

Lab2.系统调用

在进行本实验之前，你需要进行分支管理，请在工程目录下输入以下命令：

```
git fetch
git checkout lab2
make clean
```

xv6中的系统调用

系统调用是操作系统提供服务的接口，相信在操作系统的理论课程中你们已经学习过什么是系统调用了，在接下来的实验中将会通过xv6来进一步了解系统调用，看看在代码层面，系统调用是怎么样的。

在xv6中也内置了一些系统调用，比如说 `read`，`write`，`exit`，`open`，`fork` 等，更多的细节可以在源码中查看，[xv6 book](#)中第一章也提供了xv6所有的系统调用的列表。

在正式探索系统调用的整个过程之前，我们需要在用户空间使用一些系统调用来实现程序，来看看系统调用是怎么和我们发生交互的。

查看系统调用的列表我们可以发现在xv6中是存在 `sleep` 这个系统调用的，但是，当我们在系统中使用相应的用户程序的时候，却报错了：

```
exec sleep failed
```

这是因为在用户空间中我们还没有实现这个程序。事实上，在用户空间中调用 `sleep` 的系统调用，其实就是在使用内核中函数 `sys_sleep` 的功能。

在文件 `user/usys.s` 中我们可以看到这样一段代码：

```
.global sleep
sleep:
    li a7, SYS_sleep
    ecall
    ret
```

这就是使用汇编代码帮助 `sleep` 从用户空间跳到内核空间。更具体的细节会在下面讲解。

接下来，我们需要在用户空间中写一个 `sleep` 程序来使用该系统调用。

实现sleep

在 `user/sleep.c` 中填写你的代码实现，如何获取传递给程序的命令行参数，可以参照 `user/echo.c`。如果用户忘记在使用 `sleep` 时传入函数，你也需要打印错误信息。在实现过程中，直接使用系统调用 `sleep` 即可，内核中已经为你写好。

此外，命令行参数是作为字符串传递的，而 `sleep` 的功能是传入一个数字作为参数，暂停相应的秒数，所以你需要将字符串转化为数字，`xv6` 的字符串处理函数中已经内置有该函数，请仔细阅读源码找出并使用它。

最后，你还需要将你的 `sleep` 程序添加到 `Makefile` 中的 `UPROGS` 中，这样你就可以在 `xv6` 中测试你的 `sleep` 了。我们还给你准备了一些测试样例来帮助你判断自己的实现是否正确，使用 `python3 grade-lab-2 sleep` 可以测试你的程序。

完成 `sleep` 的实现之后，你发现你似乎可以给你的 `xv6` 添加你想要的功能了。

在这里，再给你们一个编程练习，通过这个练习你们可以更加熟练地掌握如何添加自己想要的用户程序，以及更多了解到 `xv6` 中的一些命令行工具是怎么实现的。

实现find

在 `user/find.c` 中填写你的代码实现，如果你不知道怎么实现，查看 `user/ls.c` 中的代码并思考 `ls` 是怎么实现的，`ls` 和 `find` 的实现非常相近，你只需要修改少量代码即可实现。

需要允许 `find` 递归下降到子目录中，但是不要在“.”和“..”目录中递归。使用 `python3 grade-lab-2 find` 可以测试你的程序。

注意！

对文件系统的更改会在 `qemu` 的运行过程中一直保持；要获得一个干净的文件系统，请运行 `make clean`，然后 `make qemu`。

在find程序中支持正则表达式

在 `find` 程序的名称匹配中支持正则表达式。什么是正则表达式？简单来说正则表达式是一组由字母和符号组成的特殊文本，它可以用来从文本中找出满足你想要的格式的句子。如果你想知道更多关于正则表达式的内容，可以自己去找答案，[这里](#)也可以快速学习正则表达式。

至于在这里如何支持正则表达式？`user/grep.c` 或许会给你一些启发。

从用户空间到内核空间

值得思考的问题是，系统调用从用户空间到内核空间的这个过程，发生了什么？上文 `user/usys.s` 中存在一个 `ecall` 指令，这就是灵魂所在，`ecall` 指令是一个特殊指令，用于发起系统调用，它的作用是将控制权转移到操作系统内核中的一个特定地址，以执行特权级操作。而这种用户程序执行 `ecall` 指令的情况，其实是 `trap` 的其中一种。

接下来我们会从 `trap` 的一些细节开始，探索系统调用从用户空间到内核空间的过程。我们会通过使用 `gdb` 跟踪如何在 `shell` 中调用 `write` 系统调用来探索这个过程。

从用户空间程序的角度来说，调用 `write` 其实就是代码中的 C 函数的调用。而在上文我们提到，在文件 `user/usys.s` 中的 `ecall` 指令就是灵魂所在。

