M5：系统调用 Profiler (sperf)

May 23, 2025

2025

⏰Soft Deadline: 2025 年 6 月 6 日 23:59:59

你需要首先阅读实验须知，其中包含了代码获取方法、提交方法、如何查看提交结果等。在命令行中 `git pull origin M5` 下载框架代码。

1. 背景

“Everything is a state machine.” 能够帮助我们解析 “状态机执行” 的工具，自然对调试、性能诊断等任务是至关重要的。我们在课堂上展示过各类各类分析状态机执行轨迹的工具，包括 gdb (step/stepi)、strace 等。为了加深大家对这些工具的认识，我们使用课堂上学过的进程管理 API，去启动一个这样的工具，并获得它们的输出。这个实验也替代了通常《操作系统》课程中常见的管道 (shell) 作业。

2. 实验描述

📝实验要求：实现系统调用 Profiler

实现一个命令行工具，它能启动另一个程序，并统计该程序中各个系统调用的占用时间。对于较短时间运行的程序，你可以一次性打印出耗时最多的几个系统调用。对于耗时较长的程序，你需要保证每秒大 **打印 10 次** (不要过多) 系统调用的耗时信息。

2.1 总览

```
sperf _COMMAND_ [_ARG_]...
```

2.2 描述

它会运行 COMMAND 命令 (如果 COMMAND 是以 `/` 开头的绝对路径，则直接执行；否则在 `PATH` 环境变量中搜索到第一个存在且可执行的文件)，并为 COMMAND 传入 ARG 参数 (列表)，然后统计命令执行的系统调用所占的时间。例如执行 `sperf find /` 会在系统中执行 `find /`，并且在屏幕上显示出耗时最多的若干系统调用的时间。sperf 假设 `COMMAND` 是单进程、单线程的，无需处理多进程和多线程的情况。

⚠️**只能使用 execve**

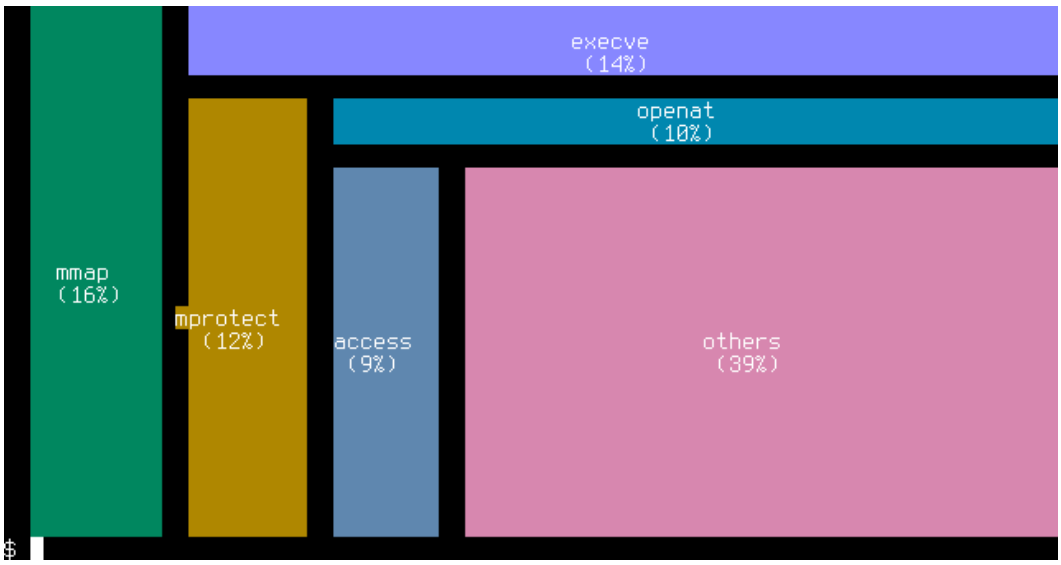
为了强迫大家理解 execve 系统调用，sperf 实现时，**只能使用 execve 系统调用**；使用 glibc 对 execve 的包装 (execl, execlp, execle, execv, execvp, execvpe) 将导致编译错误。在实际编程中，exec 系列函数，例如以 p (path) 结尾的函数会从 PATH 环境变量中搜索可执行文件。

2.3 解释

你可以以任何方式输出系统调用花费的时间，例如你可以每隔一段时间把输出到标准输出：

```
$ ./sperf-64 tree /
Time: 0.1s
lstat (50%)
getdents (20%)
...
====================
Time: 0.2s
lstat (50%)
getdents (20%)
...
```

