

LAB2

内核环境: 5.5.11-050511-generic

Task1 :

主要过程:

开启两个进程, 每个进程创建 5 个 cpu 密集线程, taskset 绑定 cpu 核, renice 调整优先级。(实际实现过程是启动进程时用 taskset 绑定 cpu 核, 后续用 renice 调整 nice 值) 用到的关键命令:

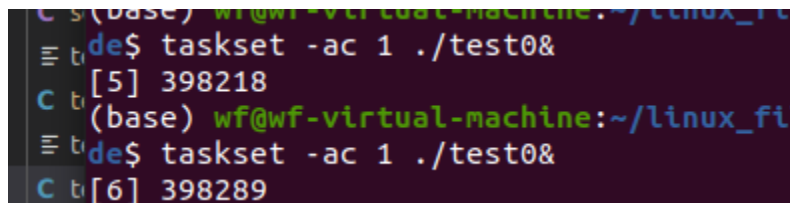
```
taskset -ac 1 ./test& //启动时绑定 (core 1)
taskset -pc 3 pid //启动后设置 (core 3)
taskset -c -p pid 查看绑定到哪个 cpu 核上
```

```
nice -n 5 gedit & //启动时绑定
renice -n 优先级数字 pid//启动后设置
nice 值-20 ~ 19
数字越小, 优先级越高
```

htop F5: 显示进程树

htop -p --pid=PID,PID... 只显示给定的 PIDs

过程和结果展示:



```
C s (base) wf@wf-virtual-machine:~/linux_fit
≡ t de$ taskset -ac 1 ./test0&
[5] 398218
C t (base) wf@wf-virtual-machine:~/linux_fit
≡ t de$ taskset -ac 1 ./test0&
C t [6] 398289
```

```

#include <pthread.h>

#include <unistd.h>

void *tfn()
{
    int a = 0;
    while (1)
    {
        a = a + 1;
        if (a == 1000)
            a = 0;
    }
}

int main(void)
{
    for (int i = 0; i < 5; i++)
    {
        pthread_t tid;

        pthread_create(&tid, NULL, tfn, NULL);

        // sleep(1);

        // printf("我是进程, 我的进程ID = %d\n", getpid());
    }
    while (1)
    {
    }
}

```

但是这样的实现有个小问题，主进程的 state 为 R, 这导致主进程和剩余的 5 个线程占用的 cpu 基本一致，这导致一共有 12 个 cpu 密集型进程 (while(1) 什么也不干也可以跑满 cpu)。

PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
398289	wf	15	-5	43620	620	536	R	70.9	0.0	7:10.30	./test0
398294	wf	15	-5	43620	620	536	R	11.7	0.0	1:10.33	./test0
398293	wf	15	-5	43620	620	536	R	11.7	0.0	1:05.36	./test0
398292	wf	15	-5	43620	620	536	R	11.7	0.0	1:05.35	./test0
398291	wf	15	-5	43620	620	536	R	11.7	0.0	1:05.36	./test0
398290	wf	15	-5	43620	620	536	R	11.7	0.0	0:57.91	./test0
398218	wf	19	-1	43620	616	536	R	29.9	0.0	3:50.02	./test0
398223	wf	19	-1	43620	616	536	R	5.9	0.0	0:42.19	./test0
398222	wf	19	-1	43620	616	536	R	5.2	0.0	0:42.18	./test0
398221	wf	19	-1	43620	616	536	R	5.2	0.0	0:42.18	./test0
398220	wf	19	-1	43620	616	536	R	4.6	0.0	0:42.18	./test0
398219	wf	19	-1	43620	616	536	R	4.6	0.0	0:42.17	./test0

