M3: 系统调用Profiler (sperf)

截止日期

Soft Deadline: 4月28日23:59:59

收到的作业

引子：系统调用?

我们已经知道，从应用角度看，操作系统就是API (系统调用)的提供者。是否好奇**一个真正的程序在运行时，到底怎样和操作系统交互**?

操作系统里提供了足够的工具满足你的好奇心——那就是踪迹工具trace。我们可以使用 `strace` 追踪程序执行时的系统调用；用 `ltrace` 追踪程序执行时的库函数调用。这些工具都是诊断问题的神器。基于这些工具，我们甚至可以实现更有趣的hacking工具——在这个实验中，你将借助strace实现一个程序的性能诊断工具，它能够帮助你找到程序运行时耗时最多的那些系统调用。

听起来就很棒对吧，而且这一点都不困难。你只需要学会用进程管理API (fork-execve-wait)和管道(pipe)就可以了！

背景

大家在做OJ题的时候都有Time Limit Exceeded的不良体验。很多时候是因为你的算法实现错误、复杂度分析错误、死循环等导致的，但也的确有那么一些时候，是因为程序稍稍慢了一点，再给我+1s的时间就能跑出来了。

这时候，查看你的程序**在什么地方花了最多的时间**就是一件非常必要的事情，因为导致你超时的也许就是1-2个循环。不出所料，有各种各样的profiler (https://en.wikipedia.org/wiki/Profiling_(computer_programming))分析程序执行的信息。

在这个实验中，我们实现一个命令行工具，它能够分析程序运行时，各个系统调用的耗时，一举两得。更重要的是，这个实验中你能获得hacking的乐趣——在操作系统上编程原来是一件很酷的事情！

