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

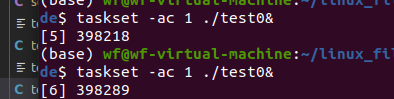
nice值-20～19

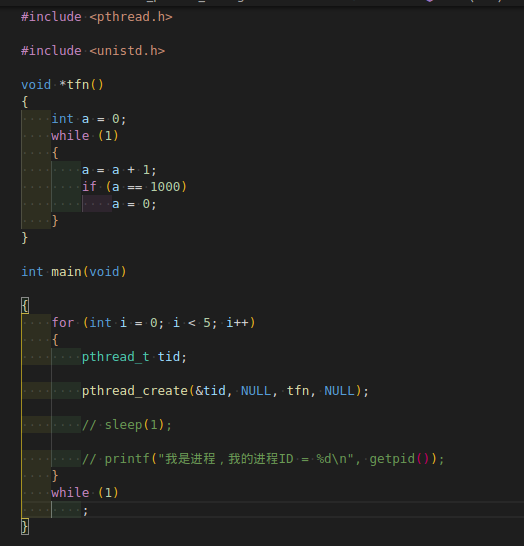
数字越小，优先级越高

htop F5：显示进程树

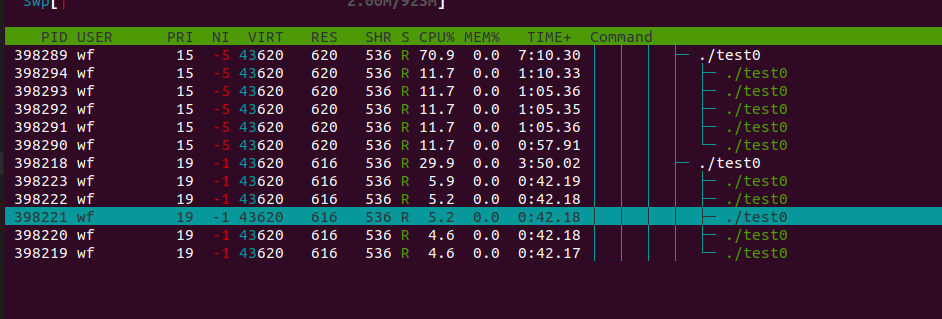
htop -p --pid=PID,PID…　　　 只显示给定的PIDs

过程和结果展示：

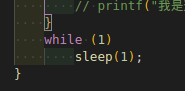




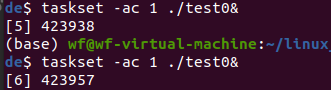
但是这样的实现有个小问题，主进程的state为R,这导致主进程和剩余的5个线程占用的cpu基本一致，这导致一共有12个cpu密集型进程（while（1）什么也不干也可以跑满cpu）。

