仓库链接：https://github.com/zhaoyihui233/mit-6.S081-lab-2021/tree/main/lab3%20Page%20tables

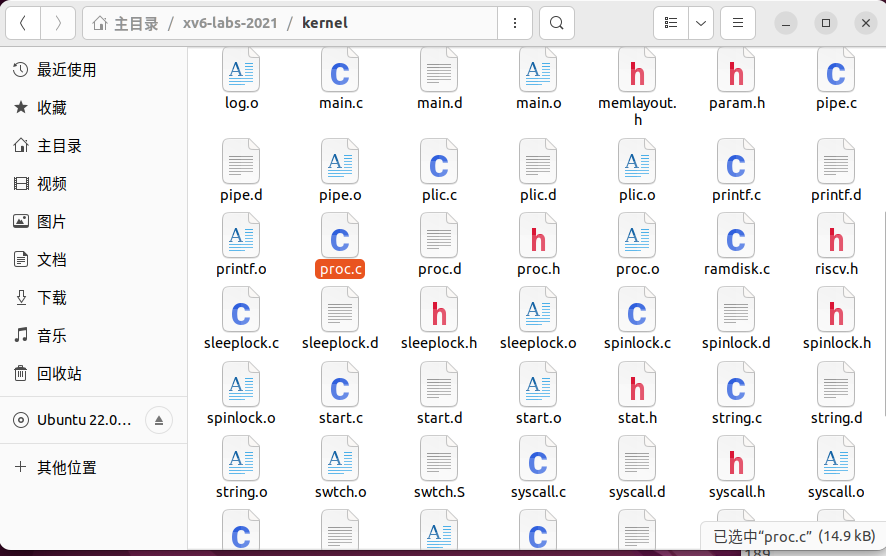
# 1 Speed up system calls

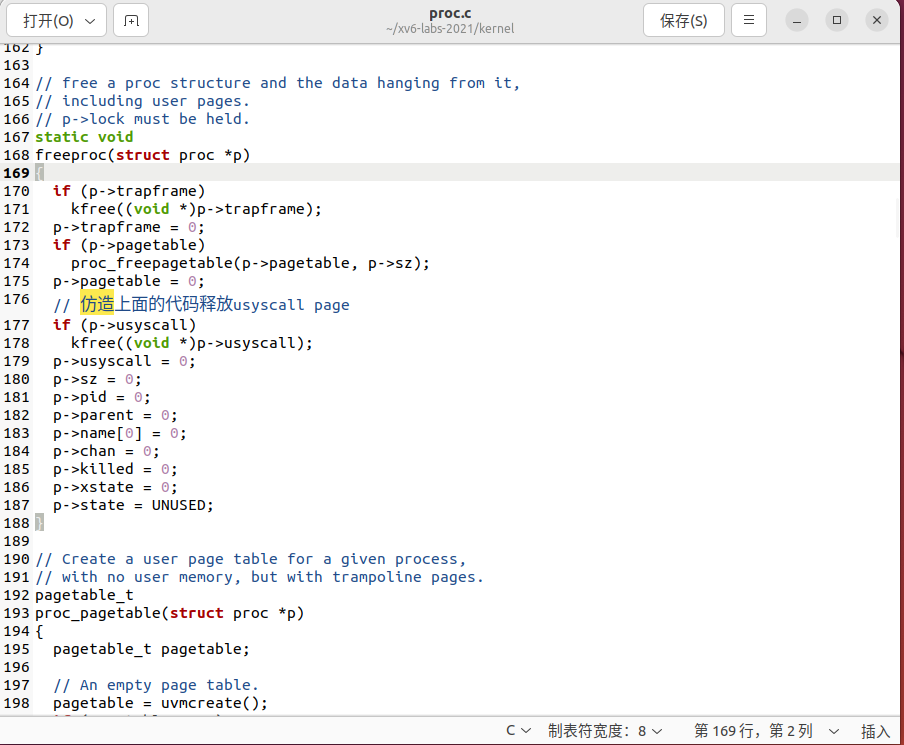
## 1.1实验内容

在本次实验中，我们要编写代码，通过加一层缓存的形式来加速系统调用。在用户态与内核态之间开辟了一个可读的内存区域，用户态程序在调用系统调用时可以直接访问其中，而不需要通过内核态做中介。 为实现这些，实验要求我们加速getpid这个系统调用。