因此需要让主进程睡眠：(state :S)

```

// printf("我是
}
while (1)
{
    sleep(1);
}

```

最终的形式：

```
de$ taskset -ac 1 ./test0&
[5] 423938
(base) wf@wf-virtual-machine:~/linux_files$
de$ taskset -ac 1 ./test0&
[6] 423957
```

```
(base) wf@wf-virtual-machine:~/linux_files/linux_exp/lab2$ taskset -c -p 423938
pid 423938 的当前亲和列表: 1
(base) wf@wf-virtual-machine:~/linux_files/linux_exp/lab2$ taskset -c -p 423957
pid 423957 的当前亲和列表: 1
```

PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
423957	wf	15	-5	43620	612	532	S	71.8	0.0	3:02.46	./test0
423962	wf	15	-5	43620	612	532	R	14.6	0.0	0:36.49	./test0
423961	wf	15	-5	43620	612	532	R	14.0	0.0	0:36.48	./test0
423960	wf	15	-5	43620	612	532	R	14.6	0.0	0:36.48	./test0
423959	wf	15	-5	43620	612	532	R	14.0	0.0	0:36.48	./test0
423958	wf	15	-5	43620	612	532	R	14.0	0.0	0:36.48	./test0
423938	wf	19	-1	43620	612	532	S	29.2	0.0	2:57.53	./test0
423943	wf	19	-1	43620	612	532	R	6.0	0.0	0:35.50	./test0
423942	wf	19	-1	43620	612	532	R	6.6	0.0	0:35.50	./test0
423941	wf	19	-1	43620	612	532	R	6.0	0.0	0:35.49	./test0
423940	wf	19	-1	43620	612	532	R	6.0	0.0	0:35.50	./test0
423939	wf	19	-1	43620	612	532	R	6.0	0.0	0:35.49	./test0

由于实现实时进程时 vscode 终端崩溃导致无法操作，重启并重新启动 10 个线程：

PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
3593	wf	22	2	43628	616	536	S	71.7	0.0	1:56.49	./task1_ord
3598	wf	22	2	43628	616	536	R	14.7	0.0	0:23.29	./task1_ord
3597	wf	22	2	43628	616	536	R	14.7	0.0	0:23.29	./task1_ord
3596	wf	22	2	43628	616	536	R	14.1	0.0	0:23.28	./task1_ord
3595	wf	22	2	43628	616	536	R	14.1	0.0	0:23.28	./task1_ord
3594	wf	22	2	43628	616	536	R	14.7	0.0	0:23.29	./task1_ord
3587	wf	26	6	43628	620	536	S	29.5	0.0	1:54.46	./task1_ord
3592	wf	26	6	43628	620	536	R	6.0	0.0	0:22.88	./task1_ord
3591	wf	26	6	43628	620	536	R	6.0	0.0	0:22.89	./task1_ord
3590	wf	26	6	43628	620	536	R	6.0	0.0	0:22.89	./task1_ord
3589	wf	26	6	43628	620	536	R	6.0	0.0	0:22.88	./task1_ord
3588	wf	26	6	43628	620	536	R	5.4	0.0	0:22.88	./task1_ord

启动实时进程(pid 3799):

```

C task1_rt.c > main(void)
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4  #include <sched.h>
5
6  void main(void)
7  {
8      pid_t pid = getpid();
9      struct sched_param param;
10     param.sched_priority = sched_get_priority_max(SCHED_FIFO); // 也可用
        SCHED_RR
11     sched_setscheduler(pid, SCHED_RR, &param); // 设置当前进程
12     pthread_setschedparam(pthread_self(), SCHED_FIFO, &param); // 设置当前线程
13     int a = 0;
14     char pid1[10];
15     printf("pid=%d\n", getpid());
16     while (1)
17     {
18         // a = 0;
19         // a = a + 1;
20         // if (a == 1000)
21         // a = 0;
22     }
23 }

