# 1 System call tracing

## 1.1实验内容

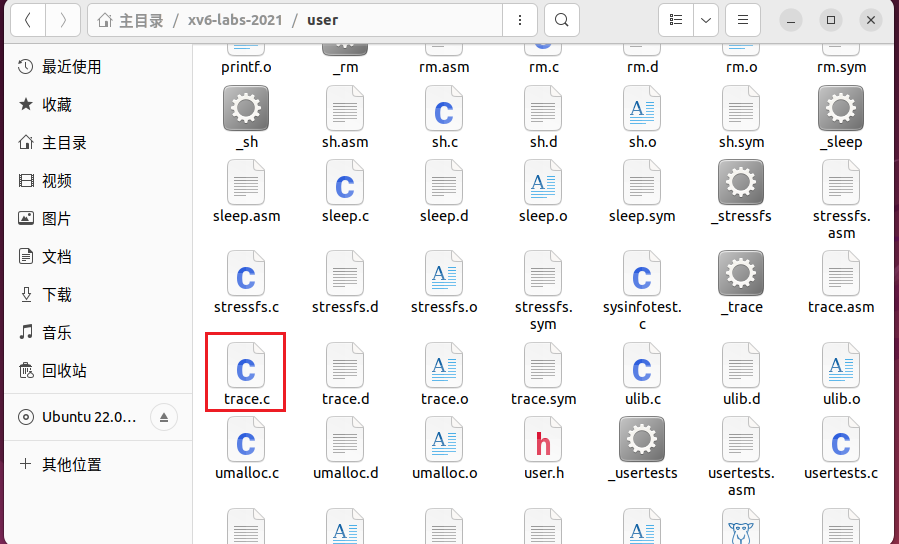
在本次实验中，我们要自己编写一个系统调用trace来“跟踪”其他系统调用，这个系统调用在调试后续实验代码时会很有帮助。trace会输出调用指定系统调用的进程ID、系统调用名称及其返回值。trace仅做跟踪作用，不应该影响到原进程。trace的参数只有一个整数参数，被称为mask，表示1左移位的位数。不同位数的1，代表跟踪不同的系统调用。不同系统调用对应的mask可在kernel/syscall.h文件里查看。

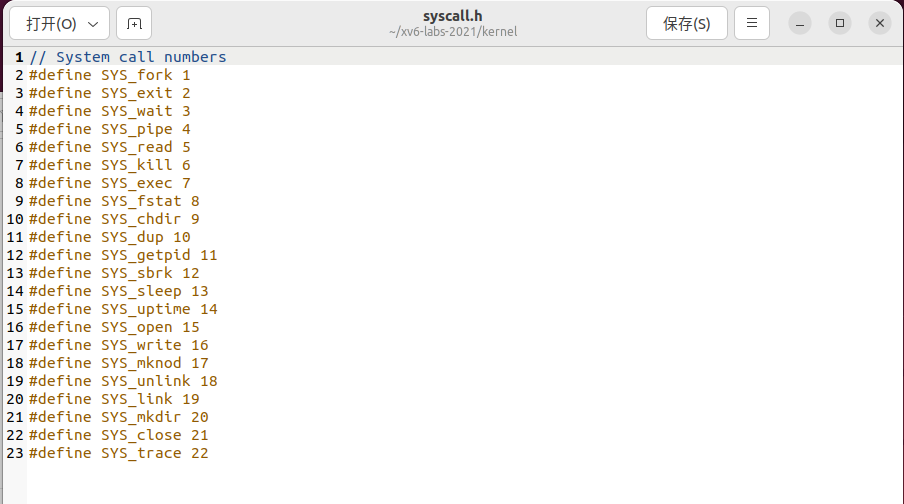
实验已实现user/trace.c，但需要切换git分支才能找到。这里我们在xv6-labs-2021文件夹下打开终端，输入git fetch，可以看到所有分支和当前分支，当前分支在util，我们要切换到实验所用的分支，输入git checkout syscall即可，这时就能看到user/trace.c文件了。该文件里就调用了待实现的系统调用trace。