实验描述

获取实验代码与提交

本学期的所有代码(minilab, OSlab)都在同一个目录中完成。请参考代码获取与提交 (OS2019_Code)。

在你原先的os-workbench基础上，执行

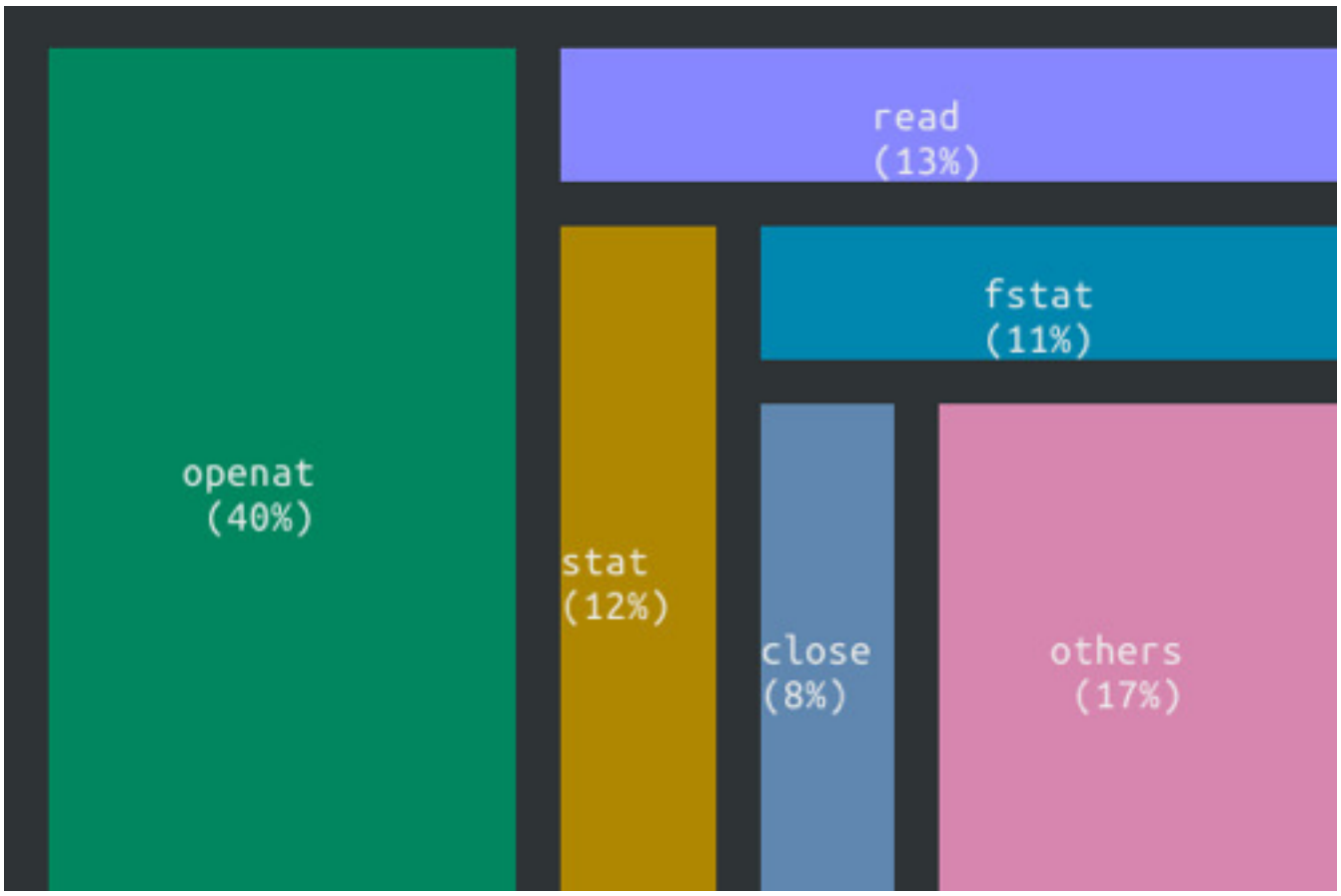
```
git pull origin M3
```

将本实验的框架代码下载到本地。在 `sperf/` 下编译能得到二进制文件 `sperf-32` 和 `sperf-64` 。

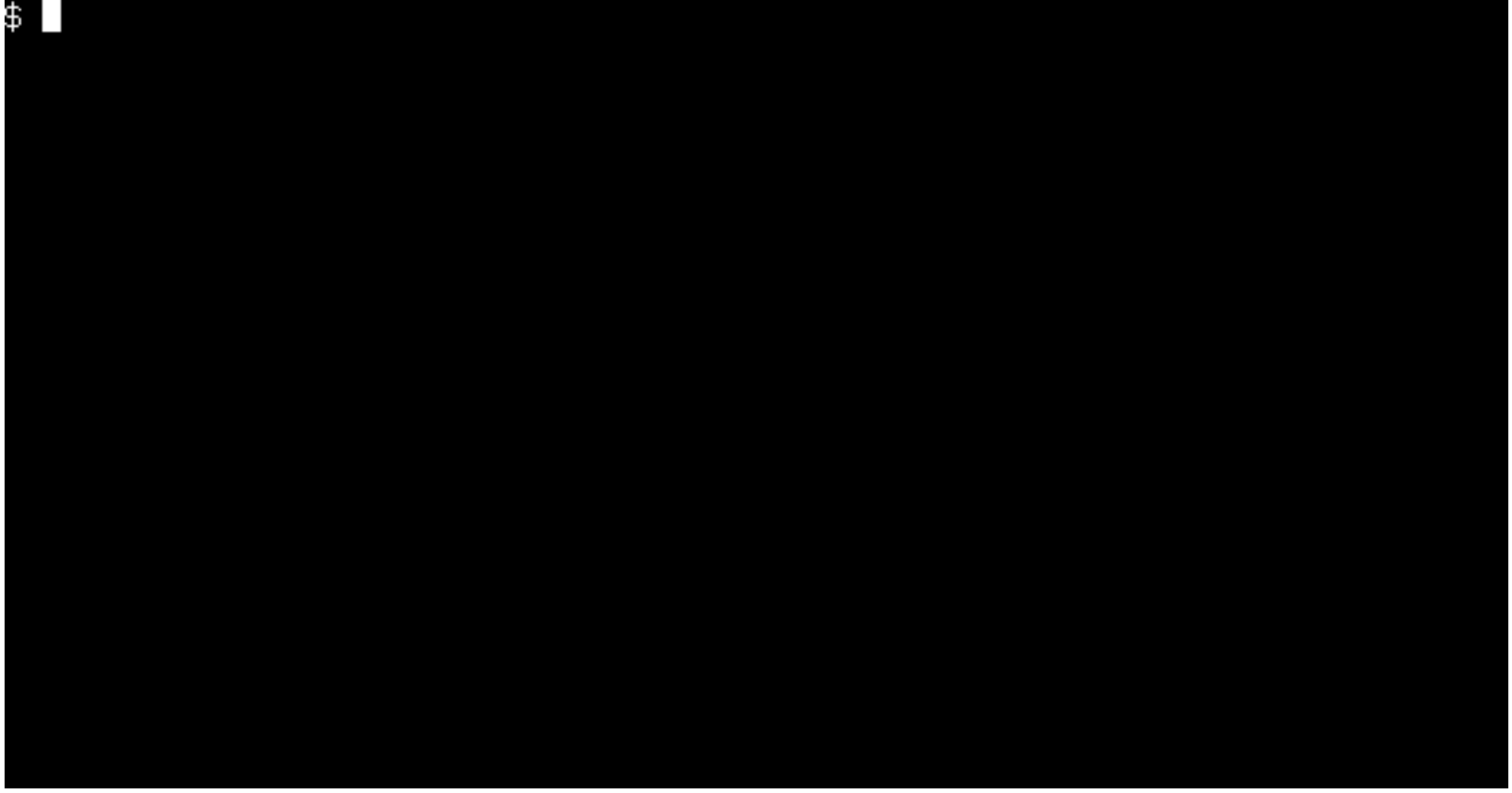
实现命令行工具 `sperf` ：

sperf COMMAND [ARG]...

它执行COMMAND命令(为COMMAND传入ARG参数。例如 `sperf find /` 会统计 `find /` 的系统调用时间。假设 `COMMAND` 是单进程的，无需处理多进程的情况)，在COMMAND运行时，统计它执行的每个系统调用所占的时间，并且把这些系统调用时间的相对比例(处以总系统调用的时间)以用户友好的方式展示在终端上。下面这张图就是在字符终端里绘制的系统调用统计图(系统调用所花时间与所占面积成正比)，它统计了执行 `python -c "print()"` 执行一小段Python代码时，在各个系统调用上花费的时间。很酷对吧?



在运行时，还可以实时显示一段时间内的系统调用时间统计:



RTFM (`man 2 brk`)之后，你立即知道这段Python程序运行，在操作系统中最耗时的部分是内存管理。你的程序不必严格与参考程序一致，它甚至可以只是最简单地打印每个系统调用的时间，甚至不需要任何好看的“界面”，例如：

```
open: 36%
read: 14%
...
```

实验指南

在pstree中，我们已经学习了如何用API (库函数和系统调用)与操作系统交互。在这个实验中，我们需要用 `fork` 创建子进程，在子进程中调用 `strace` ，并将父子进程用 `pipe` (匿名管道)连接起来，在父进程中读取 `strace` 的统计信息并显示。