```

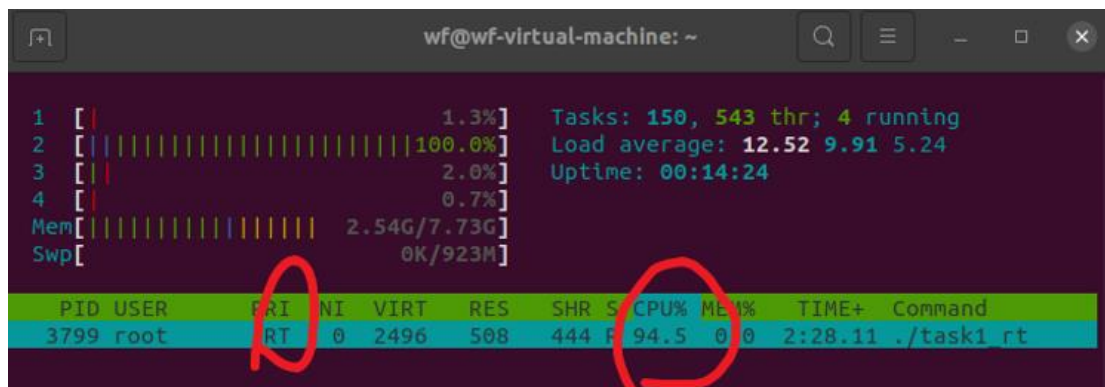
(FIFO,RR: 两种常用的实时调度算法)

并切换到 core 1:

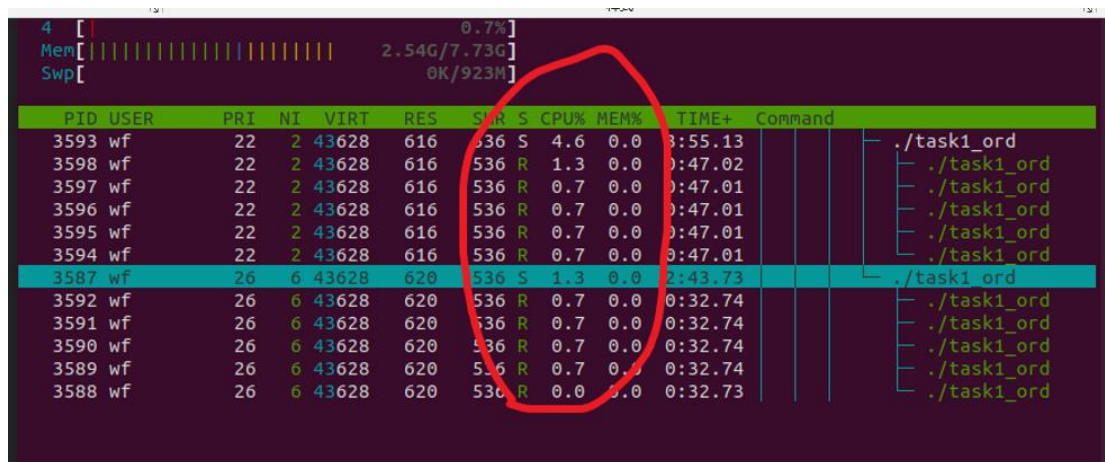
```

wf@wf-virtual-machine: ~
(base) wf@wf-virtual-machine:~$ ^C
(base) wf@wf-virtual-machine:~$ sudo taskset -p 1 3500^C
(base) wf@wf-virtual-machine:~$ taskset -c -p 3500
pid 3500 的当前亲和力列表: 0
(base) wf@wf-virtual-machine:~$ taskset -cp 0 3500
pid 3500 的当前亲和力列表: 0
taskset: 设置 pid 3500 的亲和力失败: 不允许的操作
(base) wf@wf-virtual-machine:~$ sudo taskset -cp 0 3500
pid 3500 的当前亲和力列表: 0
pid 3500 的新亲和力列表: 0
(base) wf@wf-virtual-machine:~$ taskset -c -p 3500
pid 3500 的当前亲和力列表: 0
(base) wf@wf-virtual-machine:~$ sudo taskset -cp 1 3500
pid 3500 的当前亲和力列表: 0
pid 3500 的新亲和力列表: 1
(base) wf@wf-virtual-machine:~$ ^C
(base) wf@wf-virtual-machine:~$ taskset -c -p 3799
pid 3799 的当前亲和力列表: 0-3
(base) wf@wf-virtual-machine:~$ sudo taskset -cp 1 3799
pid 3799 的当前亲和力列表: 0-3
pid 3799 的新亲和力列表: 1
(base) wf@wf-virtual-machine:~$

