Lab2：系统调用

实验前，基于之前下载的 xv6 的源码，请各位贵宾切换到 lab2 的分支：

```
$ git fetch
$ git checkout syscall
$ make clean
```

注意：因为我们基于 xv6-riscv 的默认代码仓库增加了 lab1 的内容，所以，在切换的时候，终端可能会提示代码发生了变化而导致无法切换，这个时候敲 git stash，再进行切换应该可以解决问题。MIT6.828 课程的一个特点是：各个 lab 之间的代码没有相互的叠加和复用，独立性和自包含性都比较好。

本次实验，我们将往 xv6 的默认内核加入新的系统调用：trace()和 sysinfo()。在动手之前，请先至少通读一遍“中文实验文档”（本文档），之后再务必多读几遍本次实验的“官方实验指南”，并在必要的时候翻阅“官方参考书”的相关章节：

官方实验指南：https://pdos.csail.mit.edu/6.828/2021/labs/syscall.html

官方参考书：https://pdos.csail.mit.edu/6.828/2021/xv6/book-riscv-rev2.pdf

2.1 预备知识

系统调用我们可以简单地理解为：OS 为应用层提供的一个功能服务函数。一般情况下，它的实现在内核层，并提供了接口导出至用户层。一个进程在执行了某个系统调用所提供的接口后，CPU 的执行模式会暂停当前正在执行的进程，并让 CPU 发生从用户层到内核层的切换（trap，陷入），让 OS 的内核去执行该系统调用的主体逻辑。完成之后，CPU 会再次切换回用户层以继续执行之前被暂停的用户代码。从某种意义上，我们可以把内核理解为一个具备高权限的特殊“进程”，而上述描述，可以看作是两次特殊的 context switching（进程上下文切换）：用户进程->内核“进程”->用户进程。

下面将给出一个具体的 xv6-riscv 系统调用流程，它并没有把每一个细节都说的清清楚楚，否则会使得内容显得过于杂乱而失去原本的主线脉络。大家如果看了之后觉得还不是太懂也完全没有关系——先了解到一个大概即可。后面的 2.2 和 2.3 的实验也并非强制要求大家去深刻理解这些流程细节。

以大家在第一个 lab 用过的 pipe()系统调用为例，xv6 中系统调用的处理流程如下：

（1）在应用层，编译器，例如 gcc，发现程序中有“pipe”关键字，便会根据 xv6-riscv 系统初始化的信息，判断其为 xv6-riscv 的系统调用，于是，将 pipe()的参数传入 RISC-V 的 CPU 寄存器 a0—a5 中，随后调用 usys.S 中对应的汇编代码（在 xv6 目录中执行

make qemu 命令后，usys.S 文件便会由 user/usys.pl 所生成），如下所示：

```
.global pipe
pipe:
  li a7, SYS_pipe
  ecall
  ret
```

（2）上面这段汇编代码的含义是：将 pipe 的系统调用号 SYS_pipe 传入寄存器 a7 中（系统调用号的定义在 kernel/syscall.h 中），再调用 ecall 指令让 CPU 从用户态（user mode）下陷到内核态（supervisor mode）；

（3）CPU 依次执行两个代码路径，分别是：uservec（kernel/trampoline.S 的第 16 行）和 usertrap（kernel/trap.c 的第 37 行），它们所做的工作主要是：保存被中断的用户程序现场信息“至”其对应的进程结构体的 trapframe 内存页面中（trapframe 隶属于该进程地址空间中的一个页面），并将 CPU 的 PC（程序计数器）寄存器设定为“内核处理该系统调用的代码开始地址”，还要将 CPU 上相应寄存器保存的进程页表地址切换至内核页表地址（这里细节繁杂，看个大概就行，深究请参阅官方参考书的 4.1、4.2、4.3 章节）；

（4）CPU 在 kernel/syscall.c 中的第 108 行开始检查第(2)步中的 SYS_pipe 调用号是否存在已经定义好的处理入口（“处理入口”通过一个类型为“函数指针”而定义，每个系统调用都有自己的“处理入口”；因此，所有系统调用的“处理入口”形成一个看起来比较吓人的“函数指针”数组）；

（5）如果有处理入口来匹配 SYS_pipe（在这里，SYS_pipe 的函数入口名为 sys_pipe），则执行其所对应的函数体，而 sys_pipe 主体函数的实现在 kernel/sysfile.c 中的第 457 行左右；