在我们的参考实现中，通过在字符终端里绘制系统调用图的方式展示系统调用的耗时 (系统调用所花时间与所占面积成正比)。你完全可以把你的 sperf 的输出管道给另一个 “可视化” 工具来生成它——这与本次实验无关。



但无论如何，一个可视化的工具都可以帮我们提供关于系统和程序运行的有效信息，例如：

- 对于 memory intensive 的纯计算型程序，brk 和 munmap 花去了最多的时间；
- 对于文件系统遍历，文件系统元数据访问 lstat 和 getdents 花去了最多的时间。

3. 正确性标准

sperf 对 profiling 结果的输出格式没有特别要求，并假设被 trace 的程序是单进程程序 (Online Judge 测试用例会保证这一点)。如果进程没有发生系统调用，或系统调用耗时很长没有结束 (例如等待输入的 `read` 或大量数据的写入)，你相应的信息也可以不必更新 (即遇到很长的系统调用时，你不需要立即将这个系统调用的时间统计进来，可以等它执行结束后再统计)。

为了 Online Judge 判定方便，我们要求你的程序满足以下输出格式：

> 实验要求：实现系统调用 Profiler
>
> 每次输出耗时 top 5 的系统调用、每个系统调用至多输出一次，包括系统调用的小写名字 (strace 输出的名字)和系统调用耗时比例按照 “`(XX%)`”。使用
>
> ```
> printf("%s (%d%%)\n", syscall_name, ratio);
> ```
>
> 输出即可。在每次统计信息输出完毕后，打印 80 个 `\0` (注意不是 `'0'`，是数值为 0 的字符，它们不会在终端上显示)。我们将以这 80 个 `\0` 作为划分。

输出过于频繁会致 Output Limit Exceeded。保证在被追踪进程不断持续有系统调用发生时，每秒打印大约 10 次即可。此外，如果进程退出，你的 sperf 也应该相应退出。这个行为类似于 strace 和 perf。

> ⚠️及时清空 stdout 的缓冲区
>
> 不同于你的本地测试 (stdout 为终端)，Online Judge 在评测时会把 stdout 重定向到文件 (或管道)，因而 libc 会进入 fully buffered mode (就像我们课堂上讲 fork 在 `./a.out` 和 `./a.out | wc -l` 看到不同行数的例子那样)。因此请确保你在每一轮输出后使用 fflush——否则 timeout 被强行终止时，我们可能看不到你的输出。

4. 实验指南

4.1 Trace 工具

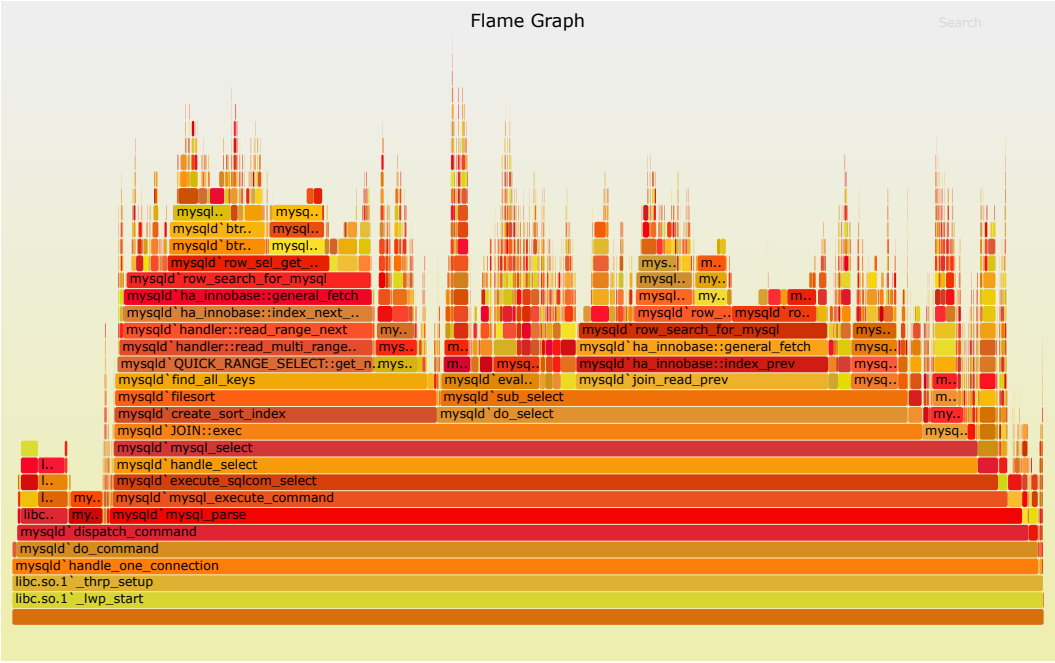
操作系统里提供了足够的工具满足你 "检查状态机执行" 的需求——那就是追踪工具 trace。例如，使用 strace 可以追踪程序执行时的系统调用。