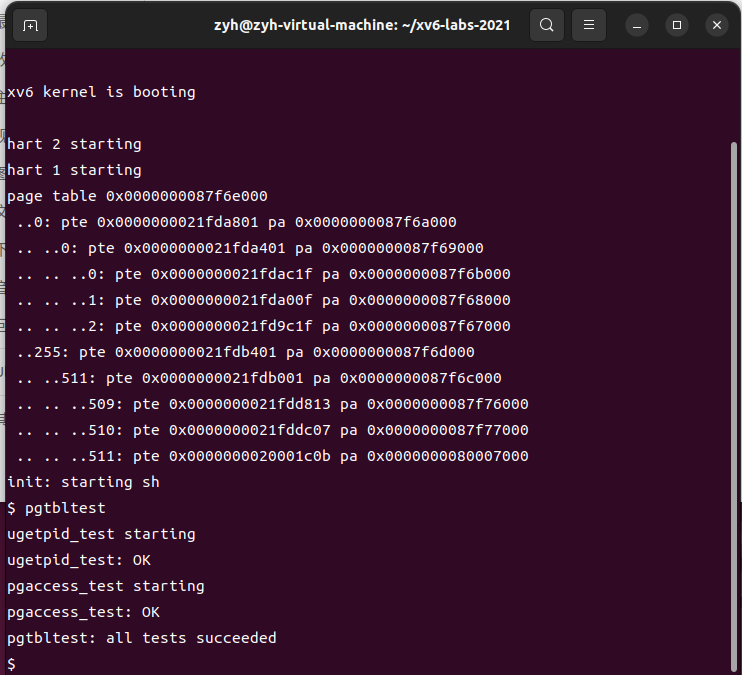
根据提示，需要的工作有：一，修改kernel/proc.c文件里的proc\_pagetable函数，为上述内存空间实现映射。二，修改kernel/proc.c文件里的allocproc函数，为新建进程分配usyscall页表。三，修改kernel/proc.c文件里的allocproc函数，进程释放时新增的页表相应的也要释放。四，修改kernel/proc.c文件里的proc\_freepagetable函数，正确释放新增的页表空间。

## 1.2代码位置与截图

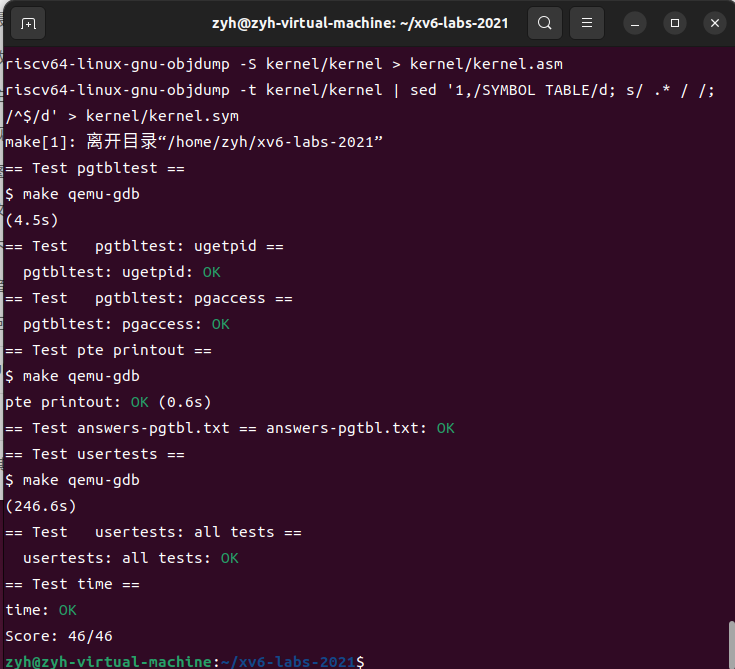




## 1.3实验结果



make grade:



## 1.4代码实现

### 1.4.1实现思路

根据提示，先在proc\_pagetable函数里添加为usyscall page设置映射关系的代码：

if (mappages(pagetable, USYSCALL, PGSIZE, (uint64)(p->usyscall), PTE\_R | PTE\_U) < 0)

{

uvmunmap(pagetable, TRAPFRAME, 1, 0);

uvmunmap(pagetable, TRAMPOLINE, 1, 0);

uvmfree(pagetable, 0);

return 0;