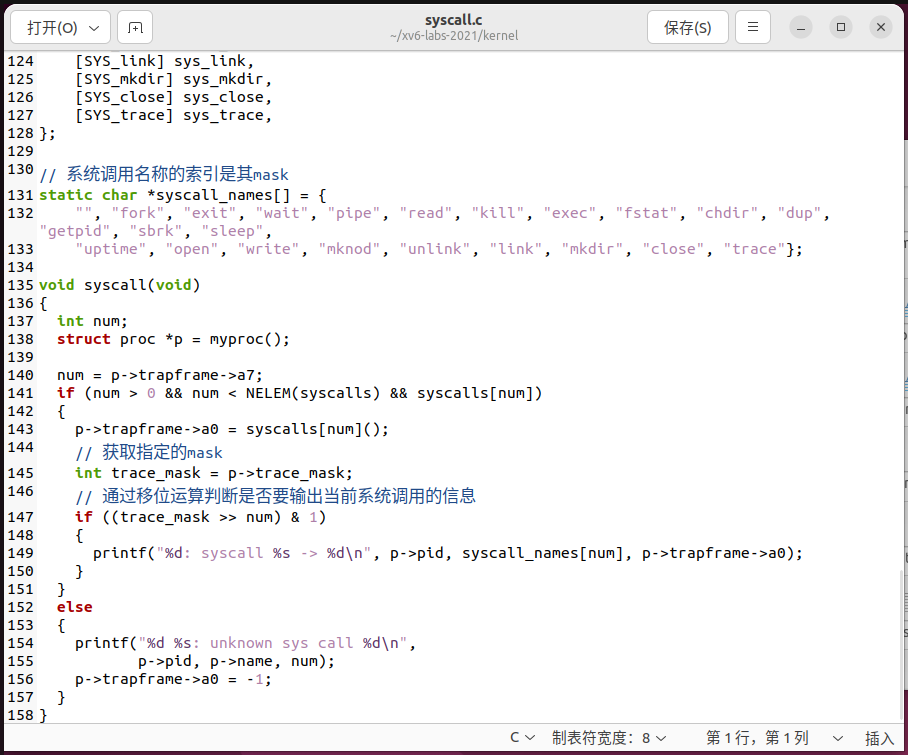
在编写trace系统调用的代码的同时，还要做好配置。一，尽管实验已经给我们准备了trace.c文件，但还是要我们自己在Makefile文件中添加$U/\_trace\。二，在user/user.h文件里要声明int trace(int)系统调用。三，在user/usys.pl文件里要添加entry(“trace”)。在kernel/syscall.h文件里添加#define SYS\_trace 22，这里的数字就是trace这个系统调用的mask，跟着上面其他系统调用的mask定义规律来即可。四，在kernel/syscall.c文件中新增sys\_trace函数定义，并在函数指针数组中新增sys\_trace。

