MYSETNICE_ 335
#define EFALUT 14

int main()
{
    int pid, flag, nicevalue;
    int prev_prio, prev_nice, cur_prio, cur_nice;
    int result;

    printf("Please input variable(pid, flag, nicevalue): ");
    scanf("%d%d%d", &pid, &flag, &nicevalue);

    result = syscall(_SYSCALL_MYSETNICE_, pid, 0, nicevalue, &prev_prio,
                    &prev_nice);
    if (result == EFALUT)
    {
        printf("ERROR!");
        return 1;
    }

    if (flag == 1)
    {
        syscall(_SYSCALL_MYSETNICE_, pid, 1, nicevalue, &cur_prio, &cur_nice);
        printf("Original priority is: [%d], original nice is [%d]\n", prev_prio,
              prev_nice);
        printf("Current priority is : [%d], current nice is [%d]\n", cur_prio,
              cur_nice);
    }
    else if (flag == 0)
    {
        printf("Current priority is : [%d], current nice is [%d]\n", prev_prio,
              prev_nice);
    }

    return 0;
}

```

之后使用gcc进行编译，根据要求输入对应的值。

结束了。到这一步，你就熟悉了过程了，对于编译和添加内核有个基本的了解了。

最后附上需要用到的内核源码截图。

Linux源码地址

第一张图和第二张图是一个函数set\_user\_nice()

```

void set_user_nice(struct task_struct *p, long nice)
{
    bool queued, running;
    int old_prio, delta;
    struct rq_flags rf;
    struct rq *rq;

    if (task_nice(p) == nice || nice < MIN_NICE || nice > MAX_NICE)
        return;

    /*
     * We have to be careful, if called from sys_setpriority(),
     * the task might be in the middle of scheduling on another CPU.
     */
    rq = task_rq_lock(p, &rf);
    update_rq_clock(rq);
}

```

```

/*
 * The RT priorities are set via sched_setscheduler(), but we still
 * allow the 'normal' nice value to be set - but as expected
 * it wont have any effect on scheduling until the task is
 * SCHED_DEADLINE, SCHED_FIFO or SCHED_RR:
 */
if (task_has_dl_policy(p) || task_has_rt_policy(p)) {
    p->static_prio = NICE_TO_PRIO(nice);
    goto out_unlock;
}
queued = task_on_rq_queued(p);
running = task_current(rq, p);
if (queued)
    dequeue_task(rq, p, DEQUEUE_SAVE | DEQUEUE_NOCLOCK);
if (running)
    put_prev_task(rq, p);

p->static_prio = NICE_TO_PRIO(nice);
set_load_weight(p, true);
old_prio = p->prio;
p->prio = effective_prio(p);
delta = p->prio - old_prio;

if (queued) {
    enqueue_task(rq, p, ENQUEUE_RESTORE | ENQUEUE_NOCLOCK);
    /*
     * If the task increased its priority or is running and
     * lowered its priority, then reschedule its CPU:
     */
    if (delta < 0 || (delta > 0 && task_running(rq, p)))
        resched_curr(rq);
}
if (running)
    set_curr_task(rq, p);
out_unlock:
    task_rq_unlock(rq, p, &rf);
}
EXPORT_SYMBOL(set_user_nice);

```

## task\_on\_rq\_queued()

```

static inline int task_on_rq_queued(struct task_struct *p)
{
    return p->on_rq == TASK_ON_RQ_QUEUED;
}

```

---

```

/* task_struct::on_rq states: */
#define TASK_ON_RQ_QUEUED 1
#define TASK_ON_RQ_MIGRATING 2

```

---

## task\_current()

```

static inline int task_current(struct rq *rq, struct task_struct *p)
{
    return rq->curr == p;
}

```

```

/*
 * Convert user-nice values [ -20 ... 0 ... 19 ]
 * to static priority [ MAX_RT_PRIO..MAX_PRIO-1 ],
 * and back.
 */
#define NICE_TO_PRIO(nice)      ((nice) + DEFAULT_PRIO)
#define PRIO_TO_NICE(prio)      ((prio) - DEFAULT_PRIO)

```

## effective\_prio()