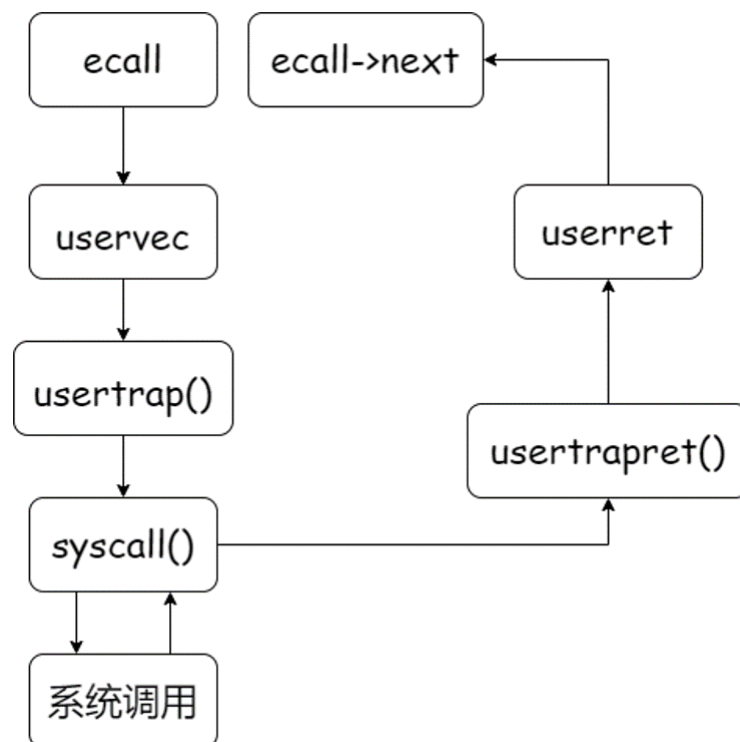
```
.global write
write:
    li a7, SYS_write
    ecall
    ret
```

从代码中可见，函数 `write` 使用 `ecall` 指令跳转到内核态。在这个过程中，内核中执行的第一个指令是一个由汇编语言写的函数，叫做 `uservec`。在这里我们不需要过度关心 `uservec` 的细节。之后，在 `uservec` 函数中，代码执行跳转到了由 C 语言实现的函数 `usertrap`。跳转到了 C 语言的函数，我们便能更好的理解了。在 `usertrap` 函数中，我们执行了一个叫做 `syscall` 的函数。

阅读 `kernel/syscall.c`，我们会发现会有一个对系统调用编号进行映射的数组，而在 `syscall` 函数中，会根据我们在 `ecall` 指令之前传入的系统调用编号进行查找，在 `kernel/syscall.h` 中我们会发现每一个系统调用都会有一个对应的编号，在这里 `SYS_write` 对应的编号为，而对应的函数就是 `sys_write`，这个函数的所作所为就是 `write` 这个系统调用的所作所为。完成之后，会返回到 `syscall` 函数。

接下来我们需要回到用户空间。而之前的 `ecall` 指令中断了用户空间代码的执行，恢复到用户空间，需要做一系列的事情，而 `usertrap` 函数的最后会执行 `usertrapret` 函数，该函数会完成这部分工作。当然，有一些工作只能由汇编代码完成，所以 `usertrapret` 函数最后会跳转到汇编代码执行 `userret` 函数。

可以用一幅图来概括这个过程：



这就是系统调用执行的整个过程，接下来需要使用gdb来走一遍这一个过程。

这个过程要从xv6的shell最原始的命令提示符光标“\$”说起。相信完成了Lab1，你们已经知道“\$”源自于user/sh.c文件中的getcmd函数，而且与一个write系统调用相关。我们就从这个系统调用的ecall指令开始调试。

在ecall指令处打一个断点

查看相应的汇编代码，找到ecall指令的地址，在此处打一个断点。此处每个人生成的地址可能会不同，请自己动手完成。

接下来可以打印程序计数器，验证我们所处的地址。

此外还可以打印寄存器查看，这里面包含了不少信息，比如说寄存器a0，a1，a2是Shell传递给write系统调用的参数，程序计数器pc和堆栈指针sp的地址现在都在距离0比较近的地址，证明现在的代码在用户空间。

接下来我们需要单步进入ecall指令调用的函数了。可是，你可能会发现，进入之后的地址可能很小，并不是一个内核地址。好吧，这是一个gdb-multiarch的小问题，接下来你需要想办法进入ecall指令进入的内核地址。

进入下一条指令在内核中的地址

解决方法其实很简单，你只需要在某个与trap相关的寄存器所存的地址处打一个断点再进入即可，至于是哪一个地址，自己仔细想想，看看跟trap相关的几个寄存器都是什么作用。

进入之后你会发现，这条指令就是 `uservec` 函数中的第一条指令。此时如果查看寄存器可以发现，相对于进入内核之前而言寄存器的值并没有发生变化。`uservec` 函数后面的指令会将用户空间的寄存器保存，在此之前我们不能使用任何寄存器，不然恢复到用户空间时程序可能会出错。

