**Lab 2 system calls**

1. **System call tracing (moderate)**
   1. **实验目的**

在这个实验中，通过实现 trace 系统调用，可以让用户跟踪和监控系统中发生的各种活动和事件并打印。观察到trace.c已经在用户态中给出实现，故只需要在内核中修改系统调用相关部分并在用户态中完善声明等即可。

* 1. **实验步骤**
     1. 添加声明及变量

Makefile(UPROGS): 编译声明

......

$U/\_trace\

user/user.h:系统调用函数声明

......

uint64 trace(int);

user/usys.pl:entry项

......

entry("trace");

kernel/syscall.h:系统调用码

#define SYS\_trace 22

kernel/syscall.c:系统调用函数引用

...

extern uint64 sys\_trace(void);

...

[SYS\_mmap] sys\_trace,

}

kernel/proc.h:为proc结构体添加掩码字段traceMask

struct proc {

......

uint64 traceMask; // 掩码 用于识别是否执行了对应标记的系统调用

};

* + 1. 实现系统调用函数sys\_trace，更改syscall函数

kernel/sysproc.c

uint64 sys\_trace(void)

{

int mask;

if(argint(0, &mask) < 0)

return -1;

myproc()->traceMask = mask;

return 0;

}

...

void

syscall(void)

{

int num;

struct proc \*p = myproc(); // 获取当前进程的 proc 结构

num = p->trapframe->a7; // 从 a7 寄存器中获取系统调用号

if(num > 0 && num < NELEM(syscalls) && syscalls[num]) { // 调用相应的系统调用函数，将返回值存放在 a0 寄存器中

p->trapframe->a0 = syscalls[num]();

if (p->traceMask & (1 << num)) { // 根据判断是否需要trace这个系统调用

// 1 << num的结果被用作一个掩码,用来检查进程是否设置了要跟踪特定的系统调用()

printf("%d: syscall %s -> %d\n", p->pid, sysnames[num], p->trapframe->a0);

}

} else {

printf("%d %s: unknown sys call %d\n",

p->pid, p->name, num);

p->trapframe->a0 = -1;

}

}

* + 1. 进程初始化时初始化进程追踪设置，根据实验提示：fork函数需要修改使子进程能够继承父进程追踪设置

kernel/proc.c。

static struct proc\*

allocproc(void)

{

......

//初始化进程掩码

p->traceMask = 0;

return p;

}

......

int

fork(void)

{

......

//子进程继承父进程追踪设置

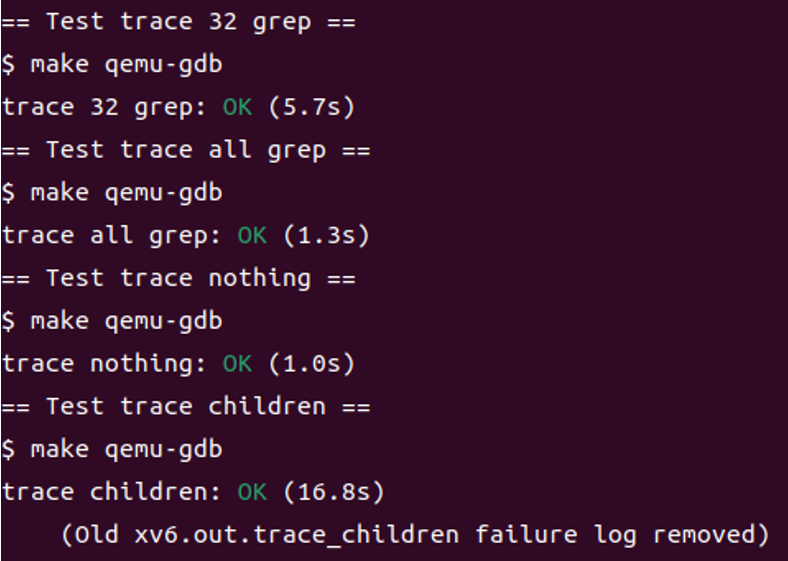
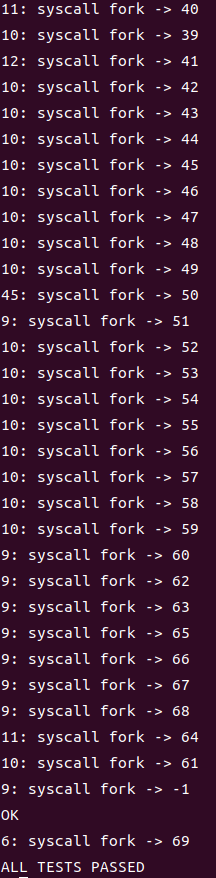
np->traceMask = p->traceMask;