（6）执行完毕后，内核会将该系统调用的返回值放入对应进程结构体的 trapframe 页面中；

（7）跟第（3）步对应，CPU 依次执行两个代码路径以返回至用户层，分别是：usertrapret（kernel/trap.c 的第 90 行）和 userret，它们所做的主要工作是：依据对应进程结构体的 trapframe 内存页面，恢复被中断的用户程序的现场信息，并将 CPU 的 PC（程序计数器）寄存器设定为“之前用户被中断的 PC 值”，还需要将 CPU 上相应寄存器保存的“内核页表始址”切换换回“进程页表始址”；

（8）最后，执行第（1）步代码框中的 ret 指令，CPU 回到用户态。

在上述过程的第（5）步中，处理入口，也即系统调用函数体本身，大都会对应用层传递下来的参数进行解析和使用。值得一提的是，因为应用层的数据难免会存在一些安全隐患，比如一个指针参数指向了该进程没有权限访问的内存地址，那么，xv6-riscv 的系统调用函数体本身就需要对这些数据进行检查了。这里用到的核实方法主要是遍历该进程的页表，来判断这些传下来的地址数据是否合法（详见官方参考书的 4.4 章节）。

2.2 名为 trace()的新系统调用

初步地，要让一个新加入系统的系统调用发挥作用，一般要做两个层次的工作：（1）在内核层设计和实现该系统调用的接口和执行主体；（2）在应用层设计一个用户程序，并让该程序去调用在第一个层次中新加入系统的系统调用，从而发挥作用。在 2.2 和 2.3 的实验中，第二个层次，也即应用层，xv6 已经在 user/目录提供了相应的用户程序，我们的工作只需要集中在内核层即可。

了解了 xv6-riscv 系统调用的大致工作原理后，我们来尝试加入一个名为 trace()的新系统调用。

trace()系统调用的执行逻辑是：跟踪某个进程及其子进程所执行的系统调用，并把（1）进程名、（2）被调用的系统调用名（3）及其返回值，在终端进行打印输出。这个系统调用可以为 xv6 的用户程序增加一个比较好用但是功能还比较原始的跟踪调试功能。

trace()系统调用只有一个参数，为整型的“掩码”（mask），其对应的二进制位来表示需要追踪具体的哪个系统调用，比如：trace(1 << SYS_fork)，表示用来追踪对应进程的 fork()系统调用。其中的 SYS_fork 表示 fork()系统调用的编号，而这些编号在 xv6 源码里的 kernel/syscall.h 有定义。

```
$ trace 32 grep hello README
3: syscall read -> 1023
3: syscall read -> 966
3: syscall read -> 70
3: syscall read -> 0
```

上面是一个 trace()执行的效果示例：“$ trace 32 grep hello README”中的“trace”是一个可执行的用户层程序，它的源码已经存在于 xv6 源码下的“user/trace.c”（一定要切换到 lab2 的分支才会有!）。在此“trace.c”中，包含了对系统调用“trace()”的调用。因为同名，大家不要把这里的“程序-trace“和”系统调用-trace“给搞混淆了。

该示例表示，trace 对参数 32 进行追踪，而 32 为“1 << SYS_read”的结果，因此，它的追踪对象为 read 系统调用。“grep hello README”为被追踪的进程，它的运行逻辑是在

“README”文件里查找名为“hello”的字符串，并打印包含了该字符串的行。“3: syscall read -> 1023”这一行输出中的“3”表示 grep 的进程号，“syscall read”表示被追踪的系统调用名称，“-> 1023”表示 read 的返回值。

```
$ trace 2147483647 grep hello README
4: syscall trace -> 0
4: syscall exec -> 3
4: syscall open -> 3
4: syscall read -> 1023
4: syscall read -> 966
4: syscall read -> 70
4: syscall read -> 0
4: syscall close -> 0
```

上述 trace()示例比较特殊的地方是它的参数值：2147483647。因为其转换为二进制后为：31 个 1，所以，其表示的是对 xv6 所有定义的系统调用进行追踪。可以看到该示例的输出有“trace”、“exec”、“open”、“read”和“close”等多个系统调用。

```
$ grep hello README
$
```