接下来，就要在kernel/sysproc.c文件里实现该系统调用。

## 1.2代码位置与截图

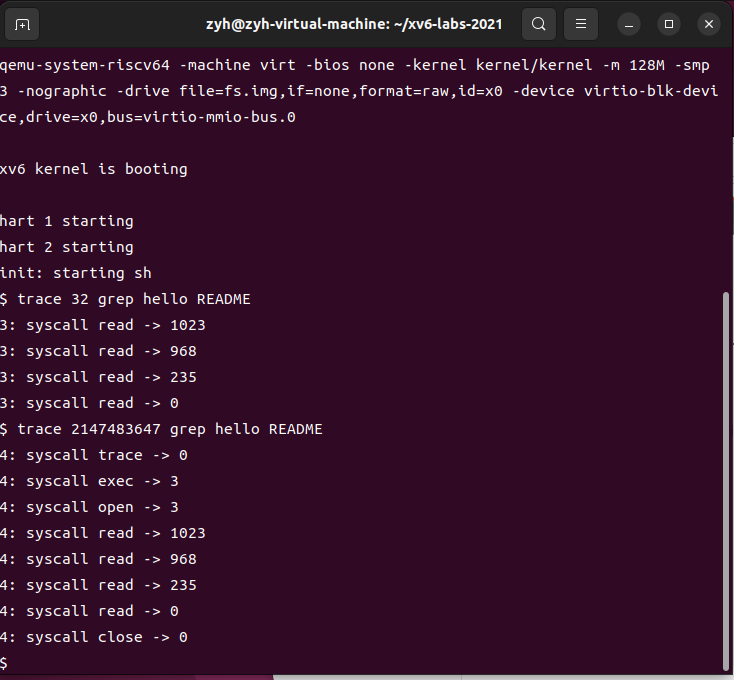


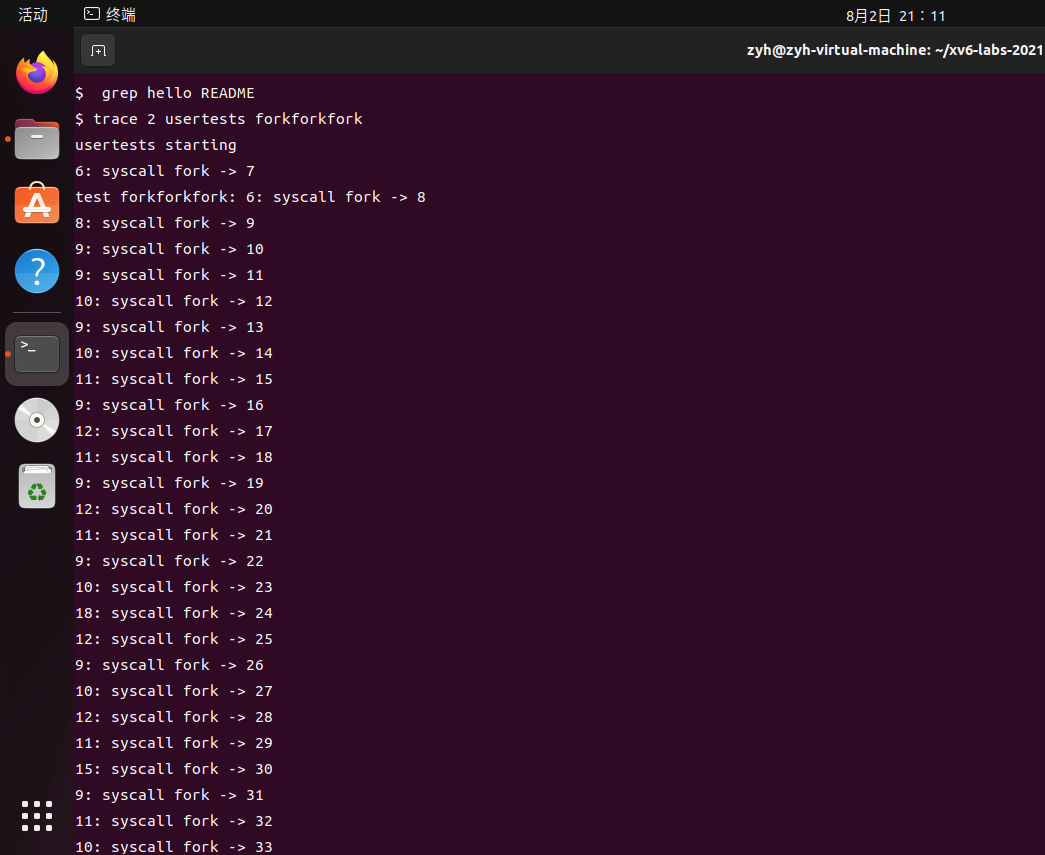


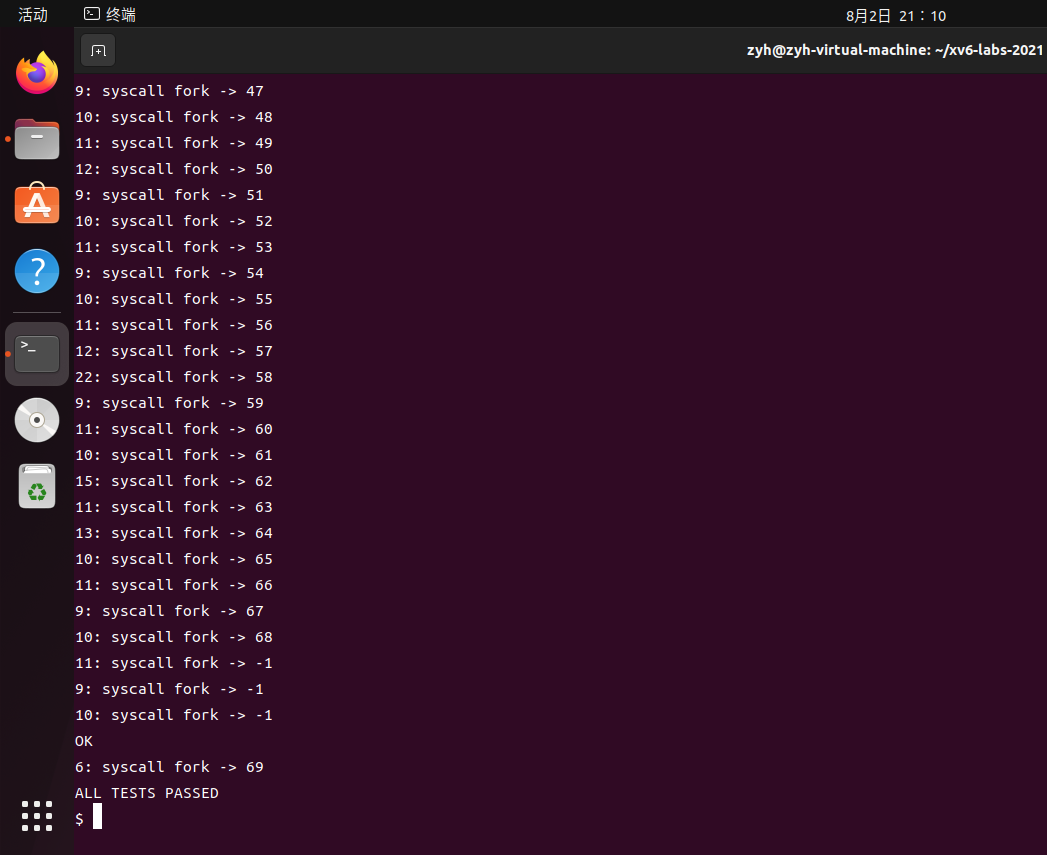


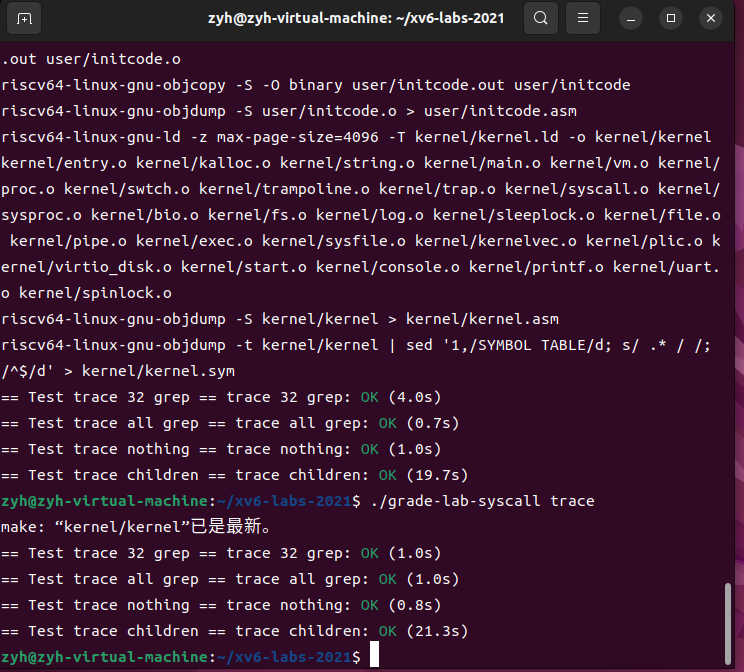


## 1.3实验结果

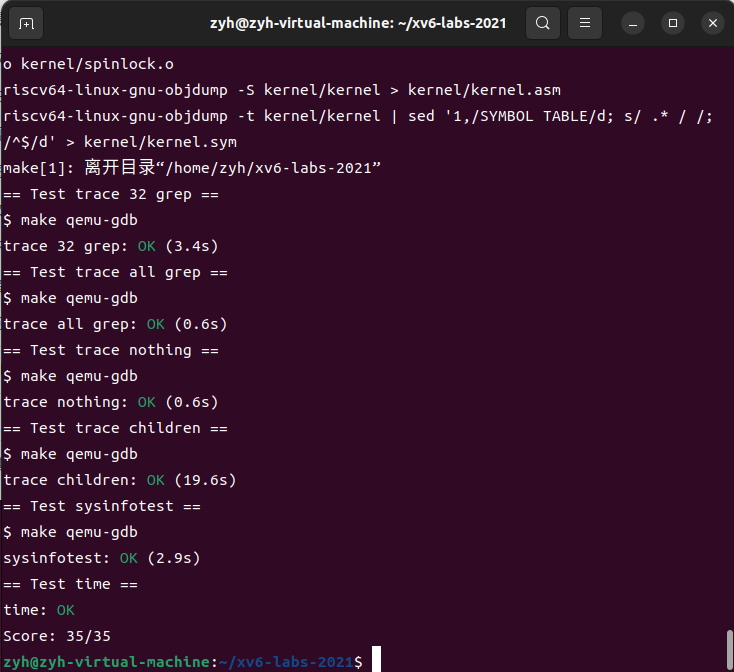








make grade:



## 1.4代码实现

### 1.4.1实现思路

syscall函数是每次调用系统调用时都是执行的，所以打印进程ID、系统调用名称及其返回值的工作会在这个函数里进行。但我们只输出指定的系统调用的调用情况，而指定信息由sys\_trace函数获取。于是，在这两个函数之间应该要通信，这样syscall函数才能在该输出时输出，不该输出时就不输出。

为了实现两个函数之间的通信，根据实验网页的提示，在kernel/proc.h的proc结构体里，我新增了一个int变量trace\_mask。同时为了该变量能正常释放内存，在freeproc函数里也要仿造其他变量添加p->trace\_mask=0。

另外，根据实验网站的提示，也为了正确响应trace 2 usertests forkforkfork这段输入，需要修改fork把父进程的trace\_mask也复制给子进程。

前置工作都搭建好后，编写sys\_trace函数和syscall函数。sys\_trace函数使用进程结构体获取当前进程，并给其trace\_mask属性赋予输入mask。syscall函数使用进程结构体获取当前进程，通过trace\_mask属性就获取到了输入的mask值。在原有syscall函数的基础上，加上if判断输出信息即可。