release(&np->lock);

return pid;

}

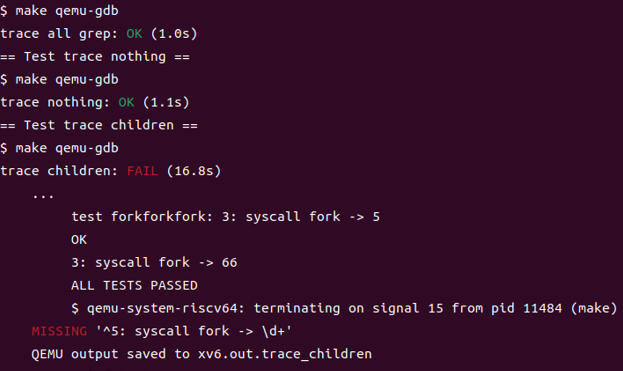
* + 1. make测试



* 1. **实验中遇到的问题和解决方法**

问题：make qemu的测试中全部正确但make grade中trace forkforkfork部分出错。

解决方法：发现一开始没有进行step3的子进程继承父进程追踪设置，导致子进程没有正确初始化。



解决方法：发现忽视了实验手册的：修改 fork()（参见 kernel/proc.c），需要将父进程的跟踪掩码复制到子进程。在对应之处进行修改。

int

fork(void)

{

......

//子进程继承父进程追踪设置

np->traceMask = p->traceMask;

release(&np->lock);

return pid;

}

* 1. **实验心得**

在Trace系统调用实验中，我学习了如何在xv6内核中实现一个跟踪功能，允许程序员追踪指定系统调用的执行情况。通过修改内核中的syscall函数，我可以检测系统调用号并根据跟踪掩码决定是否打印出跟踪信息。这个实验帮助我更深入地理解了xv6内核的结构，以及如何在内核中实现自定义功能。学习并熟悉了后续更多实验的基础操作流程（例如添加自定义函数、系统调用等）

1. **Sysinfo (moderate)**
   1. **实验目的**

基于在kernel/sysinfo.h中给出的sysinfo结构体（包括空余内存和进程数目两个int型数据）实现sysinfo的系统调用函数，使用户态可以查询空闲内存。

xv6的文件inode结构体中，包括了12个直接块与1个间接块，共能支持12+1\*256=268块（xv6中BSIZE=1024），即文件大小限制为268\*1024bytes.本实验要通过减少1个直接块，增加1个双重间接块来实现支持11+1\*256+256\*256=65803块的文件大小实现。

* 1. **实验步骤**
     1. 同上一个实验，根据实验手册提示，添加系统调用、用户态中的声明等

Makefile(UPROGS): 编译声明

......

$U/\_sysinfotest\

......

user/user.h:系统调用函数声明

......

uint64 sysinfo(struct sysinfo\*);

user/usys.pl:entry项

......

entry("trace");

entry("sysinfo");

kernel/syscall.h:系统调用码

#define SYS\_trace 22

#define SYS\_sysinfo 23

kernel/syscall.c:系统调用函数引用

...

extern uint64 sys\_sysinfo(void);

...

