

Introduction to Operating Systems

Final Project (`strace`)

Fall 2024

1. Introduction

When a program runs into issues such as a sudden crash or unexpected output, a developer can detect the problem by debugging the source code. Ideally a developer should always have access to the source code to detect the problem, however in practice this is not always the case. Softwares, services or libraries developed by other teams can be delivered as an executable binary file and access to source code is not always provided. Linux provides developers a powerful tool called “`strace`” with the ability to show system calls or library calls, which a program made at a low level. When running into problems, we can use `strace` to roughly guess what the program does and what system calls it fails. It helps in detecting the issues without digging into the details of source code. Please refer to the manual page: <https://man7.org/linux/man-pages/man1/strace.1.html>

2. Goal

Unfortunately, `strace` is not built into the xv6 operating system. Our goal of the project is to make a version of `strace` for xv6.

3. Before you start

All implementation requires some small explanations (2-3 sentences) of your approach. You can explain more if you would like but it should be less than or equal to the introduction of the project prompt. Please check the rubric alongside each task. The project is worth 100 points (10 points for written report)

Screenshots showing the result of each task should be provided in your report. Some requirements can be tricky to test. **Your points will be deducted if screenshots of results are not provided.**

4. Tasks

4.1: Get familiar with Linux `strace` [5 points]

- **4.1.1:** Use `strace` in Linux to learn about `echo` command or any other command of your choice. Show a list of system calls being made, total number of calls, time for running `strace` on a particular command. (screenshots of your run should be enough for this task)
 - **4.1.2:** Pick 4 random system calls (ex: `nmap`, `write`, `open`, etc) of your choice and explain their functionality for the command that you run (ex: `echo`, `ls`, `cd`, etc) in 2-3 sentences
-

4.2: Building `strace` in `xv6`

- **4.2.1: Implement “`strace on`” [10 points]**

When typing “`strace on`” in the terminal, the mode of `strace` is on and therefore the next typed in command will be traced. The system call list will be printed on screen in format `pid (process id), command name, system call name, return value`.

```
$ strace on
$ echo hello
TRACE: pid = 4 | command_name = sh | syscall = trace | return value = 0
TRACE: pid = 4 | command_name = sh | syscall = exec
hTRACE: pid = 4 | command_name = echo | syscall = write | return value = 1
eTRACE: pid = 4 | command_name = echo | syscall = write | return value = 1
lTRACE: pid = 4 | command_name = echo | syscall = write | return value = 1
lTRACE: pid = 4 | command_name = echo | syscall = write | return value = 1
oTRACE: pid = 4 | command_name = echo | syscall = write | return value = 1

TRACE: pid = 4 | command_name = echo | syscall = write | return value = 1
TRACE: pid = 4 | command_name = echo | syscall = exit
$
```

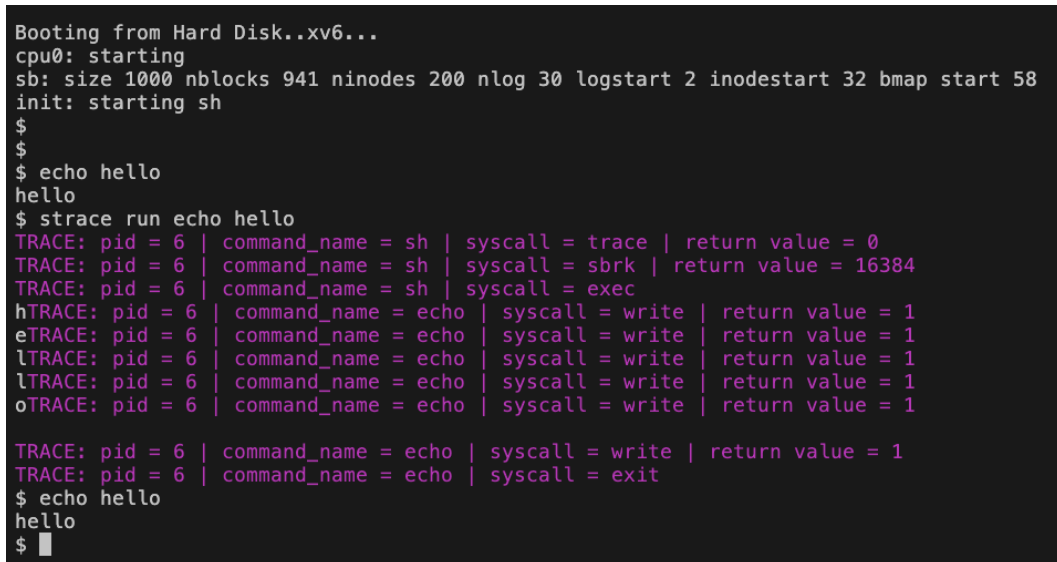
Figure 1: “`strace on`” sample

- **4.2.2: Implement “`strace off`” [10 points]**

When typing “`strace off`” in the terminal, the mode of `strace` is off and therefore the next typed in command won’t be traced and nothing is printed on screen besides the routine result of the commands.