```



与此同时发现 10 个线程几乎没有 cpu 资源了：



可以看到实现了实时进程抢占普通进程 cpu 的效果。

但是切换后虚拟机非常的卡，以至于甚至要 ctrl+C 点击很快才能终止实时进程，终止后恢复正常，没有把实时进程只绑定到 core1 之前不卡，不太清楚原因，可能是核 1 不太开心。

Task2:

1. 进程管理:

Linux 内核通过进程描述符（Process Descriptor）来管理进程，相应的数据结构是 task_struct，它定义在 include/linux/sched.h 中。在第 673 行添加数据成员 ctx，如下所示：

```

672  #endif
673  //declare ctx here
674  int → → → ctx;
675

```

2. 进程创建

Linux 内核进程创建实质是对父进程的复制，源码位于 kernel/fork.c 中。

查找系统调用 fork()，调用 _do_fork 函数，如下：

```

2514  ✓ #ifdef __ARCH_WANT_SYS_FORK
2515  SYSCALL_DEFINE0(fork)
2516  {
2517  #ifdef CONFIG_MMU
2518  → struct kernel_clone_args args = {
2519  →   .exit_signal = SIGCHLD,
2520  → };
2521
2522  → return _do_fork(&args);
2523  #else
2524  → /* can not support in nommu mode */
2525  → return -EINVAL;
2526  #endif
2527  }
2528  #endif

```

_do_fork 中的 copy_process 是核心复制函数：

```

2423
2424  → p = copy_process(NULL, trace, NUMA_NO_NODE, args);
2425  → add_latent_entropy();

```

里面定义了 task_struct *p:

```

1830  → inc_prio = -1, retval,
1831  → struct task_struct *p;
1832  → struct multi_mmap_state;

```

对 p 初始化。

```

1911  → p = dup_task_struct(current, node);
1912  → if (!p)

```

: 复制父进程的内容到

p。

因此对子进程 ctx 初始化应该是在第 1911 行之后，第 2041 行开始初始化子进程的调度策略、优先级、调度类等进程调度相关成员，并接着复制父进程的信息（文件、信号、内存等）。此处为进程初始化的代码段，将 ctx 在此处初始化是合理的，所以在第 2041 行添加 ctx 初始化语句如下：

```

2041  → // initialize ctx here
2042  → p->ctx = 0;

```

3. 进程调度

Linux 内核关于进程调度的源码在 kernel/sched/core.c 中，所有的调度都发生在 schedule() 函数中。找到 schedule() 函数，位于第 4153 行。每当进程被调度，这个函数会被执行，因为进程每得到一次调度会执行 ctx++ 操作，所以直接在 schedule() 函数中添加即可。

```

4153
4154  asmlinkage __visible void __sched schedule(void)
4155  {
4156      struct task_struct *tsk = current;
4157
4158      // increment ctx here
4159      tsk->ctx++;
4160
4161      sched_submit_work(tsk);

```

4. 创建 proc 文件

每个进程都在 /proc 下有自己的目录 /proc/<PID>，目录内文件或文件夹的创建源码位于

fs/proc/base.c 中。每个进程文件夹下所有文件的静态列表定义在数组 tgid_base_stuff[] 中，各元素类型为 pid_entry。在 /proc/<PID> 目录下创建一个文件，则需要在这个静态常量数组中增加一项。查找相关资料得知：DIR 创建目录，LNK 创建链接，REG 和 ONE 均可创建文件，REG 传入完整的文件操作，ONE 只有读操作。而此处只需要创建一个可读文件，读取 ctx 的值，所以使用 ONE，添加代码如下：

```

3026  //create proc/pid/ctx |
3027  ONE("ctx", S_IRUSR, proc_pid_ctx),
3028