[SYS\_mmap] sys\_sysinfo,

}

kernel/proc.h:为proc结构体添加掩码字段traceMask

struct proc {

......

uint64 traceMask; // 掩码 用于识别是否执行了对应标记的系统调用

};

* + 1. 分别编写获取空余内存和进程数量的函数

要收集空闲内存的数量，可以在 kernel/kalloc.c 中添加一个函数。

要收集进程数量，可以在 kernel/proc.c 中添加一个函数。

根据以上实验提示可以进行kfreenum和n\_proc函数的添加及相关头文件中的声明等

kernel/kalloc.c

//遍历所有freelist获取内存页数,乘页大小即为空闲内存

int

kfreemem(void)

{

struct run \*r = kmem.freelist;

int n = 0;

while (r) {

n++;

r = r->next;

}

return n \* PGSIZE;

}

kernel/kalloc.c

//遍历proc数组,获取进程数目

int

n\_proc(void)

{

struct proc \*p;

int n = 0;

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

if (p->state != UNUSED)

n++;

return n;

}

* + 1. 编写sysinfo函数，调用上述函数获取信息并写入用户态内存

kernel/syscall.c

//获取空闲内存与进程数量的系统信息并写入用户态

uint64

sys\_sysinfo(void) {

struct proc \*my\_proc = myproc();

uint64 p;

if(argaddr(0, &p) < 0) // 获取用户提供的buffer地址

return -1;

struct sysinfo my\_sysinfo;// 构建sysinfo结构体并初始化

my\_sysinfo.freemem = kfreemem();

my\_sysinfo.nproc = n\_proc();

// 复制该struct到用户态地址里去

if(copyout(my\_proc->pagetable, p, (char \*)&my\_sysinfo, sizeof(my\_sysinfo)) < 0)

return -1;

return 0;

}

并且在defs.h中对应位置添加声明

...

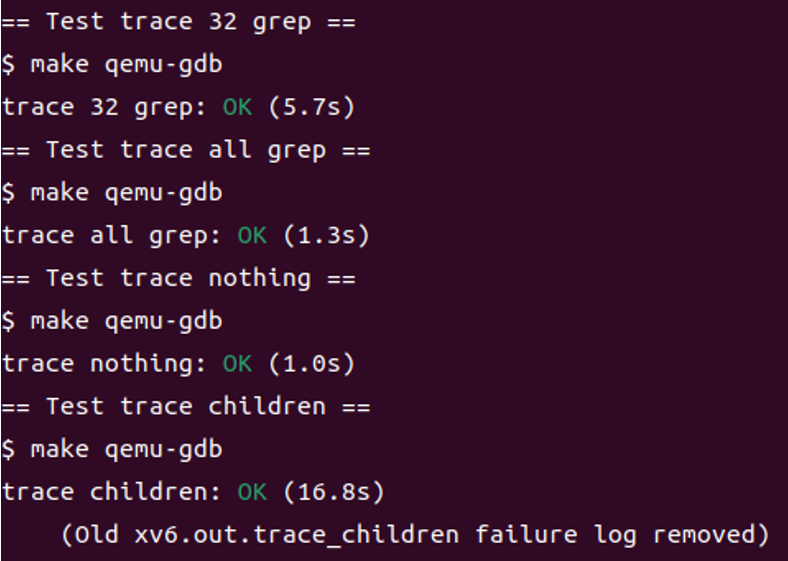
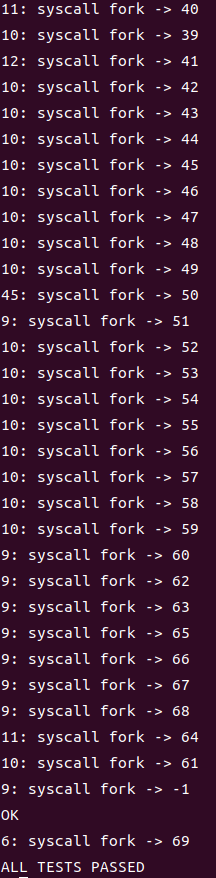
int kfreemem(void);