- **4.2.3: Implementing “strace run <command>” [10 points]**

Instead of turning on and off `strace`, we create “`strace run`” to directly start tracing to the current process that executes the command. For example: when typing “`strace run echo hello`” in the terminal, we get the output tracing of `echo hello`.



```
Booting from Hard Disk..xv6...
cpu0: starting
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$
$
$ echo hello
hello
$ strace run echo hello
TRACE: pid = 6 | command_name = sh | syscall = trace | return value = 0
TRACE: pid = 6 | command_name = sh | syscall = sbrk | return value = 16384
TRACE: pid = 6 | command_name = sh | syscall = exec
hTRACE: pid = 6 | command_name = echo | syscall = write | return value = 1
eTRACE: pid = 6 | command_name = echo | syscall = write | return value = 1
lTRACE: pid = 6 | command_name = echo | syscall = write | return value = 1
lTRACE: pid = 6 | command_name = echo | syscall = write | return value = 1
oTRACE: pid = 6 | command_name = echo | syscall = write | return value = 1

TRACE: pid = 6 | command_name = echo | syscall = write | return value = 1
TRACE: pid = 6 | command_name = echo | syscall = exit
$ echo hello
hello
$
```

Figure 2: `strace run` sample

- **4.2.4: Implementing `strace dump`[10 points]**

Implement a kernel memory that will save N number of latest events. This N number can be configurable by using `define` in `XV6`. In other words, it can be hard coded but the way to implement it is to use `#define` to declare a variable called N with a certain value. When the “`strace dump`” command is called, print all events that are saved in kernel memory.

- **4.2.5: Trace child process[10 points]**

Write a program that uses `fork` to spawn child processes. Utilize kernel memory that you have previously implemented to store trace output of all processes. Perform `strace` on this program and print all trace of processes that you have spawned.

- **4.2.6: Extra credits: Formatting more readable output**

Depending on your implementation, you might notice your command result is printed along with your `strace` result which causes tracing to be difficult to read when characters keep mixing up between lines (ex: “`ls`” has a very long list result).

Choose one of the follow approaches to solve this issue:

- Find a way to format your output such that the result of the command is printed first and `strace` result is printed after.
 - Find a way to suppress the command output and leave only `strace` output when `strace` is on.
 - A creative approach of your choice.
-

4.3: Building options for `strace`

• 4.3.1: Option: `-e <system call name>` [5 points]

When option flag `-e` is provided followed by a system call name (ex: `write`), we will print only that system call. If no such system call is made in the command, print nothing.

```
$ strace on
$ echo hello
TRACE: pid = 6 | command_name = sh | syscall = trace | return value = 0
TRACE: pid = 6 | command_name = sh | syscall = sbrk | return value = 16384
TRACE: pid = 6 | command_name = sh | syscall = exec
hTRACE: pid = 6 | command_name = echo | syscall = write | return value = 1
eTRACE: pid = 6 | command_name = echo | syscall = write | return value = 1
lTRACE: pid = 6 | command_name = echo | syscall = write | return value = 1
lTRACE: pid = 6 | command_name = echo | syscall = write | return value = 1
oTRACE: pid = 6 | command_name = echo | syscall = write | return value = 1

TRACE: pid = 6 | command_name = echo | syscall = write | return value = 1
TRACE: pid = 6 | command_name = echo | syscall = exit
$ strace -e write
$ echo hello
hTRACE: pid = 7 | command_name = echo | syscall = write | return value = 1
eTRACE: pid = 7 | command_name = echo | syscall = write | return value = 1
lTRACE: pid = 7 | command_name = echo | syscall = write | return value = 1
lTRACE: pid = 7 | command_name = echo | syscall = write | return value = 1
oTRACE: pid = 7 | command_name = echo | syscall = write | return value = 1

TRACE: pid = 7 | command_name = echo | syscall = write | return value = 1
$ echo hello
TRACE: pid = 8 | command_name = sh | syscall = trace | return value = 0
TRACE: pid = 8 | command_name = sh | syscall = sbrk | return value = 16384
TRACE: pid = 8 | command_name = sh | syscall = exec
hTRACE: pid = 8 | command_name = echo | syscall = write | return value = 1
eTRACE: pid = 8 | command_name = echo | syscall = write | return value = 1
lTRACE: pid = 8 | command_name = echo | syscall = write | return value = 1
lTRACE: pid = 8 | command_name = echo | syscall = write | return value = 1
oTRACE: pid = 8 | command_name = echo | syscall = write | return value = 1

TRACE: pid = 8 | command_name = echo | syscall = write | return value = 1
TRACE: pid = 8 | command_name = echo | syscall = exit
$
```

