## Linux0.11系统调用的添加与进程调度算法的修改

学号：201506021041 姓名：杨涛 专业：网络工程

# 实验目的

## 建立对系统调用接口的深入认识，掌握系统调用的基本过程，能完成系统调用的全面控制

## 掌握进程调度的过程，了解进程调度算法并且能够对其修改

# 实验内容

此次试验的基本内容是：

1. 在Linux 0.11上添加第二种进程调度算法
2. 添加一个系统调用实现在两种调度算法之间进行切换
3. 编写适当的应用来验证算法的正确性，重点是体现出两种调度算法的区别

# 实验过程

一、用文字简要描述向Linux 0.11添加一个系统调用call()的步骤

1. 打开bochsrc-hd

先添加用户的系统函数接口call();

cd ../src/linux 有很多文件，cd include 进入include，vi unistd.h 按i将只读变成可编辑模式，加上#define \_\_NR\_call 87 最后面加函数原型int call。如图一所示，按Esc退出编辑模式，shift+ ：键，然后wq 退出保存



图一

2.添加用户态函数对应的系统核心态函数

cd linux 进入include下的linux目录，vi sys.h编辑，添加

sys\_call

同样保存退出

3.修改系统调用的个数

cd ../../ 退到上上级目录，cd kernel 进入内核，

vi system\_call.s,

修改nr\_system\_calls = 87，

原来的是86 个系统调用，添加了call系统调用，现在改成87或者更大，因为判断系统调用是否合法是用系统调用的进程号与nr\_system\_calls 比较，小于等于就是合法的。

保存退出

4.修改调度算法

vi sched.c,在schedule函数下完成算法FIFO，flag是一个在外面的全局变量，告诉用来选择哪个调度算法。

修改schedule函数

修改部分如下

if(call==1)

{

i=NR\_TASKS;

next=NR\_TASKS-1;

while(i--){

if(!task[tasknext[i]])

continue;

if(task[tasknext[i]]->state==TASK\_RUNNING)

{next=tasknext[i];c=i;}

}

for(i=c;i<NR\_TASKS-1;i++)

tasknext[i]=tasknext[i+1];

tasknext[NR\_TASKS-1]=next;

}

else

{

/\* this is the scheduler proper: \*/

while (1) {

c = -1;

next = 0;

i = NR\_TASKS;

p = &task[NR\_TASKS];

while (--i) {

if (!\*--p)

continue;

if ((\*p)->state == TASK\_RUNNING && (\*p)->counter > c)

c = (\*p)->counter, next = i;

}

if (c) break;

for(p = &LAST\_TASK ; p > &FIRST\_TASK ; --p)

if (\*p)

(\*p)->counter = ((\*p)->counter >> 1) +

(\*p)->priority;

}

}

注释：If else语句选择调度算法，if中的是先进先出，else是原时间片分配的算法，其他照旧，注意语法错误

tasknext是64位的整形数组，存进程的下标，是全局变量，在sys\_call（）中被初始化。

并添加sys\_call(int flag)核心态函数的具体实现。如下

int sys\_call(int flag)

{ int i;

if(flag!=1)//在不使用FIFO算法的系统调用时，进程下标数组的初始化

{

for(i=0;i<NR\_TASKS;i++)

tasknext[i]=i;

}

call=flag;

schedule();

return 0;

}

注释：sys\_call是call系统调用在系统核心态的具体实现，可以放在sched.c中

保存退出

5.在src/linux 目录下输入

make clean

make

成功后，dd bs=8192 if=Image of=/dev/fd0,将生成的操作系统文件写入硬盘

直接reset，回到根目录

6.cd ../ 进入上级即用户目录，cd include ，存放用户目录的头文件的地方，我们在操作系统中修改的unistd.h,这里的也要修改，直接copy

命令如下cp ../src/linux/include/unistd.h unistd.h

如果不进行这一步，用户程序编译的时候，显示call（）系统调用不存在，因为c代码便以的预编译命令#include<unistd.h>是在/usr/include目录下找用户态系统调用函数。

至此，call()系统调用添加成功。

二、用文字简要描述你所实现的系统调用的算法，并列出实现算法的关键代码

关键代码：

i=NR\_TASKS;

next=NR\_TASKS-1;

while(i--){

if(!task[tasknext[i]])

continue;

if(task[tasknext[i]]->state==TASK\_RUNNING)

{next=tasknext[i];c=i;}

}

for(i=c;i<NR\_TASKS-1;i++)

tasknext[i]=tasknext[i+1];

tasknext[NR\_TASKS-1]=next;

算法描述：初始状态下，tasknext数组存放0到NR\_TASKS-1的整型数，与进程控制块task一一对应，

每次按tasknext数组从后往前访问，找到最前一个处于运行状态的进程，记录该进程的下标即task下标于next，之后的tasknext数组都向前挪移一位，由于是以tasknext数组访问进程控制块，就等效于进程控制列task向前移动一位。tasknext的最后一位指向刚找到的运行态进程的下标。schedule的最后将会低级调度到该进程执行。

1. 用户态程序的实现

#define \_\_LIBRARY\_\_

#include<stdio.h>

#include<unistd.h>

\_syscall1(int,call,int,flag)//由于系统调用的原型是int call(int flag) 说明该系统调用需

//要传入一个参数,因此syscall后面是1,

//添加的系统调用都要在用户态程序中声明

int main()

{ int i;

call(0);//非FIFO算法，完成tasknext的初始化

if(!fork())

{

for(i=0;i<5;i++){

call(1);

printf("1\n");

}

exit(0);

}

if(!fork())

{

for(i=0;i<5;i++){

call(1);

printf("2\n");

}

exit(0);

}

if(!fork())

{

for(i=0;i<5;i++){

call(1);

printf("3\n");

}

exit(0);

}

for(i=0;i<3;i++)

waitpid(-1,NULL,0);

call(0);//回到原来的系统调用算法

return 0;

}

实验心得