```

/*
 * Calculate the current priority, i.e. the priority
 * taken into account by the scheduler. This value might
 * be boosted by RT tasks, or might be boosted by
 * interactivity modifiers. Will be RT if the task got
 * RT-boosted. If not then it returns p->normal_prio.
 */
static int effective_prio(struct task_struct *p)
{
    p->normal_prio = normal_prio(p);
    /*
     * If we are RT tasks or we were boosted to RT priority,
     * keep the priority unchanged. Otherwise, update priority
     * to the normal priority:
     */
    if (!rt_prio(p->prio))
        return p->normal_prio;
    return p->prio;
}

```

---

```

/*
 * Calculate the expected normal priority: i.e. priority
 * without taking RT-inheritance into account. Might be
 * boosted by interactivity modifiers. Changes upon fork,
 * setprio syscalls, and whenever the interactivity
 * estimator recalculates.
 */
static inline int normal_prio(struct task_struct *p)
{
    int prio;

    if (task_has_dl_policy(p))
        prio = MAX_DL_PRIO-1;
    else if (task_has_rt_policy(p))
        prio = MAX_RT_PRIO-1 - p->rt_priority;
    else
        prio = __normal_prio(p);
    return prio;
}

```

---

```

void update_rq_clock(struct rq *rq)
{
    s64 delta;

    lockdep_assert_held(&rq->lock);

    if (rq->clock_update_flags & RQCF_ACT_SKIP)
        return;

#ifdef CONFIG_SCHED_DEBUG
    if (sched_feat(WARN_DOUBLE_CLOCK))
        SCHED_WARN_ON(rq->clock_update_flags & RQCF_UPDATED);
    rq->clock_update_flags |= RQCF_UPDATED;
#endif

    delta = sched_clock_cpu(cpu_of(rq)) - rq->clock;
    if (delta < 0)
        return;
    rq->clock += delta;
    update_rq_clock_task(rq, delta);
}

```

---

```

static inline int task_has_rt_policy(struct task_struct *p)
{
    return rt_policy(p->policy);
}

static inline int task_has_dl_policy(struct task_struct *p)
{
    return dl_policy(p->policy);
}

```

---

```

static inline int task_nice(const struct task_struct *p)
{
    return PRIO_TO_NICE((p)->static_prio);
}

```

---

```

#define MAX_USER_RT_PRIO    100
#define MAX_RT_PRIO        MAX_USER_RT_PRIO

#define MAX_PRIO            (MAX_RT_PRIO + NICE_WIDTH)
#define DEFAULT_PRIO       (MAX_RT_PRIO + NICE_WIDTH / 2)

```

```
#define MAX_NICE      19
#define MIN_NICE     -20
#define NICE_WIDTH   (MAX_NICE - MIN_NICE + 1)

/*
 * Priority of a process goes from 0..MAX_PRIO-1, valid RT
 * priority is 0..MAX_RT_PRIO-1, and SCHED_NORMAL/SCHED_BATCH
 * tasks are in the range MAX_RT_PRIO..MAX_PRIO-1. Priority
 * values are inverted: lower p->prio value means higher priority.
 *
 * The MAX_USER_RT_PRIO value allows the actual maximum
 * RT priority to be separate from the value exported to
 * user-space. This allows kernel threads to set their
 * priority to a value higher than any user task. Note:
 * MAX_RT_PRIO must not be smaller than MAX_USER_RT_PRIO.
 */

#define MAX_USER_RT_PRIO      100
#define MAX_RT_PRIO          MAX_USER_RT_PRIO

#define MAX_PRIO              (MAX_RT_PRIO + NICE_WIDTH)
#define DEFAULT_PRIO         (MAX_RT_PRIO + NICE_WIDTH / 2)
```

来源; <https://www.cnblogs.com/tsruixi/p/10777242.html>

发布于 2022-02-20 22:26

Linux 内核

Linux

内核编译

写下你的评论...



还没有评论，发表第一个评论吧

推荐阅读



## 如何进行Linux内核编译 以及添加系统调用

会飞的鱼

## 怎么编译Linux内核？

1. Linux 内核介绍Linux内核（英语：Linux kernel）是一种开源的类Unix操作系统宏内核。整个Linux操作系统家族基于 该内核部署在传统计算机平台（如个人计算机和服务

器，以Linux发行版的形…

韦东山嵌入… 发表于韦东山嵌入…



## 如何编译 Linux 内核

Linux...

发表于Linux.