操作系统实验报告

实验一:lINUX内核编译及添加系统调用

x’x’x’x’xx’x’x

2019

目录

[一、实验目的 2](#_Toc8272436)

[二、实验要求 2](#_Toc8272437)

[三、开发平台 2](#_Toc8272438)

[四、实验设计方案 3](#_Toc8272439)

[五、测试程序和测试截图 12](#_Toc8272440)

[六、实验遇到的问题及解决方式 16](#_Toc8272441)

[七、实验结果讨论 17](#_Toc8272442)

LINUX内核编译及添加系统调用

## 一、实验目的

通过对内核编译和系统调用的添加，理解Linux系统处理系统调用的流程以及增加系统调用的常用方法。

## 二、实验要求

（1）添加一个系统调用，实现对指定nice值的修改和读取功能，并返回进程最新的nice值和优先级prio。建议的调用原型为

|  |
| --- |
| **int** mysetnice(pid\_t pid, **int** flag, **int** nicevalue, **void** \_user\* prio, **void** \_user\* nice); |

参数含义：

Pid:进程ID

Flag： 值为0，表示读取nice；值为1表示修改nice值。

Nicevalue：为指定进程设置新的nice值。

Prio，nice：指向进程当前优先级和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、开发平台

Ubuntu18.04 gcc gdb vscode

## 四、实验设计方案

1. 系统调用的添加分为三个步骤分别是：
   1. 修改系统调用表添加所需要的系统调用名和入口点
   2. 声明系统调用响应的函数原型
   3. 在系统调用的函数文件中添加响应系统调用的函数，实现相应的的功能
2. 对于第一步的操作我选择在arch/x86/entry/syscalls/syscall\_64.tbl文件中添加如下内容

|  |
| --- |
| 335 64 whphellosyscall \_\_x64\_sys\_whpsetnice |

即系统调用号是335 类型是64 调用名是 whphellosyscall 入口点是 \_\_x64\_sys\_whpsetnice

1. 第二步添加系统调用的服务例程原型

我的操作是在include/linux/syscalls.h 文件中添加如下内容

|  |
| --- |
| *//添加系统调用 whphello*  asmlinkage **long** sys\_whpsetnice(pid\_t pid, **int** flag, **int** nicevaluse, **void** \_\_user \*prio, **void** \_\_user \*nice); |

可以看到我用的服务例程和书上给出的几乎一致，但是由于实际环境的不同我的 \_\_user和书上的\_user不同

1. 添加系统调用服务例程

调用例程设计

* 1. 程序框图

|  |
| --- |
|  |

* 1. 操作步骤

在kernel/sys.c中添加

|  |
| --- |
| SYSCALL\_DEFINE5(whpsetnice, pid\_t, pid, **int**, flag, **int**, nicevalue, **void** \_\_user \*, prio, **void** \_\_user \*, nice)  {  **int** now\_prio, now\_nice;  **struct** pid \*struct\_pid;  **struct** task\_struct \*PCB;  *// 进程PID号找到PID结构体，为空则返回错误值*  struct\_pid = find\_get\_pid(pid);  if (struct\_pid == NULL)  return EFAULT;    *// 找到与之PID结构体对应的进程控制块*  PCB = pid\_task(struct\_pid, PIDTYPE\_PID);  *// flag=1修改进程nice 为 nicevalue*  if (flag == 1)  {          if(nicevalue>19||nicevalue<-20){              return EFAULT;          }else{              set\_user\_nice(PCB, nicevalue);          }  } *//flag出错，返回EFAULT*  else if (flag != 0)  {  return EFAULT;  }  *//获取进程现在的nice值和prio值*  now\_prio = task\_prio(PCB);  now\_nice = task\_nice(PCB);  *// copy\_to\_user()将内核空间的数据复制到用户空间*  copy\_to\_user(prio, &now\_prio, sizeof(now\_prio));  copy\_to\_user(nice, &now\_nice, sizeof(now\_nice));      return 0;  } |