}

这里权限是PTE\_R和PTE\_U，说明该页表只读且用户态程序可读。

然后，需要在同文件里的allocproc函数里添加为usyscall page分配空间的代码：

if ((p->usyscall = (struct usyscall \*)kalloc()) == 0)

{

freeproc(p);

release(&p->lock);

return 0;

}

p->usyscall->pid = p->pid;

最后，在释放进程时相关的页表也要释放。于是也在同文件里的freeproc函数里添加一些代码：

if (p->usyscall)

kfree((void \*)p->usyscall);

p->usyscall = 0;

并且，还要到proc\_freepagetable函数里添加一条代码：uvmunmap(pagetable, USYSCALL, 1, 0);

# 2 Print a page table

## 2.1实验内容

本次实验中，需要我们编写一个函数vmprint来打印页表内容。该函数只有一个参数，为pagetable\_t结构体类型。函数需要定义在kernel/vm.c文件中，

为能执行我们写的函数，根据提示需要在kernel/exec.c文件里声明vmprint函数，并在exec函数里添加一行代码：if(p->pid==1) vmprint(p->pagetable)。进程PID为1的进程，就是sh进程。不过，我在编写函数时，为了方便实现，我还添加的了一个整型参数，用于表示递归的深度。所以添加的代码也就成了if(p->pid==1) vmprint(p->pagetable, 0)。

那么如何验证我们编写的代码？启动编译xv6后，如果输出了形如以下的内容就是函数编写正确：  
page table 0x0000000087f6e000

..0: pte 0x0000000021fda801 pa 0x0000000087f6a000

.. ..0: pte 0x0000000021fda401 pa 0x0000000087f69000

.. .. ..0: pte 0x0000000021fdac1f pa 0x0000000087f6b000

.. .. ..1: pte 0x0000000021fda00f pa 0x0000000087f68000

.. .. ..2: pte 0x0000000021fd9c1f pa 0x0000000087f67000

..255: pte 0x0000000021fdb401 pa 0x0000000087f6d000

.. ..511: pte 0x0000000021fdb001 pa 0x0000000087f6c000

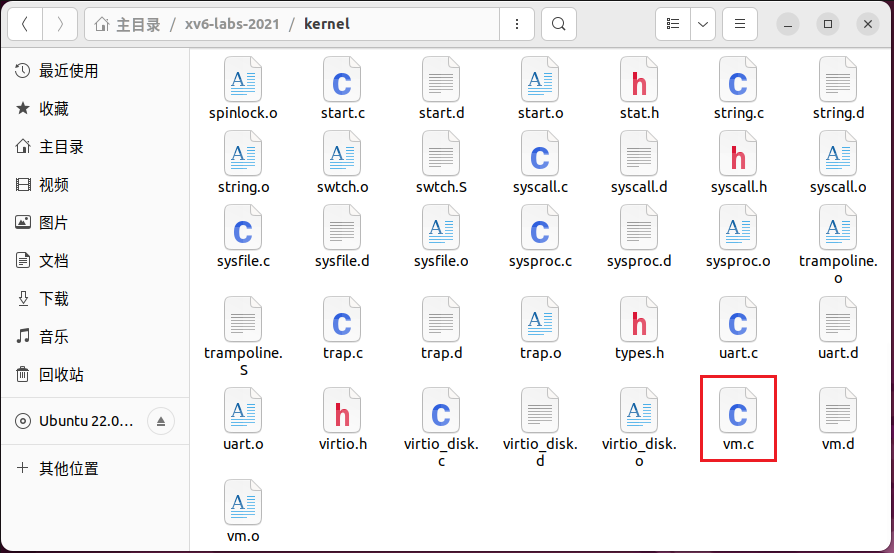
.. .. ..509: pte 0x0000000021fdd813 pa 0x0000000087f76000

