Lab 2 System calls

目录

[1. System call tracing 1](#_Toc174490857)

[1.1 实验目的 1](#_Toc174490858)

[1.2 实验步骤 1](#_Toc174490859)

[1.3 实验结果 3](#_Toc174490860)

[1.4 遇到的问题 4](#_Toc174490861)

[1.5 实验心得 5](#_Toc174490862)

[2. Sysinfo 6](#_Toc174490863)

[2.1 实验目的 6](#_Toc174490864)

[2.2 实验步骤 6](#_Toc174490865)

[2.3 实验结果 8](#_Toc174490866)

[2.4 实验中遇到的问题和解决方法 8](#_Toc174490867)

[2.5 实验心得 8](#_Toc174490868)

[3. 测试结果 9](#_Toc174490869)

System call tracing

* 1. 实验目的
* 理解和熟悉如何在内核中处理系统调用参数、进程管理和系统调用的相关代码。
* 深入了解操作系统内核的工作原理，特别是系统调用和进程管理方面。
* 学会如何修改内核以添加新的系统调用功能，并提高调试和修改内核代码的能力。
  1. 实验步骤
* 添加一个系统调用跟踪功能，这可能在以后的实验中对进行调试有帮助。创建一个新的trace系统调用来控制跟踪。它应该接受一个参数，一个整数"mask"，其位指定要跟踪的系统调用。例如，要跟踪fork系统调用，一个程序调用trace(1 << SYS\_fork)，其中SYS\_fork是来自kernel/syscall.h的系统调用编号。需要修改xv6内核，在每个系统调用即将返回时打印一行信息，如果系统调用的编号在掩码中被设置。该行应包含进程ID、系统调用的名称和返回值；不需要打印系统调用的参数。
* 在user/user.h中添加系统调用的原型

int trace(int);

* 在user/usys.pl中添加一个存根

entry("trace");

该文件会被汇编成 usys.S

* 在 kernel/syscall.h 中添加系统调用编号

#define SYS\_trace 22

* 在 kernel/proc.h 中定义的 proc 结构(用于维护进程状态)中添加 mask 变量用于记录掩码

int mask;

* 在 kernel/sysproc.c 中添加从用户态到内核态系统调用的参数传递入口，使用 argint() 函数获取用户态的系统调用命令的参数，将参数存储在 proc 的新变量中

uint64 sys\_trace(void)

{

int mask;

argint(0, &mask);

myproc()->mask = mask;

return 0;

}

* 在 fork() 函数中加入代码，使得 fork() 在复制时同时把掩码进行复制传递，父进程将自己的mask复制给子进程，以跟踪子进程的特定系统调用

np->mask = p->mask;

* 在 kernel/syscall.c 中新建一个系统调用号到名称的索引，实现在 syscall() 函数中输出
* syscall() 函数中实现输出逻辑，需要输出的值有 pid ，系统调用名和返回值，其中返回值存储在 trapframe->a0 中

void syscall(void)

{

int num;

struct proc \*p = myproc();

num = p->trapframe->a7;

if (num > 0 && num < NELEM(syscalls) && syscalls[num])

{

// Use num to lookup the system call function for num, call it,

// and store its return value in p->trapframe->a0

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

if (p->mask & (1 << num))

{

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

}

}

else

{

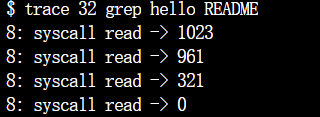
printf("%d %s: unknown sys call %d\n", p->pid, p->name, num);

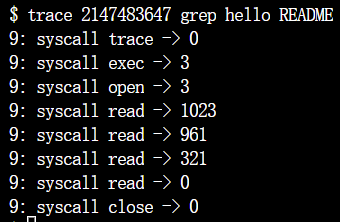
p->trapframe->a0 = -1;

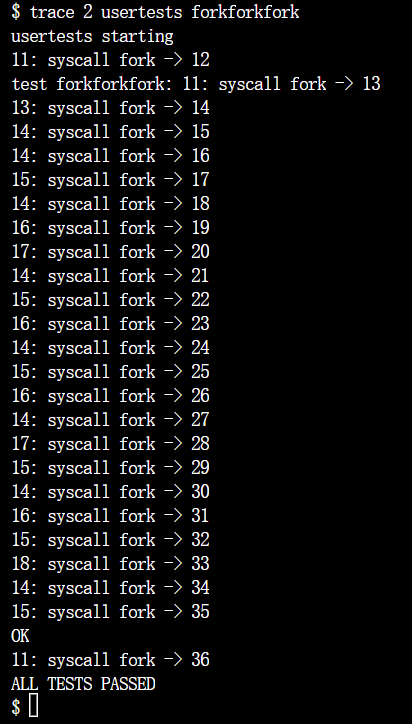
}

}

* 1. 实验结果
* 在xv6 shell中运行程序进行测试，结果如下：







* 1. 遇到的问题
* 问题：

不知道如何将系统调用参数从用户空间传递到内核空间

* 解决方法：
* 在内核中，可以使用 argint 、 argptr 和 argfd 等函数来获取系统调用的参数。这些函数定义在 kernel/syscall.c 文件中，可以用来从用户空间获取整数、指针和文件描述符等类型的参数。
* 问题：