> ☕实现 strace
>
> 如果我们希望在 Linux 上实现 strace，有哪些可能的途径？AI 给了我们一些方案，包括 ptrace 系统调用、Linux Audit、eBPF、SystemTap 和 LTTng。你可以根据自己的兴趣、掌握程度来选择性地了解这些技术。例如，如果你对课程内容感到吃力，不妨优先完成实验；但如果你对编程已经有很不错的自信，就可以试一试其他的机制。

Trace 工具是系统编程时非常常用的问题诊断工具。根据调试理论，trace 在一定程度上反应了程序执行的流程，因此可以帮助你缩小 bug 所在的范围 (回顾调试的目标是找到第一个发生 error 的程序状态)；此外，trace 中额外的信息还能帮助诊断性能等问题 (例如你的程序是否在某个系统调用上花去了大量的时间)。除了我们熟知的 strace, ltrace，其实 trace 广泛地存在于计算机系统的各个层面上：

- 操作系统内核 (例如 Linux) 提供了多种 trace 机制，有兴趣的同学可以参考 ftrace；
- 浏览器的开发者模式 (F12 控制台，想必大家已经用它查看过课程网站的 slides 是如何实现的了) 中，能够 trace 网络等请求——别小看它，这也是一种 trace！而且它对调试网站的功能性问题和性能问题都是至关重要的；
- 通过 Java Virtual Machine Tool Interface (JVMTI) 在 Java 虚拟机中创建 agent，收集程序执行的信息。

在 trace 的基础上，我们实现程序的**性能诊断** 工具 (虽然 strace 的设计意图本身并不主要是为了诊断性能问题)，它能够帮助你找到程序运行时耗时最多的那些系统调用，从而找到程序的性能瓶颈。Profiler 能够提供系统性能分析的报告——而且这件事一点也不难，我们只要分析 trace 的输出就可以了。使用适当的工具，甚至可以根据 trace 的结果输出一个可交互的 Flame Graph



4.2 获得系统调用序列

我们知道 (并且每年考试都会考)，strace 可以获得进程系统调用的序列：

```
$ strace ls
execve("/bin/ls", ["ls"], 0x7ffd3a99eb80 /* 23 vars */)  = 0
brk(NULL)                                               = 0x55c642aa9000
access("/etc/ld.so.nohwcap", F_OK)                      = -1 ENOENT
access("/etc/ld.so.preload", R_OK)                      = -1 ENOENT
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=93852, ...})    = 0
...
```

而系统调用是一个 “状态机状态” 与操作系统交互的**唯一** 方式 (这个说法并不严格；还可以通过共享内存实现与操作系统内核的通信)。strace 从 execve “重置状态机” 的系统调用开始，列出进程和操作系统的一切交互。所以从本质上讲，我们只需要**解析 strace 的输出** 就能完成这个实验。

> ☕阅读系统调用序列
>
> 这个实验的一个有意的 “设计” 是强迫大家阅读 strace 输出的系统调用序列，进而解析它们。这样，大家就会被迫去看一看真实运行在操作系统上的进程是怎样调用操作系统 API 的。如果你对任何系统调用感到困惑，你都可以直接询问 AI。

当然了，我们不仅希望得到系统调用的序列，还需要有系统调用的执行时间才能完成这个实验。strace 的输出中并没有，嗯，那你应该想到计算机系统中的 “公理”：

> 💡计算机系统公理
>
> 如果你有一个合理的需求，就一定有工具可以满足你。

所以你会猜测 strace 也提供了这个功能。你可以搜索网络、阅读手册，或者直接问一问 AI “如何获得一个进程每个系统调用的时间？”。人工智能给你的回答会有惊喜 (例如，它会给出 Windows 和 macOS 应该怎么获得系统调用的时间)。在建立了 strace 的基本概念后，我们依然推荐大家遍历式地阅读 strace 的手册，获得对这个命令行工具更全面的理解。

4.3 实现 sperf

所以，现在我们就可以基于 strace 设计一个完成实验的路线图了。在这里，我们把重要的步骤为大家分解好：

1. 使用 `fork` 创建一个新的进程；
2. 子进程使用 execve 调用 `strace COMMAND ARG...`，启动一份 strace；注意在 execve 成功返回以后，子进程已经不再受控制了，strace 会不断输出系统调用的 trace，直到程序结束。程序不结束 strace 也不会结束；
3. 父进程想办法不断读取 strace 的输出，直到 strace 程序结束，读取到输出后，如果距离上次输出已经超过 100ms，就把统计信息打印到屏幕上。