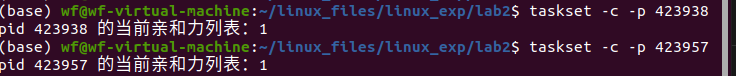


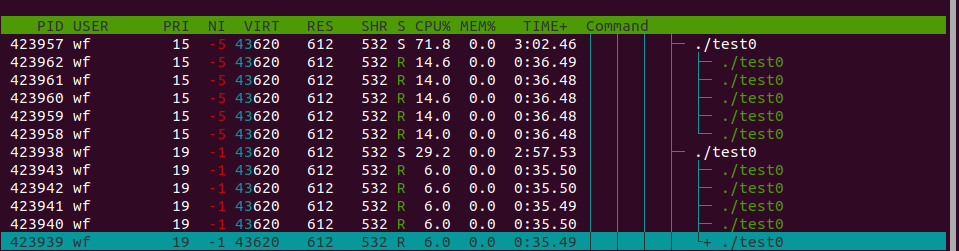
因此需要让主进程睡眠：（state :S）



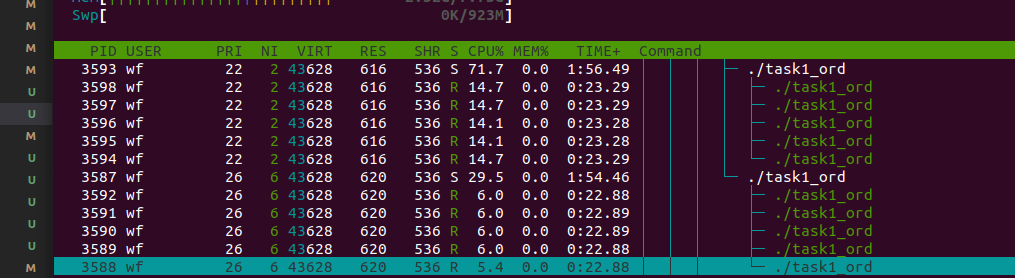
最终的形式：



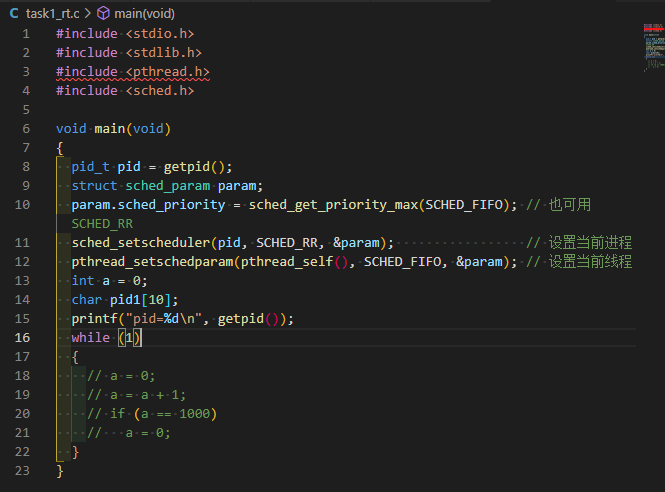




由于实现实时进程时vscode终端崩溃导致无法操作，重启并重新启动10个线程:

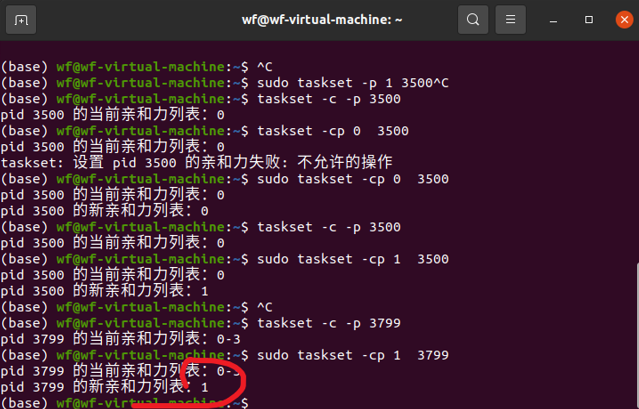


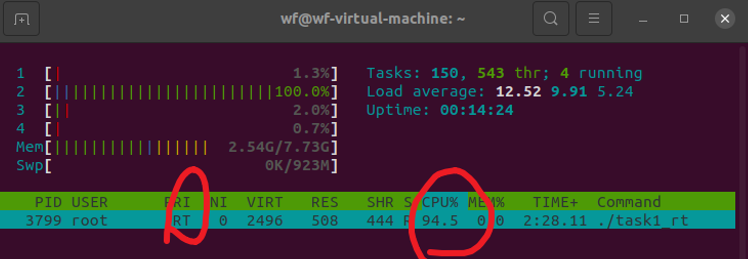
启动实时进程(pid 3799)：



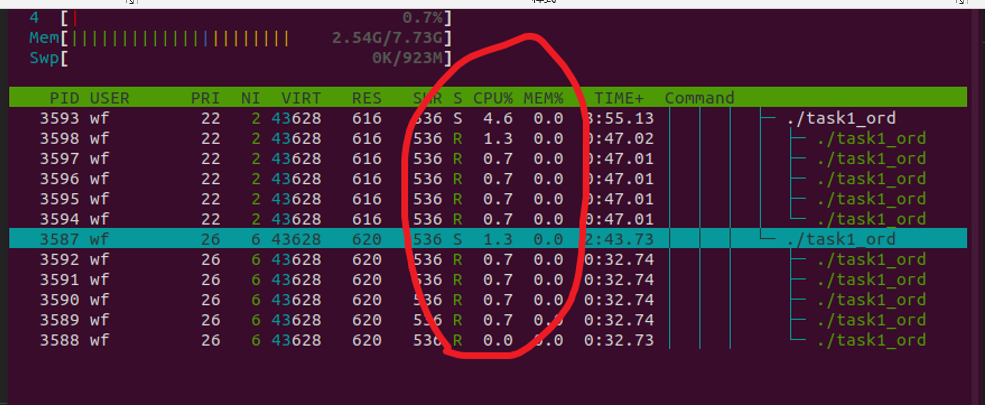
（FIFO,RR：两种常用的实时调度算法）

并切换到core 1：





与此同时发现10个线程几乎没有cpu资源了：



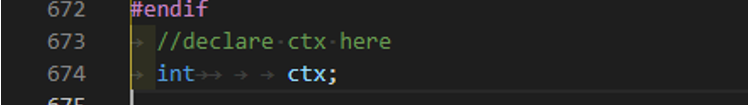
可以看到实现了实时进程抢占普通进程cpu的效果。

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， 可能是核1不太开心。

Task2:

1. 进程管理:

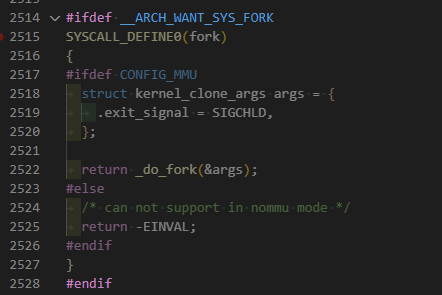
Linux 内核通过进程描述符（Process Descriptor）来管理进程，相应的数据结构是 task\_struct ，它定义在 include/linux/sched.h 中。在第 673 行添加数据成员 ctx ，如下所示：



2. 进程创建

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

查找系统调用fork()，调用\_do\_fork函数，如下：



\_do\_fork中的copy\_process是核心复制函数：



里面定义了task\_struct \*p:

：对p初始化。

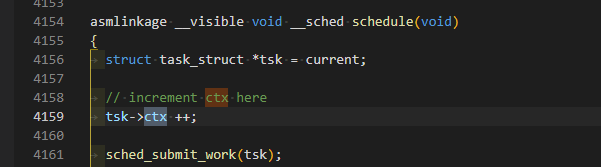
：复制父进程的内容到p。

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:



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。



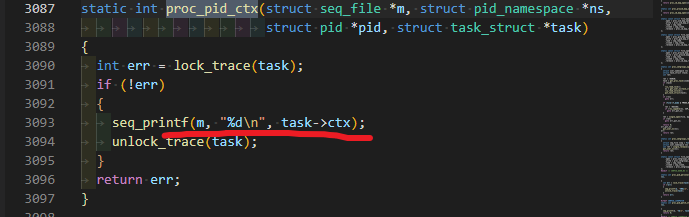
4．创建 proc文件

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

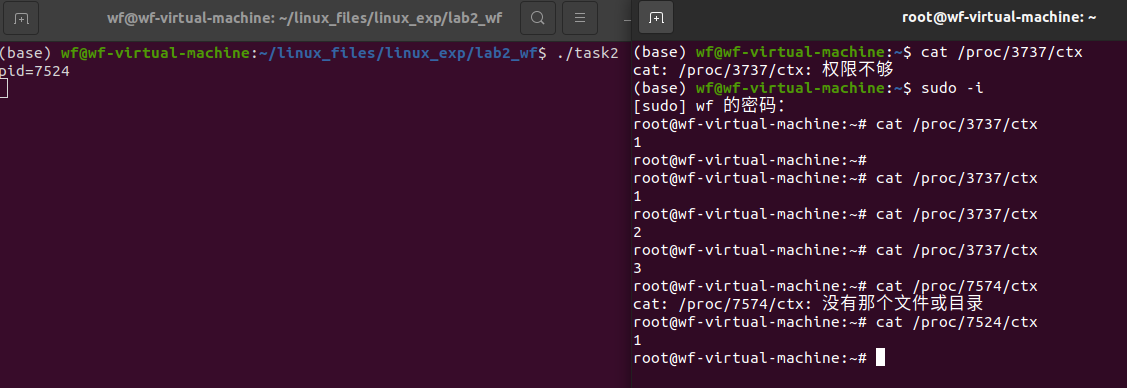
fs/proc/base.c 中。每个进程文件夹下所有文件的静态列表定义在数组 tgid\_base\_stuff[] 中，各元素类型为 pid\_entry 。在 /proc/<PID> 目录下创建一个文件，则需要在这个静态常量数组中增加一项。查找相关资料得知： DIR 创建目录， LNK 创建链接， REG 和 ONE 均可创建文件， REG 传入完整的文件操作， ONE 只有读操作。而此处只需要创建一个可读文件，读取 ctx 的值，所以使用 ONE ,添加代码如下:

=

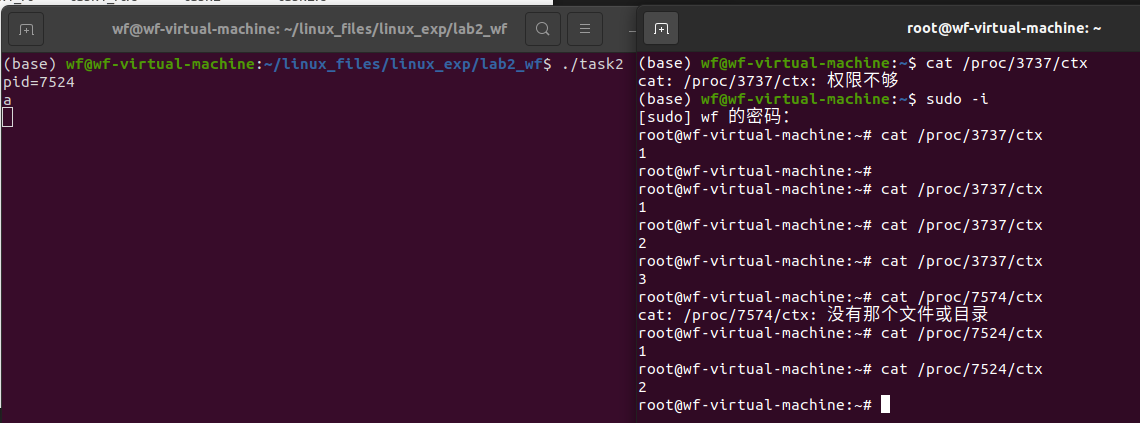
这实现了proc中有ctx这个伪文件，权限只读，而访问ctx文件的时候调用proc\_pid\_ctx函数，它应该实现将 ctx 打印在屏幕上的功能，如下，通过seq\_printf函数将ctx的值打印到用户空间，也就是shell里。

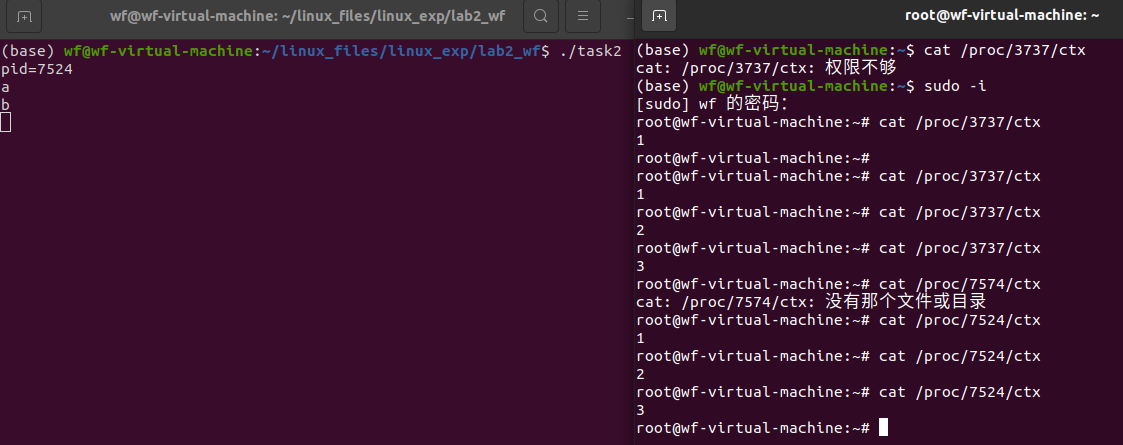


实现效果：



（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）





遇到的主要问题以及解决办法：

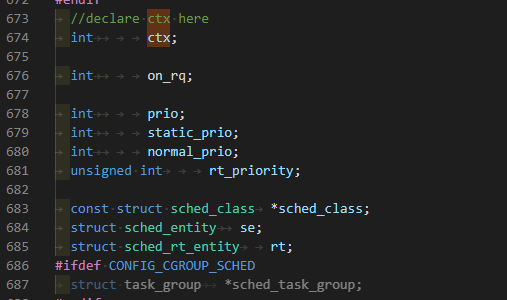
Task1：

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。



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：



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一些 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 刷新间隔)

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

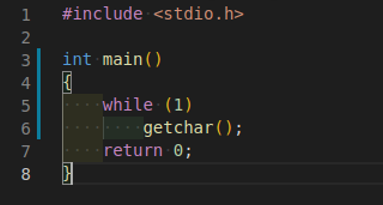
make-j cpu数量之类的，不加的话默认用所有的

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/qq_36662437/article/details/81476572>，讲解的很详细。

遗留问题：



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。