上述示例没有运行“用户程序-trace”，所以没有执行“系统调用-trace”，也不会有对应的系统调用跟踪输出。

```
$ trace 2 usertests forkforkfork
usertests starting
test forkforkfork: 407: syscall fork -> 408
408: syscall fork -> 409
409: syscall fork -> 410
410: syscall fork -> 411
409: syscall fork -> 412
410: syscall fork -> 413
409: syscall fork -> 414
411: syscall fork -> 415
```

上述示例中，是对用户程序 usertests 的 fork 系统调用进行追踪，而 usertests 的运行逻辑是：主进程（408）使用 fork 创建了子进程（409），而子进程（409）又用 fork 创建了子进程（410），后者同样又用 fork 创建了子进程（413）……以此类推。这个示例，是在要求大家在添加 trace 系统调用的时候，其作用范围要覆盖到被追踪进程的所有子孙进程。

下面是一些在“官方实验指南”中给出的提示，我先列出来，然后再进行翻译和补充：

> - Add $U/_trace to UPROGS in Makefile
> - Run **make qemu** and you will see that the compiler cannot compile user/trace.c, because the user-space stubs for the system call don't exist yet: add a prototype for the system call to user/user.h, a stub to user/usys.pl, and a syscall number to kernel/syscall.h. The Makefile invokes the perl script user/usys.pl, which produces user/usys.S, the actual system call stubs, which use the RISC-V ecall instruction to transition to the kernel. Once you fix the compilation issues, run **trace 32 grep hello README**; it will fail because you haven't implemented the system call in the kernel yet.
> - Add a sys_trace() function in kernel/sysproc.c that implements the new system call by remembering its argument in a new variable in the proc structure (see kernel/proc.h). The functions to retrieve system call arguments from user space are in kernel/syscall.c, and you can see examples of their use in kernel/sysproc.c.
> - Modify fork() (see kernel/proc.c) to copy the trace mask from the parent to the child process.
> - Modify the syscall() function in kernel/syscall.c to print the trace output. You will need to add an array of syscall names to index into.

根据提示，我们在添加对应系统调用的时候可以一步步来：

(1) 首先，像 lab1 那样，在 Makefile 文件里添加“user/trace.c“的对应编译选项（表示要对 user/下的 trace.c 进行编译），然后，运行 make qemu，会得到 trace.c 中“系统调用-trace”无法识别的错误或警告，这是正常的，因为当前系统里根本就没有“系统调用-trace”的接口定义和实现；

(2) 所以再根据上面的提示，在对应的文件里添加“系统调用-trace”的原型或声明（需要更改三个源码文件，分别是：user/user.h；user/usys.pl；kernel/syscall.h）；

(3) 提示里出现了 stub 这个英文单词，大家可以去搜一下，什么叫 stub 代码（对应的中文翻译叫做“桩代码“或者”桩程序“），其在比较大型的系统开发任务中经常被用到；

(4) 当加入了“系统调用-trace”的原型或声明之后，再敲 make qemu，不出意外的话，会发现 xv6 可以正常启动了；

(5) 然后一旦在 xv6 的终端里运行 trace.c 对应的可执行文件，会发现还是会报错（来自于 xv6 的报错），因为“系统调用-trace”只有个接口，内部没有完整的

实现；

(6) 所以，接下来，我们得在 kernel/sysproc.c、kernel/proc.h、还有 kernel/syscall.c 里加入对应的实现代码；

(7) 还要修改 kernel/proc.c 的 fork()源码，以让被跟踪进程的子孙进程也能够被追踪；

(8) 最后，kernel/syscall.c 的 syscall()函数也要得到修改，以输出相应的跟踪信息（进程号，系统调用名和返回值）；

(9) 这里还要加一个字符数组，用以对相关系统调用名的打印输出。

2.3 名为 sysinfo 的新系统调用

此系统调用的加入过程和 2.2 中的 trace()大致相同，但是执行逻辑不一样。因为有了 2.2 中的开发经验，请大家仔细阅读“官方实验指南”的相关部分自行解题，这里不再赘述。下面会给出参考答案链接给大家在必要的时候使用。

2.4 参考答案

trace

sysinfo

2.5 提交实验报告

请大家提交 2.2，2.3 中加入两个系统调用的详细实验流程报告（文档为 PDF 格式，千万别直接提交 doc 或者 doc 的可编辑格式），并在文档的末尾书写一定的实验心得。