## 实验一报告：Linux内核编译及添加系统调用

### 一、实验目的

Linux是开源操作系统，用户可以根据自身系统需要裁剪、修改内核，定制出功能更加合适、运行效率更高的系统，因此，编译Linux内核是进行内核开发的必要基本功。在系统中根据需要添加新的系统调用是修改内核的一种常用手段，通过本次实验，学生应理解Linux系统处理系统调用的流

### 二、实验内容

* （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 prio。建议调用原型为:
* int mysetnice(pid\_t pid, int flag, int nicevalue, void \_\_user \* prio, void \_\_user \* nice)
* 参数含义：
* pid:进程 ID。  
  nicevalue：为指定进程设置的新nice值。  
  flag:若值为 0,表示读取 nice 值;若值为 1,表示修改 nice 值。  
  prio、nice:进程当前优先级及 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)若程序中调用了Linux的内核函数,要求深入阅读相关函数源码。

### 三、设计方案

这个系统调用需要对指定进程的nice值进行修改和读取，同时也要返回进程最新的nice值及优先级prio，分成以下功能：

* 根据进程索引pid找到对应的进程控制块PCB
* 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进程返回

### 四、实验过程

* **添加系统调用号**
* arch/x86/entry/syscalls/syscall\_64.tbl

|  |  |  |  |
| --- | --- | --- | --- |
| * 系统调用号 | * 应用二进制接口 | * 系统调用名称 | * 服务例程入口 |
| * 333 | * 64 | * mysetnice | * sys\_mysetnice |

* **系统调用原型**
* asmlinkage long sys\_mysetnice(pid\_t pid, int flag, int nicevalue, void \_\_user\* prio, void \_\_user\* nice);
* **系统调用服务例程**
* SYSCALL\_DEFINE5(mysetnice, pid\_t, pid, int, flag, int, nicevalue, void \_\_user\*, prio, void \_\_user\*, nice)  
  {  
   struct pid \*ppid;   
   //进程描述符指针，指向一个枚举类型  
   struct task\_struct \*task; //任务描述符信息  
   ppid = find\_get\_pid(pid); //通过索引pid\_t返回pid  
   task = pid\_task(ppid, PIDTYPE\_PID); //通过pid返回进程task  
    
   int curr\_nice;  
   curr\_nice = task\_nice(task); //返回当前进程的nice值  
   int curr\_prio;  
   curr\_prio = task\_prio(task); //返回当前进程的prio值  
    
   if(flag == 1){ //如果flag等于1修改进程的nice值   
   set\_user\_nice(task, nicevalue);  
   curr\_nice = task\_nice(task); //重新获取修改过的nice值  
   curr\_prio = task\_prio(task); //重新获取修改过的prio值   
   }  
   else if(flag != 0){ //如果flag不等于1或者0返回错误码  
   return EFAULT;  
   }  
    
   copy\_to\_user(nice, &curr\_nice, sizeof(curr\_nice));   
   //将nice值拷贝到用户空间  
   copy\_to\_user(prio, &curr\_prio, sizeof(curr\_prio));   
   //将prio值拷贝到用户空间  
   return 0;  
  }
* 编译内核
* make menuconfig
* make
* make modules
* make modules\_install
* make install
* update-grub2

### 五、实验测试

* 用户态测试程序
* #include <unistd.h>  
  #include <sys/syscall.h>  
  #include <stdio.h>  
    
  #define \_SYS\_TSSETNICE\_ 333  
  #define EFAULT 14  
    
  int main()  
  {  
   int pid, flag, nicevalue;  
   int prev\_prio, prev\_nice, curr\_prio, curr\_nice;  
   int result;  
    
   printf("Hello, this is TS's syscall test...");  
   printf("Please input variable like this: pid, flag(0/1), nicevalue");  
   scanf("%d%d%d", &pid, &flag, &nicevalue);  
    
   result = syscall(\_SYS\_TSSETNICE\_, pid, 0, nicevalue, &prev\_prio, &prev\_nice);  
    
   if(result == EFAULT){  
   printf("ERROR!\n");  
   return 1;  
   }  
   else if(flag == 1){  
   syscall(\_SYS\_TSSETNICE\_, pid, 1, nicevalue, &prev\_prio, &prev\_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", curr\_prio, curr\_nice);  
   }  
   else if(flag == 0){  
   printf("Current priority is %d, Current nice is %d\n", curr\_prio, curr\_nice);  
   }  
   else{  
   printf("flag is not exist\n");  
   }  
   return 0;  
  }
* 查看当前终端的进程pid号，选择flag=0查看当前nice和prio值
* 选择flag=1修改nice和prio值