```

这实现了 proc 中有 ctx 这个伪文件，权限只读，而访问 ctx 文件的时候调用 proc_pid_ctx 函数，它应该实现将 ctx 打印在屏幕上的功能，如下，通过 seq_printf 函数将 ctx 的值打印到用户空间，也就是 shell 里。

```

3087  static int proc_pid_ctx(struct seq_file *m, struct pid_namespace *ns,
3088                          struct pid *pid, struct task_struct *task)
3089  {
3090      int err = lock_trace(task);
3091      if (!err)
3092      {
3093          seq_printf(m, "%d\n", task->ctx);
3094          unlock_trace(task);
3095      }
3096      return err;
3097  }

```

实现效果：

```

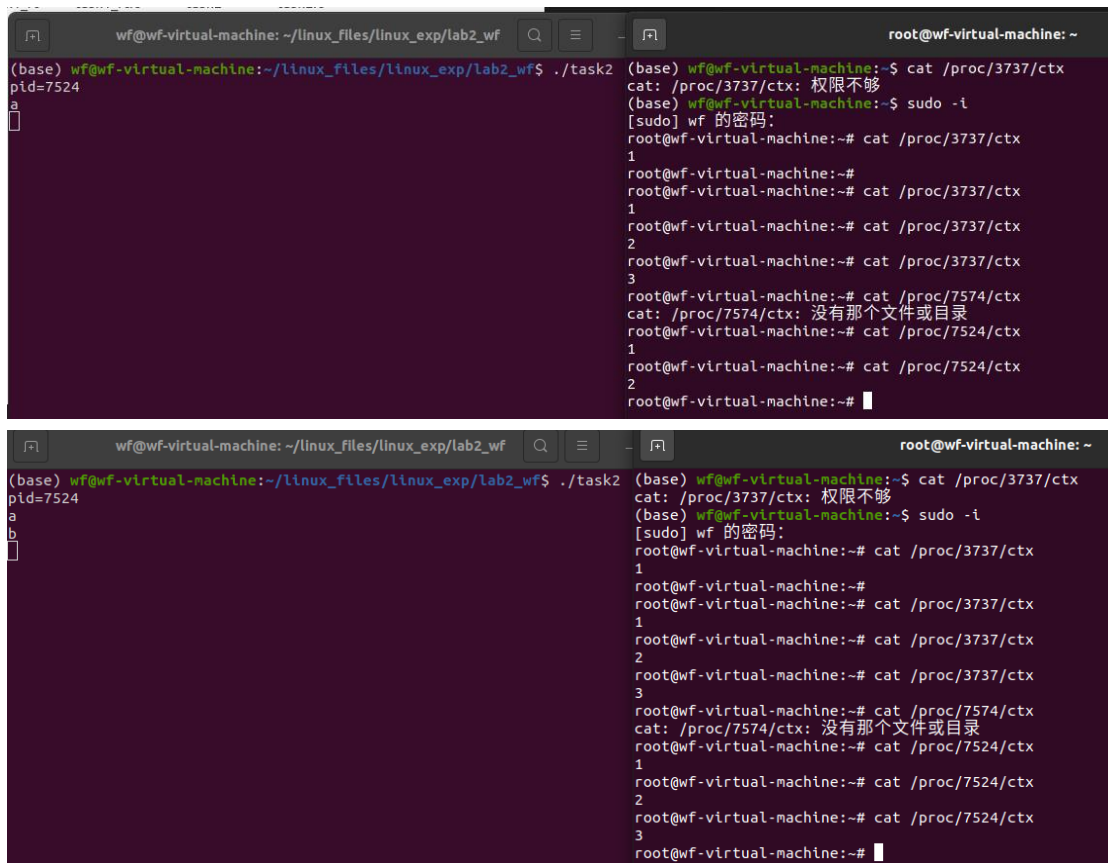
wf@wf-virtual-machine: ~/linux_files/linux_exp/lab2_wf
(base) wf@wf-virtual-machine:~/linux_files/linux_exp/lab2_wf$ ./task2
pid=7524

root@wf-virtual-machine: ~
(base) wf@wf-virtual-machine:~$ cat /proc/3737/ctx
cat: /proc/3737/ctx: 权限不够
(base) wf@wf-virtual-machine:~$ sudo -i
[sudo] wf 的密码:
root@wf-virtual-machine:~# cat /proc/3737/ctx
1
root@wf-virtual-machine:~# cat /proc/3737/ctx
1
root@wf-virtual-machine:~# cat /proc/3737/ctx
2
root@wf-virtual-machine:~# cat /proc/3737/ctx
3
root@wf-virtual-machine:~# cat /proc/7574/ctx
cat: /proc/7574/ctx: 没有那个文件或目录
root@wf-virtual-machine:~# cat /proc/7524/ctx
1
root@wf-virtual-machine:~#