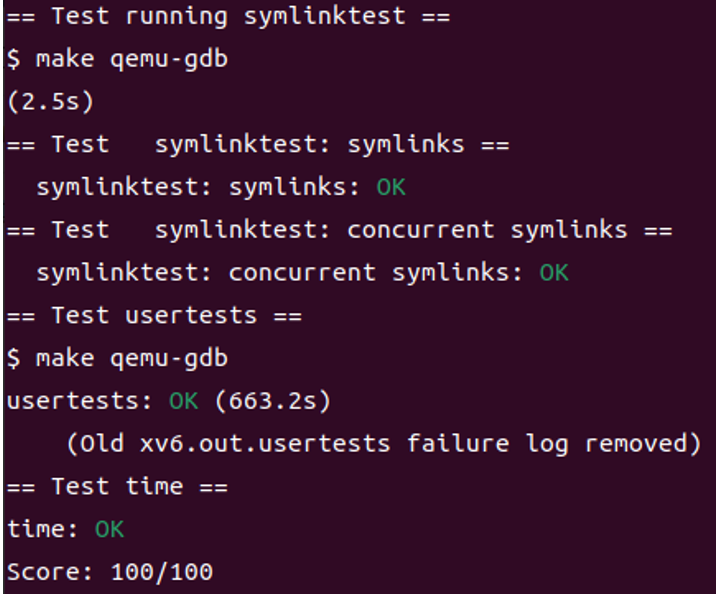
...

int n\_proc(void);

...

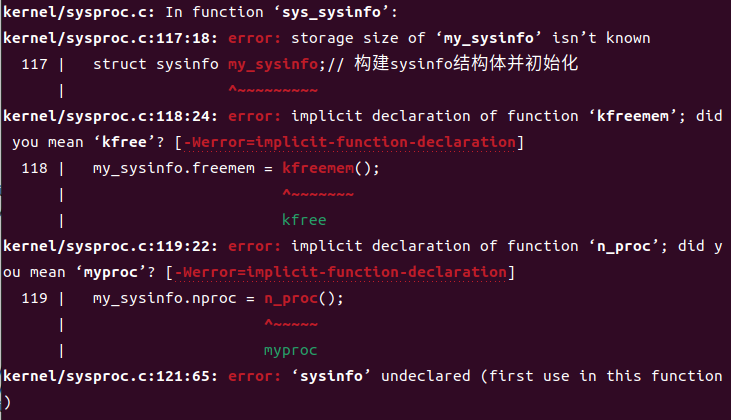
* + 1. make测试





* 1. **实验中遇到的问题和解决方法**

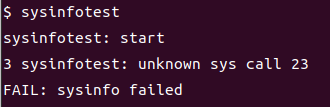
问题：



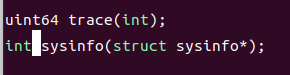
解决方法：第一个及第四个报错是头文件包括错误导致的，仅需在sysproc.c中#inlucde "sysinfo.h"即可

第二三个报错原因为缺失了step3的在kernel/defs.h中声明两个函数的步骤。

问题：出现未知系统调用。



解决方法：检查syscall.c与syscll.h后，查询网络发现类似错误，随后发现笔误：user/user.h中声明的函数返回类型(int)与syscall.c中不匹配(uint64) 。改正为uint64即可。



* 1. **实验心得**

Sysinfo系统调用实验让我实现了一个新的系统调用，用于收集系统信息。通过在内核中添加sysinfo函数，我可以将有关内存和进程数的信息填充到传递的结构中，并通过复制函数将这些信息返回给用户空间。这个实验帮助我熟悉了系统调用的添加和实现过程，同时也加深了我对进程管理和内存分配的理解。