.. .. ..510: pte 0x0000000021fddc07 pa 0x0000000087f77000

.. .. ..511: pte 0x0000000020001c0b pa 0x0000000080007000

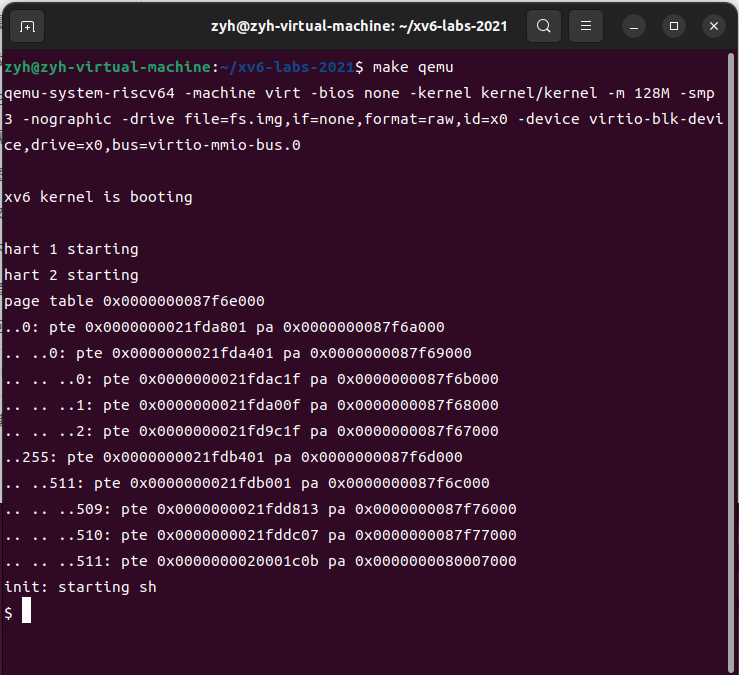
可以看到，对页表树的输出应该采取的是深度优先的形式，这也是我们在编写代码时采用的算法。由xv6文档可知，一个PTE包含512项，但通过观察给出的输出，我们只需要输出有信息意义的PTE。这一点可以通过PTE的一个标志位PTE\_V来判断。

## 2.2代码位置与截图

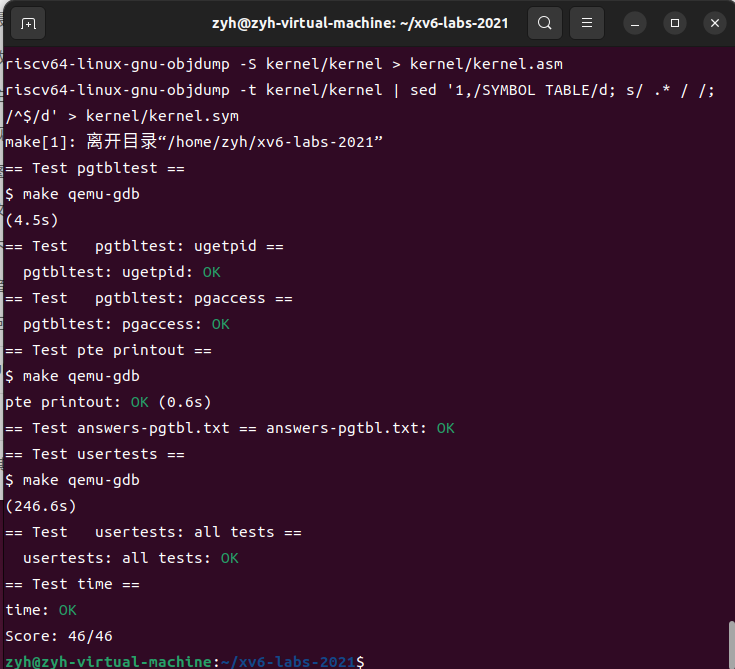




## 2.3实验结果



make grade：



## 2.4代码分析&结果分析

### 2.4.1代码实现

根据提示，我们可以参考freewalk函数来编写我们的代码，该函数也在vm.c文件里。实际上，只需要稍加修改就能为我们所用。

// 为方便输出,定义一个字符串数组

static char \*prefix[] = {

[0] = "..",

[1] = ".. ..",

[2] = ".. .. .."};

// 我们要编写的函数vmprint

void vmprint(pagetable\_t pagetable, uint64 depth)

{

if (depth == 0)

{

printf("page table %p\n", pagetable);

}

if (depth > 2)

{

return;

}

char \*buf = prefix[depth];

for (int i = 0; i < 512; i++)

{

pte\_t pte = pagetable[i];

if (pte & PTE\_V)

{

printf("%s%d: pte %p pa %p\n", buf, i, pte, PTE2PA(pte));

uint64 child = PTE2PA(pte);

vmprint((pagetable\_t)child, depth + 1);