```

(注意到是从 1 开始的，而不是我写的初始化 0，这是因为进程启动后就运行了一次

schedule 函数，加了 1)



```
(base) wf@wf-virtual-machine:~/linux_files/linux_exp/lab2_wf$ ./task2
pid=7524
a
b

(base) wf@wf-virtual-machine:~/linux_files/linux_exp/lab2_wf$ cat /proc/3737/ctx
cat: /proc/3737/ctx: 权限不够
(base) wf@wf-virtual-machine:~/linux_files/linux_exp/lab2_wf$ sudo -i
[sudo] wf 的密码:
root@wf-virtual-machine:~# cat /proc/3737/ctx
1
root@wf-virtual-machine:~# cat /proc/3737/ctx
1
root@wf-virtual-machine:~# cat /proc/3737/ctx
2
root@wf-virtual-machine:~# cat /proc/3737/ctx
3
root@wf-virtual-machine:~# cat /proc/7574/ctx
cat: /proc/7574/ctx: 没有那个文件或目录
root@wf-virtual-machine:~# cat /proc/7524/ctx
1
root@wf-virtual-machine:~# cat /proc/7524/ctx
2
root@wf-virtual-machine:~#
```

遇到的主要问题以及解决办法：

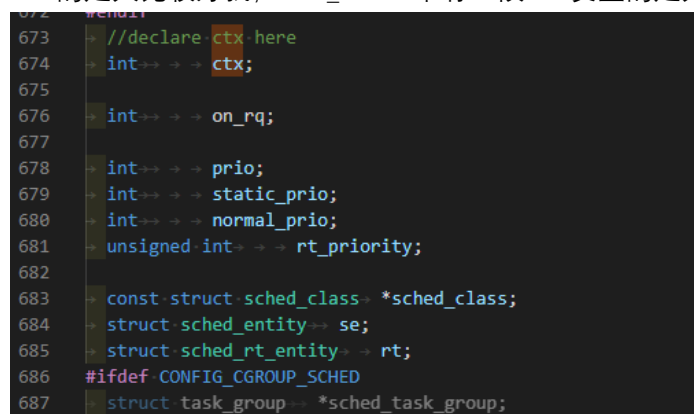
Task1:

这个实现不难，主要是虚拟机的 vscode shell 可能有问题，偶尔会出现终端白色字体和背景色一样的情况，后续主要使用系统自带的 shell。

Task2:

添加的位置问题：

Ctx 的定义比较好找，task_struct 中有一段 int 变量的定义，写在这里即可。



```
673 //declare ctx here
674 int ctx;
675
676 int on_rq;
677
678 int prio;
679 int static_prio;
680 int normal_prio;
681 unsigned int rt_priority;
682
683 const struct sched_class *sched_class;
684 struct sched_entity se;
685 struct sched_rt_entity rt;
686 #ifdef CONFIG_CGROUP_SCHED
687 struct task_group *sched_task_group;
```