> **给不想看提示的同学**
>
> 上面这句话给出了足够的信息，去网上搜索吧！

获得各个系统调用的时间

这是整个实验里最困难的部分——作为操作系统新手，我们连系统调用是什么都还不太明白，却立即让我们得到所有系统调用的序列？这时候不妨求助一些外部工具吧。 `strace` 命令能够帮助我们统计进程的系统调用。执行 `strace cmd arg1 arg2 ...` 能统计执行 `cmd arg1 arg2 ...` 的系统调用。例如以下是 `strace ls` 的输出(截取了部分)：

```
execve("/bin/ls", ["ls"], [/* 46 vars */]) = 0
brk(NULL)                                  = 0×8f21000
access("/etc/ld.so.nohwcap", F_OK)         = -1 ENOENT
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1,
access("/etc/ld.so.preload", R_OK)         = -1 ENOENT
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
...
```

可以看到， ls 从 execve 系统调用开始执行。 strace 命令中有一个选项能打印出系统调用所花的时间。所以从本质上讲，我们只需要**解析strace的输出**就行。

把玩 strace

想知道进程是如何和操作系统交互的？ strace 会告诉你的。你能猜到每个系统调用都是做什么的吗？ strace 是很常见的性能诊断工具——如果你看到某个I/O操作花费了过多的时间，你也许很快就能找到性能问题的原因。

系统调用时间统计

实现实验要求的功能分成以下步骤：

1. 运行程序，解析参数。
2. 使用 fork 创建一个子进程，然后分别：
    1. 子进程使用 execve 调用 strace cmd arg1 arg2 ... 。
    2. 把 strace 的输出(strace 输出到了哪里？如何让 strace 输出系统调用的时间？)连接到父进程的输入(例如stdin，当然其他任何一个文件描述符也行)。
3. 子进程已经不再受控制了， strace 会不断输出系统调用的统计。
4. 父进程会不断读取 strace 的输出、解析，并把统计信息实时反映在屏幕上(例如，像例子里那样)。

做些必要的调研再往下看

man strace 阅读一下手册的DESCRIPTION部分，就能得到上面问题的答案。此外，如果你对这几个步骤中的某些感到疑惑，例如“fork-execve”，参考书和网上的资料都很充足。

解析 strace 的输出

strace 的输出大致就和一个C函数调用类似，从一行字符串里提取出我们想要的信息，不过是个简单的OJ题，就是从下面这个字符串里提取出0.000011。

```
mmap2(0×b76d1000, 10780, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MA
```

解析有很多种办法，最方便的当然是用正则表达式啦！有兴趣的同学可以试试 `regex.h` 。

创建进程和管道

使用 `pipe` 系统调用可以创建管道。 `pipe` 系统调用：

```
int pipe(int fildes[2]);
```

它会返回两个**文件描述符**，一个只能读出，另一个只能写入，分别对应了读端和写端——顾名思义，向写端写入数据，读入端就能读到了。所有的文件描述符都会在 `fork` 的时候被继承，所以父进程创建一个管道，子进程也得到了这两个文件描述符：

```
if (pipe(fildes) == 0) {
  // 出错处理
}
pid = fork();
if (pid == 0) {
  // 子进程，执行strace命令
  execve( ... );
  // 不应该执行此处代码，否则execve失败，出错处理
} else {
  // 父进程，读取strace输出并统计
}
```

这部分是比较难理解的代码，因为在 `fork` 以后，同时有两个“一模一样”的程序在执行了。花一点时间阅读参考书和网上的资料，慢慢就会习惯的。

把 `strace` 的输出重定向到管道

在执行了 `execve` 成功后， `strace` 会不断地把输出写到某个编号固定的文件描述符里。所以想把子进程的输出连接到父进程的输入，我们需要执行的操作实际是给定两个文件描述符 `fd` 和 `fildes[x]` (管道的写口)，我们希望把 `fildes[x]` “覆盖”到 `fd` 上。这通过 `dup2` 系统调用就能实现了。

> **思考题：为什么需要 `dup2` ？**
>
> 看起来 `close` 和 `dup` 就能实现 `dup2` 的功能了。为什么还需要它呢？手册里有一些解释，不过稍有些含糊，你能把这个问题想清楚吗？在这个例子里， `dup2` 试图避免的Race Conditions是因为进程里的多个线程导致的。但Race Condition带来的麻烦不止如此，有兴趣的同学可以阅读一篇有趣的论文 (http://ieeexplore.ieee.org/document/5207635/)。