不知道如何在xv6中添加新的系统调用

* 解决方法：

在xv6中添加新的系统调用需要进行多个步骤，包括在用户空间和内核空间进行修改。需要在 user/user.h 中添加系统调用的原型，在user/usys.pl 中添加存根，在 kernel/syscall.h 中添加系统调用编号。同时，还需要在kernel/sysproc.c 中添加相应的系统调用处理函数。

* 1. 实验心得

在这个实验中，我学到了如何在操作系统内核中添加自定义系统调用并实现系统调用跟踪功能。通过在用户空间和内核空间之间传递参数，我掌握了系统调用扩展的基本方法。

我还学会了在内核中定义和操作数据结构，例如在进程结构中添加掩码变量，以控制系统调用的跟踪。通过修改 fork 函数来复制掩码，我进一步理解了进程间信息传递的机制。

此外，我了解了如何创建系统调用号与名称的映射表，并在 syscall 函数中输出调试信息。这一过程使我熟悉了内核编程和调试的方法，为今后的系统开发奠定了基础。

Sysinfo

* 1. 实验目的
* 添加sysinfo系统调用，并在内核中进行相应的实现。
* 填充struct sysinfo结构的字段，计算并记录系统中的可用内存和活动进程数量。
* 学习如何在内核中处理系统调用参数和结果的传递，以及如何将结果复制回用户空间。
* 加深对内核源代码的理解，特别是与进程管理、内存管理和系统调用相关的部分。
  1. 实验步骤

需要添加一个名为sysinfo的系统调用，用于收集关于运行中系统的信息。该系统调用接受一个参数：指向struct sysinfo结构的指针（请参阅kernel/sysinfo.h）。内核应该填充该结构的字段：freemem字段应设置为可用内存的字节数，nproc字段应设置为状态不是UNUSED的进程数量。提供了一个测试程序sysinfotest；如果它打印出"sysinfotest: OK"，则代表通过了这个任务。

* 在 Makefile 的 UPROGS 中添加 $U/\_sysinfotest
* 预先声明 struct sysinfo

struct sysinfo;

int sysinfo(struct sysinfo \*);

* 同上个实验，添加 sysinfo 系统调用。在 user/user.h 中声明 sysinfo() 的原型
* 可以参考 kernel/sysfile.c 中的 sys\_fstat() 和 kernel/file.c 中的 filestat() ，使用copyout() 来实现拷贝操作，把数据从内核栈拷贝到用户栈。其中 sysinfo 需要将一个 struct sysinfo 拷贝回用户空间。
* 编写 sys\_sysinfo() 函数：

uint64 sys\_sysinfo(void)

{

uint64 addr;

argaddr(0, &addr);

struct sysinfo info;

info.freemem = countMEM();

info.nproc = countProc();

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

{

return -1;

}

return 0;

}

其中， argaddr() 的作用是将系统调用参数作为指针读入。

* 为了收集可用内存的数量，在 kernel/kalloc.c 中添加一个函数 countMEM() ：

uint64 countMEM()

{

struct run \*p = kmem.freelist;

uint64 count = 0;

while (p)

{

count++;

p = p->next;

}

return count \* PGSIZE;

}

* 为了收集进程的数量，在 kernel/proc.c 中添加一个函数 countProc() ：

uint64 countProc()

{

uint64 count = 0;

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

{

if (proc[i].state != UNUSED)

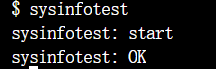
count++;

}

return count;

}

* 1. 实验结果
* 在xv6 shell中运行程序进行测试，结果如下：



* 1. 实验中遇到的问题和解决方法
* 问题：

argaddr 函数的使用没有掌握

* 解决方法：

argaddr 函数用于获取系统调用的第 n 个参数，它将参数作为指针返回。 argaddr(0, &addr) 获取系统调用的第一个参数，该参数是一个指向 struct sysinfo 的指针。这是因为 sys\_sysinfo 系统调用的第一个参数应该是一个指向 struct sysinfo 的指针，用于存储系统信息。

* 1. 实验心得

在这个实验中，我成功实现了 sysinfo 系统调用，用于收集系统的内存和进程信息。通过编写 countMEM() 函数，我深入理解了操作系统的内存管理机制，学会了如何遍历内核的空闲内存链表来统计可用内存量。在编写 countProc() 函数时，我熟悉了进程状态管理，并学会了统计当前活跃的进程数量，从而加深了对进程调度的理解。

此外，我通过使用 copyout() 函数，将内核中的数据结构安全地传递到用户空间，这一过程让我掌握了内核与用户空间之间的数据交互技术，这在系统编程中非常关键。最后，通过运行 sysinfotest 测试程序验证了 sysinfo 系统调用的功能，确保了代码的正确性和稳定性。

测试结果