}

}

}

通过观察freewalk函数，可以可以通过%p输出逻辑地址与物理地址。通过PTE2PA函数，可以将逻辑地址转化成物理地址。

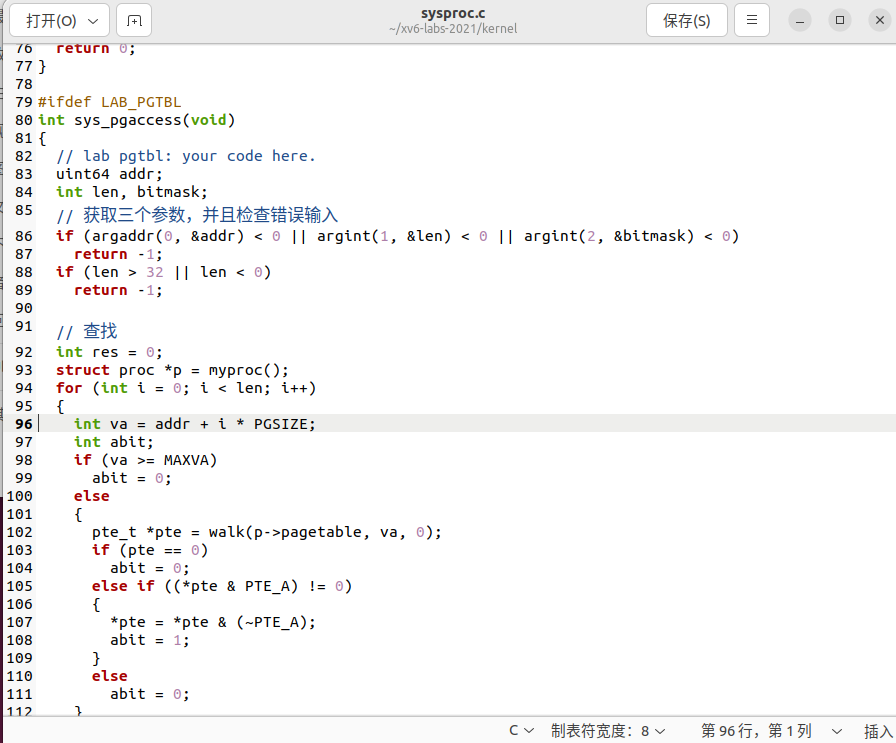
# 3 Detecting which pages have been accessed

## 3.1实验内容

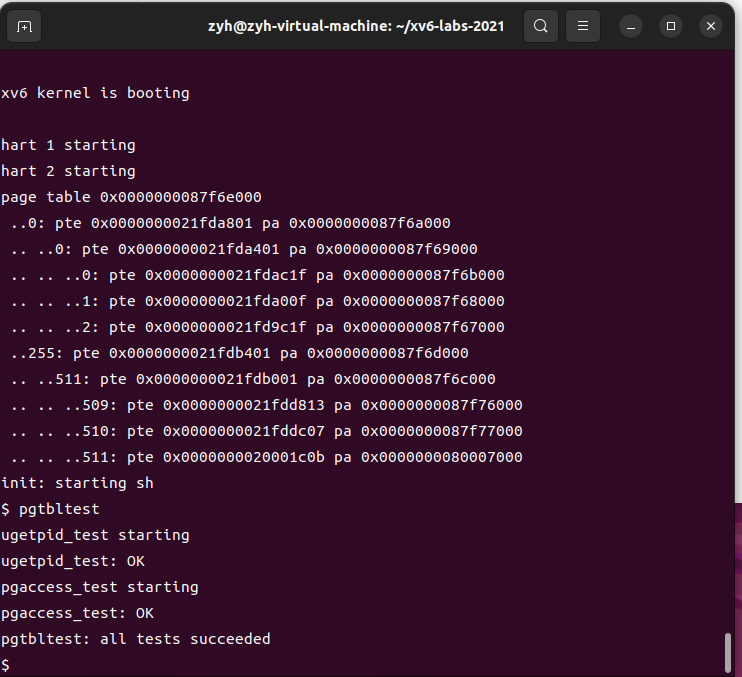
本次实验中，需要我们编写一个系统调用pgaccess，来实现获取一个页表中的页表项有哪些被acssess（读或写）过，这通过标志位PTE\_A可以看出来。PTE\_A为1，表示页表被读写过。另外，我们需要把查找到的PTE\_A为1的页表项都置为0。该系统调用接收三个参数，第一个参数是页表的第一个页表项的逻辑地址，第二个参数是要检查的页表项的个数，第三个参数是引用参数，我们的检查结果将通过该参数反馈出来。具体来说，第三个参数的每一个二进制位都代表一个页表项的PTE\_A，且从该参数的低位开始记录PTE\_A的结果。