相关函数源码分析

* 1. find\_get\_pid

|  |
| --- |
| *//该函数接受一个pid\_t类型的参数并且返回相应的pid结构体*  **struct** pid \*find\_get\_pid(pid\_t nr)  {  **struct** pid \*pid;      rcu\_read\_lock();      pid = get\_pid(find\_vpid(nr));      rcu\_read\_unlock();      return pid;  }  EXPORT\_SYMBOL\_GPL(find\_get\_pid);  *//返回的结构体如下*  **struct** pid  {      atomic\_t count;  **unsigned** **int** level;  */\* lists of tasks that use this pid \*/*  **struct** hlist\_head tasks[PIDTYPE\_MAX];  **struct** rcu\_head rcu;  **struct** upid numbers[1];  };  *//和该函数相关的其他函数源码*  *//通过nr找到 名称空间*  **struct** pid \*find\_vpid(**int** nr)  {      return find\_pid\_ns(nr, task\_active\_pid\_ns(current));  }  *//返回pid结构体*  **static** **inline** **struct** pid \*get\_pid(**struct** pid \*pid)  {      if (pid)          atomic\_inc(&pid->count);      return pid;  }  *//在名称空间中找nr对应的结构体*  **struct** pid \*find\_pid\_ns(**int** nr, **struct** pid\_namespace \*ns)  {      return idr\_find(&ns->idr, nr);  }  *//返回红黑树中的对应idr*  **void** \*idr\_find(**const** **struct** idr \*idr, **unsigned** **long** id)  {      return radix\_tree\_lookup(&idr->idr\_rt, id - idr->idr\_base);  }  EXPORT\_SYMBOL\_GPL(idr\_find);  *//在红黑树中查找的返回*  **void** \*radix\_tree\_lookup(**const** **struct** radix\_tree\_root \*root, **unsigned** **long** index)  {      return \_\_radix\_tree\_lookup(root, index, NULL, NULL);  }  EXPORT\_SYMBOL(radix\_tree\_lookup);  *//红黑树查找算法*  **void** \*\_\_radix\_tree\_lookup(**const** **struct** radix\_tree\_root \*root,  **unsigned** **long** index, **struct** radix\_tree\_node \*\*nodep,  **void** \_\_rcu \*\*\*slotp); |

* 1. pid\_task

|  |
| --- |
| *//返回进程控制块*  **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, pid\_links[(type)]);      }      return result;  }  EXPORT\_SYMBOL(pid\_task);  *//相关代码哈希表的结构体*  **struct** hlist\_node {  **struct** hlist\_node \*next, \*\*pprev;  };  *//rcu的检查*  #define rcu\_dereference\_check(p, c) \      \_\_rcu\_dereference\_check((p), (c) || rcu\_read\_lock\_held(), \_\_rcu)  *//获取哈希表的节点的入口点*  #define hlist\_entry(ptr, type, member) container\_of(ptr,type,member)  *//实现上方宏定义功能的另一个宏定义*  #define container\_of(ptr, type, member) ({              \  **void** \*\_\_mptr = (**void** \*)(ptr);                   \      BUILD\_BUG\_ON\_MSG(!\_\_same\_type(\*(ptr), ((type \*)0)->member) &&   \               !\_\_same\_type(\*(ptr), **void**),            \               "pointer type mismatch in container\_of()");    \      ((type \*)(\_\_mptr - offsetof(type, member))); }) |

* 1. set\_user\_nice

|  |
| --- |
| *//设置用户的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;  *//判断nice值是否符合规定*      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.*  *\*/*  *//为就绪队列加锁*      rq = task\_rq\_lock(p, &rf);  *//发出时钟信号*  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值其实是没有作用的，仍将值设置好p->static\_prio*      if (task\_has\_dl\_policy(p) || task\_has\_rt\_policy(p)) {          p->static\_prio = NICE\_TO\_PRIO(nice);          goto out\_unlock;  }  *//对于就绪和运行队列中的采取不同处理方式*      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;  *//将其返回原来的队列*      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);  }  *//有关nice和prio的宏定义*  #define NICE\_TO\_PRIO(nice)  ((nice) + DEFAULT\_PRIO)  #define DEFAULT\_PRIO        (MAX\_RT\_PRIO + NICE\_WIDTH / 2) *//120*  #define MAX\_USER\_RT\_PRIO    100  #define MAX\_NICE    19  #define MIN\_NICE    -20  #define NICE\_WIDTH  (MAX\_NICE - MIN\_NICE + 1) |

* 1. task\_prio

|  |
| --- |
| *//返回进程优先级*  **int** task\_prio(**const** **struct** task\_struct \*p)  {      return p->prio - MAX\_RT\_PRIO;  } |

* 1. task\_nice

|  |
| --- |
| *//返回进程nice*  **static** **inline** **int** task\_nice(**const** **struct** task\_struct \*p)  {      return PRIO\_TO\_NICE((p)->static\_prio);  } |

* 1. copy\_to\_user

|  |
| --- |
| *//copy\_to\_user()将内核空间的数据复制到用户空间*  copy\_to\_user(**void** \_\_user \*to, **const** **void** \*from, **unsigned** **long** n)  {      if (likely(check\_copy\_size(from, n, true)))          n = \_copy\_to\_user(to, from, n);      return n;  }  *//调用原型判断是否访问成功然后使用原始代码来复制数据*  \_copy\_to\_user(**void** \_\_user \*to, **const** **void** \*from, **unsigned** **long** n);  *//调用原始文件，包含汇编指令*  raw\_copy\_to\_user(**void** \_\_user \*dst, **const** **void** \*src, **unsigned** **long** size); |

内核编译:

在命令行切换到内核源码的根目录下进行操作

|  |
| --- |
| #菜单配置  Make menufonfig  #编译，并输出错误  make 2>err.log  #编译模块  make modules  #安装模块  make modules\_install  #安装内核  make install  #更新 grub2配置  update-grub2 |

第一次内核编译出现了不少缺少包，而无法进行下去的时候应该使用包管理命令安装相应的依赖软件，安装完成后需要清理掉不完整的编译文件，通过

|  |
| --- |
| Make mrproper |

清理文件，并重新重复上述步骤。但是成功编译一次后便可以不清理，编译成功的文件了。

Sudo apt install 包名 根据提示安装对应的包即可。

## 五、测试程序和测试截图

程序框图

|  |
| --- |
|  |

程序代码

|  |
| --- |
| *//测试程序*  #include <unistd.h>  #include <sys/syscall.h>  #include <stdio.h>  #define SYSCALL\_NUM 335  #define EFAULT 14  **int** main()  {  **int** PID, FLAG, NICE;  **int** pre\_prio, pre\_nice, now\_prio, now\_nice;  **int** res;  printf("请输入PID值:");  scanf("%d", &PID);  printf("\n修改nice值？（y=1,n=0）");  scanf("%d",&FLAG);  if (FLAG==1)  {  printf("\n请输入nice值：");  scanf("%d",&NICE);  }  res = syscall(SYSCALL\_NUM, PID, 0, NICE, &pre\_prio, &pre\_nice);    if (res == EFAULT)  {  printf("ERROR!");  return 1;  }    if (FLAG == 1)  {  res = syscall(SYSCALL\_NUM, PID, 1, NICE, &now\_prio, &now\_nice);  if(res == EFAULT){  printf("ERROR!");  }else  {  printf("原来的PRIORITY是：[%d], 原来的NICE值是：[%d]\n", pre\_prio, pre\_nice);  printf("现在的PRIORITY是：[%d], 现在的NICE值是：[%d]\n", now\_prio, now\_nice);    }    }  else if (FLAG == 0)  {  printf("现在的PRIORITY是：[%d], 现在的NICE值是：[%d]\n", pre\_prio, pre\_nice);  }else{  printf("你的操作不正确请重新输入flag值");  }    return 0;  } |

测试截图

|  |
| --- |
|  |

圆满完成任务

## 六、实验遇到的问题及解决方式

内核编译出错问题

1. arch/x86/entry/syscall\_64.o:(.rodata 0xa78):对‘sys\_whphellosyscall’未定义

该问题出现的原因是服务例程的入口声明方式不正确，解决方式是参照该内核中其他系统调用的样式进行添加和实现

1. 内核代码出现语法错误导致内核编译很快结束但是并没有编译成功且无法知道错误在啥地方

编译内核可以使用一个 make 2>err.log来捕获我们可能出现的错误

1. 当系统调用设置不能设置的nice值的时候并没有返回错误代码

通过对系统调用的服务例程的修改实现了对nice值的监控，当nice值不符合源代码中规定的nice值时直接返回错误代码

1. 对linux的内核函数不了解使用起来比较乏力

通过网上查找相关的系统调用的例子知道了相关函数的存在后，通过源码查看工具获取源码信息了解其工作原理

1. 书本上的查看源代码的工具和方式，比较繁琐且使用困难

使用linux源码查看网站快速查看

[elixir.bootlin.com](https://elixir.bootlin.com/linux/latest/source)

1. 每次编译代码前使用make mrproper 或 make clean还需要做很长时间的内核编译。

由于make就是在解决每次不必全部编译的问题所以当只是修改Linux源码的内容时不需要将之前的编译生成的文件全删除，因为make会自动编译那些改动过的源码，依赖其生成的也会重新编译，这样可以省去大量的内核编译时间。不必每次都重新编译所有代码。不过这还是要取决于makefile的完善程度，如果makefile不好那么还是可能会出现比较可怕的依赖问题。

1. 编译源码生成的是啥文件，内核放在啥地方，grub又是啥玩意？

编译文件会在对应的文件目录下生成 \*.o文件 根目录下生成vmlinux.o

## 七、实验结果讨论

实验结果符合实验预期，可以说实验是成功的，通过该实验我了解了Linux添加系统调用的流程和增加系统调用的方式，达到了该实验的目的。同时完成实验的过程中，我阅读了一部分Linux源码，做到了操作系统级别的程序代码的书写，通过这次实验我对Linux下的操作系统实验有了初步的了解，这为我要做的其他实验打下了重要的基础。

本次实验的结果证明了Linux作为一款开源的操作系统的的高度自由化，使用者可以自由的修改和定制属于自己的Linux内核，这为一些优秀的产品的出现做出了很大的贡献。比如Android操作系统就是在Linux的基础上开发出来的。相信通过本次学习我们会对Linux了解更加深入。同时这也为我们打开了Linux系统开发的新大门。