### 六、问题记录和总结

* 编译完成后版本不变的问题
  + 检查grub.cfg文件，发现新的内核信息已经完全编译好并写入了，但是重启之后还是原来的版本，并没有被我新编译的内核替换
  + 原因：我编译的内核版本比下载的Ubuntu自带的内核版本低，所以还是启动的高版本内核，可以通过修改grub的配置选项， 将开机选择内核版本的菜单栏显示出来（默认是不显示，通过修改配置文件显示）
  + //etc/default/grub  
      
    #GRUB\_HIDDEN\_TIMEOUT //此配置将影响grub菜单显示。若设置此选项为一个常数，则将在此时间内隐藏菜单而显示引导画面。菜单将会被隐藏，如果注释掉该行，则grub菜单能够显示，等待用户的选择，以决定进入哪个系统或内核。
* make -jn(n为参数)
* 看实验书上有些在make后面加上-j可以加快编译速度，结果我加了之后总是编译失败会卡住然后黑屏，一直以为是我系统没装好的锅，重装了好几次虚拟机，后来查资料才发现后面还要加上n，以后遇到问题一定不能空想，白白浪费时间
* 看源码想刨根问底
  + 我会把用到的函数找出来看，然后不出意外里面还是嵌套了很多函数的，简直是无穷无尽，而且很多也想不懂最底层是怎么实现的，非常茫然
  + 但是其实后来查资料看到很多优秀的学习分享，大家也没有像我这样非要找出最底层的代码，其实了解这个函数由哪些功能构成应该就可以了，既然底层都写好了api了，那就直接用好了

#### 附：相关源码

* 通过索引找到pid
* struct pid \*find\_get\_pid(pid\_t nr)  
  {  
   struct pid \*pid;  
    
   rcu\_read\_lock(); //RCU（同步机制）读过程开始  
   pid = get\_pid(find\_vpid(nr));   
   //通过find\_vpid(nr) 来找到进程描述符(应该是遍历表找到)  
   //然后通过get\_pid来让count字段加1  
   rcu\_read\_unlock(); //RCU读过程结束  
    
   return pid;  
  }
* 通过pid找到task
* struct task\_struct \*pid\_task(struct pid \*pid, enum pid\_type type)  
  {  
   struct task\_struct \*result = NULL;  
   if (pid) {  
   struct hlist\_node \*first;  
   first = rcu\_dereference\_check(hlist\_first\_rcu(&pid->tasks[type]),  
   lockdep\_tasklist\_lock\_is\_held());  
   if (first)  
   result = hlist\_entry(first, struct task\_struct, pids[(type)].node);  
   }  
   return result;  
  }
* 返回task的prio和nice
  + static\_prio是静态优先级，不会随时间改变，内核不会主动修改，只能通过系统调用修改nice值去修改它，static\_prio = MAX\_RT\_PRIO + nice +20, MAX\_RT\_PRIO的值为100，nice的范围为-20~19，值越小静态优先级越高；如果进程为非实时进程则prio = static\_prio；若进程为非实时进程，则prio = MAX\_TR\_PRIO -1 - rt\_priority，实施优先级越大进程优先级越高
  + realtime任务的调度优先级永远高于所有的非realtime任务，即非实时任务只有在没有任何实时任务运行时才可能被运行。
  + task\_struct的prio成员取值范围是0~139，值越小，优先级越高，也就是prio为0的task具有最高优先级（idle task），而 task\_prio函数返回的值，对应的范围是-100到39 ，可以看到减去了100
* static inline int task\_nice(const struct task\_struct \*p)  
  {  
   return PRIO\_TO\_NICE((p)->static\_prio);   
   //从进程task\_struct结构中获得静态优先级static\_prio，然后通过PRIO\_TO\_NICE宏将其转化成nice值  
  }  
    
  int task\_prio(const struct task\_struct \*p)  
  {  
   return p->prio - MAX\_RT\_PRIO;  
  }
* 更改nice值
* void set\_user\_nice(struct task\_struct \*p, long nice)  
  {  
   bool queued, running;  
   int old\_prio, delta;  
   struct rq\_flags rf;  
   struct rq \*rq;  
    
   //如果nice值的范围不在-20~19之间，直接退出  
   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:  
   \*/  
   //针对实时进程设置nice值，将nice值转成优先级后设置到p->static\_prio   
   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);  
     
   //将nice值转成优先级设置到static\_prio 中  
   p->static\_prio = NICE\_TO\_PRIO(nice);  
   set\_load\_weight(p);  
   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);  
  }
* 将内核态传递给用户态
* static inline int copy\_to\_user(void \_\_user volatile \*to, const void \*from,  
   unsigned long n)   
  //\*to是内核空间的指针，\*from是用户空间指针，n表示从用户空间想内核空间拷贝数据的字节数  
  {  
   \_\_chk\_user\_ptr(to, n);  
   volatile\_memcpy(to, from, n);  
   return 0;  
  }