# 2 Sysinfo

## 2.1实验内容

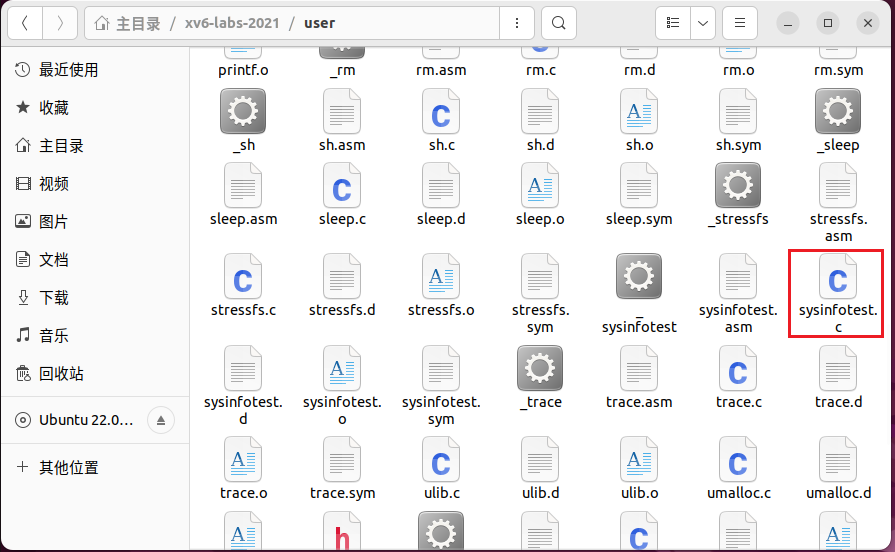
本次实验中，需要我们编写一个系统调用sysinfo来获取一些系统信息。该系统调用只有一个参数，是一个也名叫sysinfo的结构体的指针。sysinfo结构体定义在kernel/sysinfo.h文件里，它有两个属性：freemem和nproc，freemem表示内存的空闲空间，单位是字节；nproc表示正在运行的进程数目。

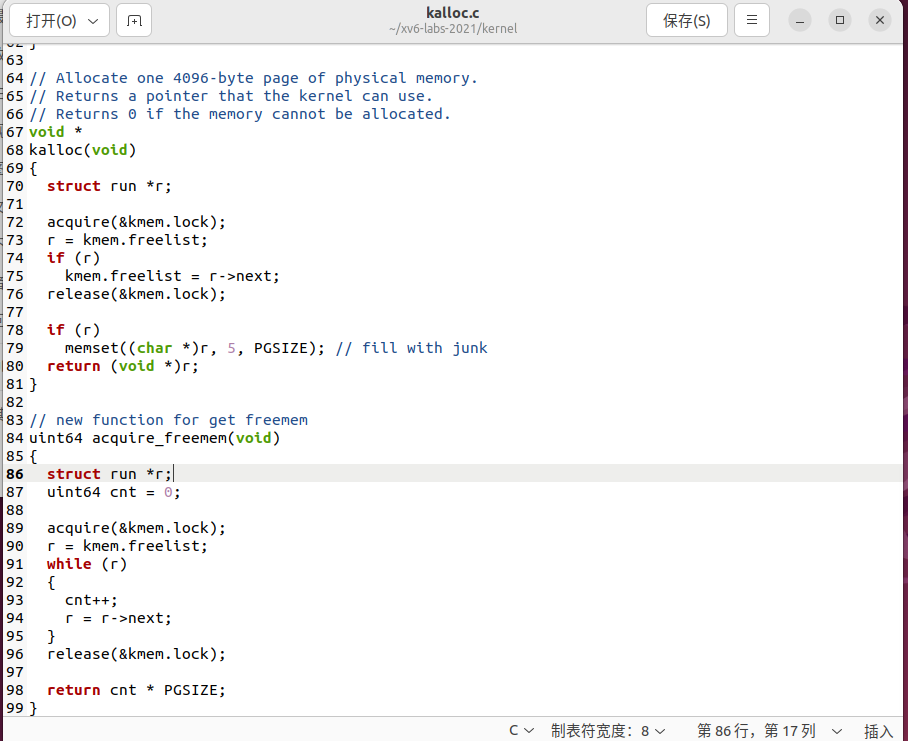
实验已实现user/sysinfotest.c，但需要切换git分支才能找到。这里我们在xv6-labs-2021文件夹下打开终端，输入git fetch，可以看到所有分支和当前分支，当前分支在util，我们要切换到实验所用的分支，输入git checkout syscall即可，步骤与上一个小实验一致。这时就能看到user/sysinfotest.c文件了。该文件里就调用了待实现的系统调用sysinfo。