这样在父进程中读取管道，就能得到 strace 的输出了。如果你还不太明白，就试着读一读参考书，或者看一些网上关于父子进程管道通信的文章。

统计系统调用时间信息

解析 strace 的输出，你很快就能得到一张表，知道每个系统调用所花的总时间。把这张表打印出来也很容易，但怎么绘制到终端上？使用ANSI Escape Code (https://en.wikipedia.org/wiki/ANSI_escape_code)就行啦。只要把特定的字符串打印出来，就能完成颜色修改、清屏、移动光标等功能。

那示例的图片是根据什么画出来的呢？我们尽可能让每个系统调用的面积正比于它用时的百分比，就是这个效果了。也许你已经猜到是怎么绘制出来的了。搞不定也没关系，只要打印出比例就好啦——然后隔一段时间用ANSI Escape Code清一下屏幕，就能动态查看程序执行时的系统调用时间占比了。

提示

你会发现一些比较难受的情况，例如 strace 的输出可能和程序的输出混在一起：

```
write(2, "Hello World\n", 12Hello World
)           = 12 <0.000046>
```

可能会对你解析 strace 的输出产生影响。这时候 fork - execve 组合的力量就显示出来的：在执行 execve(STRACE_BIN, ...) 之前，我们可以把程序的输出重定向到 /dev/null 。还记得这个文件吗？

另外，注意到引号里的字符串也可能对你产生影响。如果是这样，你就需要再次读一下 strace 的手册了。

拓展

不那么无聊的终端

刚开始学习编程的时候，都对着OJ，黑底白字的终端，对着Wrong Answer调代码，相当崩溃。但转念一想，在学会使用Escape Code (和一些其他API)控制终端以后，你只需要两个额外的API，就能实现任何有趣的东西了：

1. 准确的定时器；
2. 能够即刻捕获终端按键的API。

能“绘图”、能输入、能定时，各种游戏和小工具(比如输入函数$y = f(x)$，在终端上绘制它的曲线)，就都能实现啦。当然，这件事肯定早就有人做了，比如著名的NetHack (http://www.nethack.org)，以及如果想读一点短的代码，可以看看来自全球最大同性交友网站的终端2048游戏 (https://raw.githubusercontent.com/mevdschee/2048.c/master/2048.c)。

Profiler

> In software engineering (https://en.wikipedia.org/wiki/Software_engineering), **profiling** ("program profiling", "software profiling") is a form of dynamic program analysis (https://en.wikipedia.org/wiki/Dynamic_program_analysis) that measures, for example, the space (memory) or time complexity of a program (https://en.wikipedia.org/wiki/Computational_complexity_theory), the usage of particular instructions (https://en.wikipedia.org/wiki/Instruction_set_simulator), or the frequency and duration of function calls. Most commonly, profiling information serves to aid program optimization (https://en.wikipedia.org/wiki/Program_optimization).

任何成熟的系统都为我们提供了各式各样的Profiler，这样在遇到性能问题的时候就不再需要抓瞎(到处乱改，然后看性能是否提高)了。Linux也给我们提供了profiling的工具 `perf` 。借助各种硬件机制，它的功能比我们实验中实现得强大得多，不过功能要强大得多：

Linux kernel perf能在内核中插入各种各样的观测点(probes)，基于systemtap (http://sourceware.org/systemtap/)的脚本能实现各种定制的profiling (例如观察网络统计报文的情况、磁盘I/O的情况……)。

关于优化

你已经有profiler了，它能非常明确地告诉你程序的时间瓶颈在哪里。因此在编程时，切莫再随意地假定“这么做可能能让效率高一些”，并且为了这么做牺牲正确性、可读性等等。

> *Premature optimization is the root of all evil.* -- Donald Knuth