自增也比较简单，直接写在 schedule 函数中即可。

主要是在哪里给 ctx 赋值为零，应该是在 fork 初始化各个变量的时候，确定了在

dup_task_struct 函数之后，在其之后有很多配置修改 p 中的参数：

```
005     #endif
006     #ifdef CONFIG_CPUSETS
007     + p->cpuset_mem_spread_rotor = NUMA_NO_NODE;
008     + p->cpuset_slab_spread_rotor = NUMA_NO_NODE;
009     + seqcount_init(&p->mems_allowed_seq);
010     #endif
011     #ifdef CONFIG_TRACE_IRQFLAGS
012     + p->irq_events = 0;
013     + p->hardirqs_enabled = 0;
014     + p->hardirq_enable_ip = 0;
015     + p->hardirq_enable_event = 0;
016     + p->hardirq_disable_ip = _THIS_IP_;
017     + p->hardirq_disable_event = 0;
018     + p->softirqs_enabled = 1;
019     + p->softirq_enable_ip = _THIS_IP_;
020     + p->softirq_enable_event = 0;
021     + p->softirq_disable_ip = 0;
022     + p->softirq_disable_event = 0;
023     + p->hardirq_context = 0;
024     + p->softirq_context = 0;
025     #endif
026
```

一个比较自然的想法是在所有这些值赋值完毕后进行 ctx 的初始化，避免干扰。而之后的函数是用 p 做入参，sched_fork 等，用来初始化子进程的调度策略、优先级等，因此写在这里比较合理。

黑屏问题：make_install 写入了一半 root 没有空间了，打算关机扩展空间，但是重启黑屏，最终通过以下方式解决：reboot->shift->advanced options->recover mode->shell->rm -rf lib/modules/5.5.11 即可。

总结：

这次实验基于 Linux 内核进程理论课，通过实操源码，对 task_struct 有了进一步的理解。

Task1 用创建各个进程的方式来验证普通进程间的优先级，实时进程和普通进程的优先级等理论知识。

Task2 本身修改的代码不多，但需要阅读 Linux 内核的几个源码文件，理解代码的执行顺序，才能在合适的位置插入代码。这一过程中，我提高了我阅读分析大规模系统软件源码的能力和 C 语言的编程规范。在修改完代码之后，重新编译需要几个小时的时间，我编译了三次才成功完成实验。总之，这次实验相比实验一，更侧重于阅读源码，提高了自己阅读源码的能力，更深刻感受到了内核代码的繁多复杂。

一些命令：

htop -d 1(-d 刷新闻隔)

Ps aux | grep test (类似显示进程名称，过滤作用)

sudo apt--get install linux-source 安装内核源码，系统不自带

sudo cp -v /boot/config-\$(shell uname-r).config

du-h-max-depth=1 查看文件大小

linux-headers-2.6.31-14 是 linux 代码里面头文件。linux-headers-2.6.31-14-generic 是 Elinux 内核文件。

Make-c(change 改变工作目录，-m 回到初始调用的 pwd 目录

make-j cpu 数量之类的，不加的话默认用所有的

Sudo-i 提升终端权限

make 不加参数包括 make modules , make modules_install 复制一份到/lib/modules, make install 将程序安装至系统中。执行顺序 make-> make modules_install-> make install

内核代码中很多关于宏的操作，https://blog.csdn.net/qg_36662437/article/details/81476572，讲解的很详细。

遗留问题：

```
1  #include <stdio.h>
2
3  int main()
4  {
5      while (1)
6          getchar();
7      return 0;
8  }
```

代码用&后台运行立刻停止 (top:state:s)。

切换后虚拟机非常的卡，以至于甚至要 ctrl+C 点击很快才能终止实时进程，终止后回复正常，没有把实时进程只绑定到 core1 之前不卡，不太清楚原因。