你遇到的第一个实际问题就是使用 execve 系统调用。

```
int execve(const char *filename, char *const argv[], char *const envp[]);
```

首先，filename 是相对于进程的当前目录 (current working directory) 而言的，或者是一个绝对路径。因此，虽然你在终端里 `yes` 是没问题的，但是如果你

```
execve("yes", NULL, NULL);
perror("exec");
```

立即就会得到 “exec: No such file or directory”，除非你使用 `/usr/bin/yes`。这个问题可以通过 execvpe 解决，不过我们的实验是禁用的，所以你得自己想办法。

此外，你还需要传入正确的 `argv` 和 `envp`。如果你只是调用 yes，给 `envp` 传入 `NULL` 不算是什么大问题。但很多程序的运行都是依赖于正确的环境变量的——一个例子就是 strace。如果在 execve strace 的时候没有传入正确的环境变量，它也没法执行命令：

```
$ PATH="" /usr/bin/strace ls
/usr/bin/strace: Can't stat 'ls': No such file or directory
```

用以下代码 (不出意外) 应该能正确执行 strace：

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
    char *exec_argv[] = { "strace", "ls", NULL, };
    char *exec_envp[] = { "PATH=/bin", NULL, };
    execve("strace",          exec_argv, exec_envp);
    execve("/bin/strace",     exec_argv, exec_envp);
    execve("/usr/bin/strace", exec_argv, exec_envp);
    perror(argv[0]);
    exit(EXIT_FAILURE);
}
```

它 “手工” 模拟了 PATH 环境变量的解析——这也解释了为什么我们有一系列的 exec 函数：execl, execlp, execle, execv, execvp, execvpe。

在课堂上 (和 xv6 shell) 中，我们介绍了 pipe 系统调用，创建匿名管道，并且在 gdb 中调试了 Shell 是如何实现重定向的。我们摘抄 xv6 shell 管道命令的实现：

```c
case PIPE:
    pcmd = (struct pipecmd *)cmd;
    assert(syscall(SYS_pipe, p) >= 0);
    if (syscall(SYS_fork) == 0) {
        syscall(SYS_close, 1);
        syscall(SYS_dup, p[1]);
        syscall(SYS_close, p[0]);
        syscall(SYS_close, p[1]);
        runcmd(pcmd->left);
    }
    if (syscall(SYS_fork) == 0) {
        syscall(SYS_close, 0);
        syscall(SYS_dup, p[0]);
        syscall(SYS_close, p[0]);
        syscall(SYS_close, p[1]);
        runcmd(pcmd->right);
    }
    syscall(SYS_close, p[0]);
    syscall(SYS_close, p[1]);
    syscall(SYS_wait4, -1, 0, 0, 0);
    syscall(SYS_wait4, -1, 0, 0, 0);
    break;
```

> ⚠️请大家不要复制粘贴上面的代码
>
> 上面的代码给大家一个 “要做什么” 的提示；而如果你自己亲手从零编写，你就会被迫画出状态机，模拟多个进程之间交互的状态——这是对大家理解多进程、管道非常必要的。

strace 的输出很像是一行 C 函数调用。如果使用了正确的参数调用 strace，我们的问题就变成了从一行字符串里提取特定位置的数字字符串，例如 “`<0.000011>`”。

```
mmap2(0xb76d1000, 10780, PROT_READ|PROT_WRITE, ...) = 0xb76d1000 <0.000011>
```

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` 。

☕写出正确的 sperf

如果你想得多一些，可能会发现一些额外的麻烦。例如，strace 默认会把 trace 输出到 stderr，这个行为没啥问题。但如果 strace 追踪的程序也输出到 stderr，不就 “破坏” 了 strace 的输出了吗。而且，即便解决了上面的问题 (比如把程序的 stderr 重定向到 `/dev/null` 丢弃)，程序输出的字符串也可能对 parsing 带来不好的影响：

```
$ strace -T echo -n '", 1) = 100 <99999.9>' > /dev/null
ioctl(1, TCGETS, 0x7fff80723570)        = -1 ENOTTY <0.000100>
write(1, "\", 1) = 100 <99999.9>", 21)  = 21 <0.000149>
close(1)                                = 0 <0.000126>
```

是的，一个系统工具需要在**任何时候** 都能正常工作。上面的例子有点 “故意耍赖”，但如果你只试图匹配 `<>` 中的数字，就的确会在一些场景下产生错误。虽然我们并不会想尽一切办法构造极端的测试用例来对付大家，但大家不妨可以深入思考一下 “怎么把程序写对” 这个问题。