作为一个系统调用，它的实现在kernel/sysproc.c文件里的sys\_pgaccess函数。

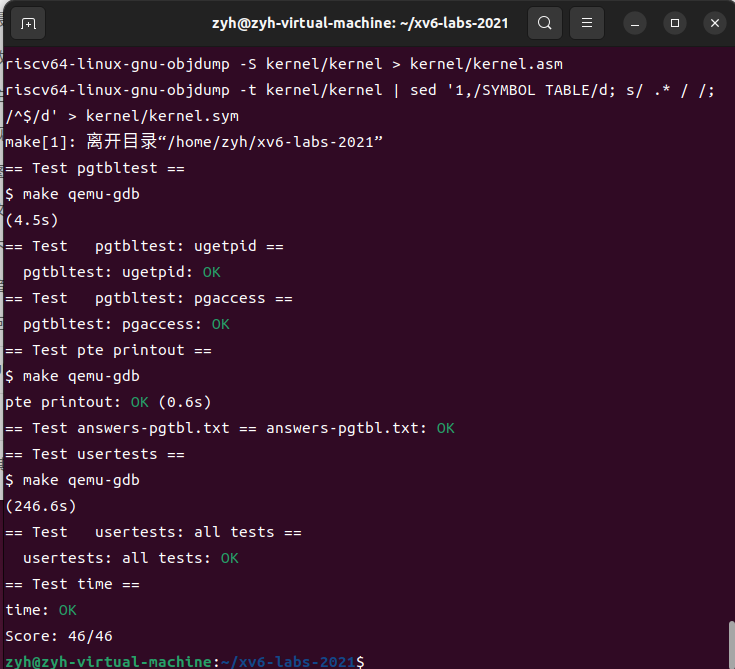
## 3.2代码位置与截图



## 3.3实验结果



make grade:



## 3.4代码分析&结果分析

### 3.4.1代码实现

根据提示，系统调用接收参数的形式与用户函数不一样，需要借助argaddr函数和argint函数，这一点可以参考其他系统调用的实现。

PTE\_A的标志位需要我们到kernel/riscv.h文件里定义出来。通过查阅手册，可写出定义#define PTE\_A (1L << 6)，其中的6就是查阅得知。

另外，在我们此次实验中，walk函数起到重要作用。因为在pgacssess的参数实际上是页表的虚拟地址，所以我们需要将之转化成物理地址，而walk函数就能实现这个功能。如果虚拟地址所指页表项不合法，那么walk返回0；如果合法，那么返回物理地址。

我们拿到一个页表项的逻辑地址后，就能判断其PTE\_A标志位是否为1。

那么，下面是系统调用的实现：

int sys\_pgaccess(void)

{

// lab pgtbl: your code here.

uint64 addr;

int len, bitmask;

// 获取三个参数，并且检查错误输入

if (argaddr(0, &addr) < 0 || argint(1, &len) < 0 || argint(2, &bitmask) < 0)

return -1;

if (len > 32 || len < 0)

return -1;

// 查找

int res = 0;

struct proc \*p = myproc();

for (int i = 0; i < len; i++)

{

int va = addr + i \* PGSIZE;

int abit;

if (va >= MAXVA)

abit = 0;

else

{

pte\_t \*pte = walk(p->pagetable, va, 0);

if (pte == 0)

abit = 0;

else if ((\*pte & PTE\_A) != 0)

{

\*pte = \*pte & (~PTE\_A); //如果PTE\_A为1,按照实验要求则需要复位为0

abit = 1;

}

else

abit = 0;

}

res = res | abit << i; // 将结果记录在res的低位起第i位

}

// 将结果赋予bitmask

if (copyout(p->pagetable, bitmask, (char \*)&res, sizeof(res)) < 0)

return -1;

return 0;

}