Figure 3: `strace` with option `-e` sample

Notes: Notice the first `strace` run is similar to regular "echo hello" run in the above figure 1. Next command is "`strace -e write`" which activates tracking of the `write` system call. We run "echo hello" again, observe that only the `write` system call is printed. One more run of "echo hello", and everything is back to normal without tracing only the `write` system call. Similar idea should be done in option `-s` and `-f`.

- **4.3.2: Option: -s [5 points]**

When option flag -s is provided, print only successful system call.

- **4.3.3: Option: -f [5 points]**

When option flag -f is provided, print only failed system call.

NOTE: Option runs only once

When option is called (ex: "strace -e write" follow by "echo hello"), the result of strace with only write system call is printed only once for the following command "echo hello". If typing the next follow command for example "ls", the option -e is turn off and all system call will be printed. In other word, options is used once per command.

- **4.3.4: Extra credits: Combine options**

Implement these 2 commands:

- "strace -s -e <system call name>": print only successful system call name.
- "strace -f -e <system call name>": print only failed system call name.

- **4.3.5: Extra credits: Implement -c options**

Option -c in strace will generate a statistical report of system calls regarding the input command such as duration, total call, failed call. Create a similar report table with using option -c.

```
[0] % strace -c echo hello
hello
```

| % time | seconds | usecs/call | calls | errors | syscall |
|--------|----------|------------|-------|--------|------------|
| 0.00 | 0.000000 | 0 | 1 | | read |
| 0.00 | 0.000000 | 0 | 1 | | write |
| 0.00 | 0.000000 | 0 | 5 | | close |
| 0.00 | 0.000000 | 0 | 4 | | fstat |
| 0.00 | 0.000000 | 0 | 8 | | mmap |
| 0.00 | 0.000000 | 0 | 4 | | mprotect |
| 0.00 | 0.000000 | 0 | 1 | | munmap |
| 0.00 | 0.000000 | 0 | 3 | | brk |
| 0.00 | 0.000000 | 0 | 1 | 1 | access |
| 0.00 | 0.000000 | 0 | 1 | | execve |
| 0.00 | 0.000000 | 0 | 1 | | arch_prctl |
| 0.00 | 0.000000 | 0 | 3 | | openat |
| 100.00 | 0.000000 | 0 | 33 | 1 | total |

Figure 4: strace with option -c sample in Linux

Notes: Even though the sample is in Linux, a similar report table is expected in XV6 for extra credits. The columns required are: calls, errors, syscall and seconds (measure the total time it takes to execute system call). You should try to test with

a command that will take time to execute system call and display result of that command.

4.4: Write output of `strace` to file [5 points]

Find an implementation of choice to write `strace` output to file.

For example: by providing option: `-o <filename>` or editing content of README.

4.5: Application of `strace` [15 points]

Write a small program that produce an unexpected behavior such as race condition, delay output, crash on condition, memory leak or your choice of implementation. Run `strace` on this program.

- Explain your program and its unexpected behavior.
 - (xv6) Does your `strace` implementation provide anything useful that can tell you something about this unexpected behavior?
 - (Linux) Implement same program on Linux (there is slight various of C language between Linux and XV6), does Linux `strace` provide more information and is it better to help you debug this issue?
-

5: What to hand in

- On Brightspace, submit a zip file containing
 - Your xv6 code (complete directory)-> zipped
 - A written report that includes all explanations and screenshots (**Report contains 10 points on rubric**).
 - A README on how to run your program.
 - It should tell how your xv6 zipped folder is different from the original one i-e which files you have changed/modified
 - Please ensure your instructions are clear on how to run what behavior is to be expected
- A text that states your partner or working alone.