在编写sysinfotest系统调用的代码的同时，还要做好配置。一，尽管实验已经给我们准备了sysinfotest.c文件，但还是要我们自己在Makefile文件中添加$U/\_ sysinfotest\。二，在user/user.h文件里要声明struct sysinfo;int sysinfo(struct sysinfo \*);系统调用。三，在user/usys.pl文件里要添加entry(“sysinfo”)。在kernel/syscall.h文件里添加#define SYS\_ sysinfo 23，这里的数字就是sysinfo这个系统调用的mask，跟着上面其他系统调用的mask定义规律来即可。四，在kernel/syscall.c文件中新增sys\_sysinfo函数定义，并在函数指针数组中新增sys\_sysinfo。

接下来，就要在kernel/sysproc.c文件里实现该系统调用。

## 2.2代码位置与截图

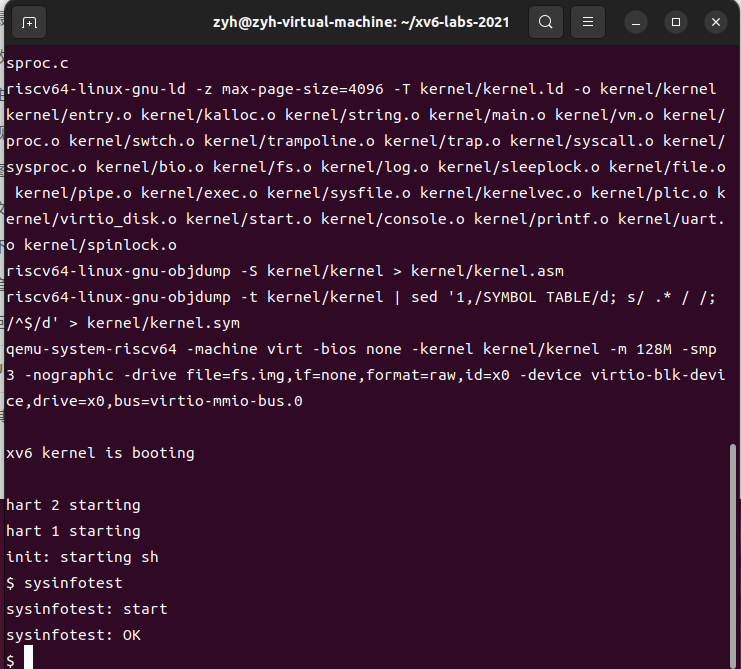


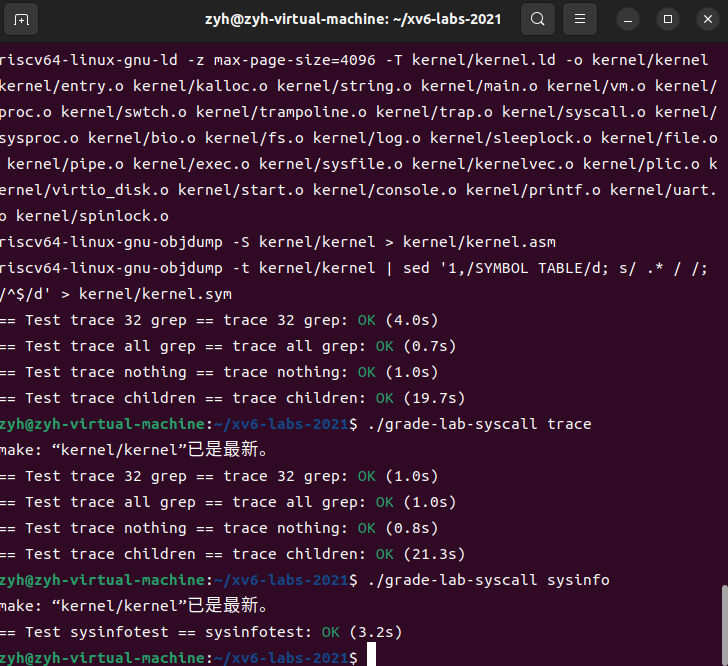




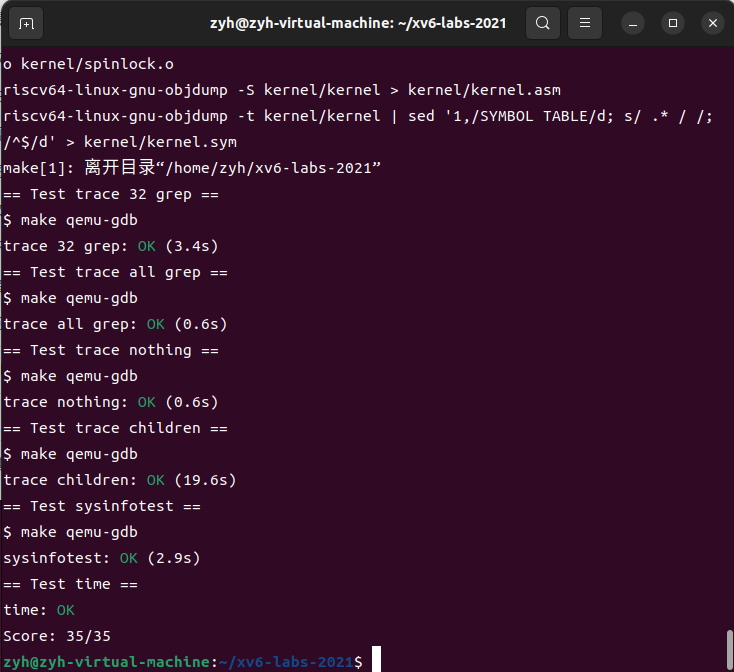


## 2.3实验结果





make grade:



## 2.4代码分析&结果分析

### 2.4.1代码实现

完成实验的关键在于如何获取当前内存的空闲空间和当前进程数。

首先，为了获取空闲的内存空间，我们需要在kernel/kalloc.c文件里添加一个函数。该文件以链表这种数据结构管理空闲页表空间，于是我们可以通过遍历链表得到空闲页表个数，再乘上单个页表的大小，就能得到空闲页表空间。这里我编写了一个函数acquire\_freemem来实现上述逻辑。

uint64 acquire\_freemem(void)

{

struct run \*r;

uint64 cnt = 0;

acquire(&kmem.lock);

r = kmem.freelist;

while (r)

{

cnt++;

r = r->next;

}

release(&kmem.lock);

return cnt \* PGSIZE;

}

接下来是获取进程数。同理，这是需要在kernel/proc.c文件里添加我们想要的函数。函数如下：

// 添加新函数来获取非不在运行的进程数目

uint64 acquire\_nproc(viod)

{

struct proc \*p;

int cnt = 0;

for (p = proc; p < &proc[NPROC]; p++)

{

acquire(&p->lock);

if (p->state != UNUSED)

{

cnt++;

}

release(&p->lock);

}

return cnt;

}

学习其他文件里的其他函数，了解怎么遍历进程。在遍历进程时，再加以判断if (p->state != UNUSED)，如为真则计数。

最后，在kernel/sysproc.c文件里写下系统调用的代码，其中需要调用上面的两个函数：

uint64

sys\_sysinfo(void)

{

struct sysinfo info;

uint64 addr;

struct proc \*p = myproc();

if (argaddr(0, &addr) < 0)

return 0;

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

return -1;

info.freemem = acquire\_freemem();

info.nproc = acquire\_nproc();

return 0;

}