接下来你就可以一步步观察上文中描述的整个过程的每一条指令的行为和各种变量的值的变化，了解到是怎么引起系统状态变化的。完成之后可以看一下，回到用户空间后的下一条指令，和没有解决进入内核地址的问题之前跳转到的下一条指令是否相同。

💡 认真走一遍这个过程

知道你们很想偷懒，觉得把上面的文字描述看完以及肉眼看看代码就懂了，但是建议你们还是好好地亲手走一遍，不然很多细节可能会被忽略掉。

那么为什么系统调用要设计这么复杂的机制呢？其实很重要的一点来自于用户空间和内核空间的隔离性，内核不能信任用户空间的任何内容，防止被恶意软件入侵导致系统崩溃。

📖 扩展阅读：关于隔离性

隔离性的思想很简单，顾名思义，就是把应用程序隔离开来。在用户空间中有多个程序，当一个程序出问题时，我们并不希望该程序会影响到其他程序的运行。所以我们在不同的程序之间要有强隔离性。

操作系统其实也是一个程序，我们不希望在一个应用程序出问题时操作系统因此而崩溃。比如说操作系统接收到了一些奇怪的参数，但我们希望操作系统能够很好地处理这些奇怪的参数，即可以很好地处理异常情况，所以操作系统和应用程序之间自然也需要有强隔离性。

如果没有操作系统，应用程序就可以直接和底层的硬件交互，比如说CPU就暴露在了应用程序眼中，同时也可以看到磁盘、内存等。这种情况是会导致应用程序之间的隔离性被破坏，当应用程序在CPU上运行时，理应让别的程序也有机会运行，但这时没有了操作系统，操作系统没有提供虚拟化CPU的假象，如果此时运行着的程序中出现了死循环，那么它将永远霸占CPU，我们甚至无法运行杀死它的程序。

不仅是CPU，内存也是如此，同样的，内存也存在虚拟化现象，让程序以为自己独自霸占了所有的物理内存，但事实上不是，如果两个程序在运行时出现抢占对方内存的情况，覆盖了对方内存中的内容，那么程序的运行必然会出问题。

由此可见，隔离性是非常重要的，系统调用的接口是精心设计的，通过抽象硬件资源，提供了强隔离性的功能。

经历了上面的过程，相信你已经对xv6中是如何设计系统调用的有了足够的了解，你可以自己设计系统调用了。

你们当中的一些人或许有听说过 `strace` 这个工具，这是一个很有用的工具，可以跟踪程序执行的系统调用，让我们打开程序的执行，看看一个程序执行背后的魔法是什么。尝试使用一下 `strace` 吧，你可以写一个简单的 `Hello, world` 程序，生成可执行文件 `a.out`，然后用以下命令看看该程序执行了哪些系统调用：

```
strace .a.out
```

当然，你还可以看看 `gcc` 在编译程序时会执行哪些系统调用：

```
strace -f gcc hello.c
```

感受到该工具的好用之处之后，接下来的任务就是在 `xv6` 中实现一个简单的系统调用跟踪功能了。用户空间的程序 `trace` 已经准备好，你可以在 `user/trace.c` 中查看。

实现系统调用 `trace`

你需要创建一个新的 `trace` 系统调用来控制跟踪。它应该有一个参数，这个参数是一个整数，称为 `mask`，它的比特位指定要跟踪的系统调用。例如，要跟踪 `fork` 系统调用，程序调用 `trace(1 << SYS_fork)`，其中 `SYS_fork` 是 `kernel/syscall.h` 中的系统调用编号。如果在 `mask` 中设置了系统调用的编号，则在每个系统调用即将返回时打印出一行。该行应该包含进程 `id`、系统调用的名称和返回值；不需要打印系统调用参数。`trace` 系统调用应启用对调用它的进程及其随后派生的任何子进程的跟踪，但不应影响其他进程。

需要在 `kernel/sysproc.c` 中添加一个 `sys_trace()` 函数，它通过将参数保存到 `proc` 结构体里的一个新变量中来实现新的系统调用。

修改 `kernel/syscall.c` 中的 `syscall()` 函数以打印跟踪输出。还需要修改 `fork()` 将 `mask` 从父进程复制到子进程。

注意，还需要在 `user/usys.pl` 处添加该系统调用，以在 `user/usys.S` 中自动生成指令。

使用 `python3 grade-lab-2 trace` 可以测试你的程序。

打印所跟踪的系统调用的参数

真实世界中的 `trace` 可以打印非常多的信息，而且打印更多的信息也有助于日后利用工具进行调试。你可以进一步完善 `trace`，打印所跟踪的系统调用的参数，也可以实现更多的功能。