CS-GY 6233 Final Project

Student 1: Jennifer (ChungEun) Chae (cec595)
Student 2: Abhinav Dwarkani (ad5658)

Task 3: Small explanations of the implementation approach:

(2-3 Sentences):

- Implementation overview:
  - First, we added a user program called "**strace.c**" where inside main, we take care of each option passed in along with "strace" command. We utilized several per-cpu and per-process level variables that we stored inside the kernel to save different information to implement. We also added a handful of new system calls in order to modify the above information. For storing the N last events, we used a per-cpu variable that is an array of struct that can store each event-related information (pid, system call name, etc.) for later printing.

Task 4: Get familiar with Linux strace

- Part 1) screenshot of running "$strace echo hello" on Anubis.

```
anubis@anubis-ide ‹ master@6c4b2dc › : ~/final-project-371525eb-cec595
[0] % strace echo hello
execve("/bin/echo", ["echo", "hello"], 0x7ffc8cb0e4f8 /* 45 vars */) = 0
brk(NULL)                               = 0x55aa210aa000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=34804, ...}) = 0
mmap(NULL, 34804, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f55d06f0000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0@n\2\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=1839792, ...}) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f55d06ee000
mmap(NULL, 1852680, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f55d0529000
mprotect(0x7f55d054e000, 1662976, PROT_NONE) = 0
mmap(0x7f55d054e000, 1355776, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x25000) = 0x7f55d054e0
00
mmap(0x7f55d0699000, 303104, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x170000) = 0x7f55d0699000
mmap(0x7f55d06e4000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1ba000) = 0x7f55d06e40
00
mmap(0x7f55d06ea000, 13576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f55d06ea000
close(3)                                = 0
arch_prctl(ARCH_SET_FS, 0x7f55d06ef580) = 0
mprotect(0x7f55d06e4000, 12288, PROT_READ) = 0
mprotect(0x55aa2108c000, 4096, PROT_READ) = 0
mprotect(0x7f55d0723000, 4096, PROT_READ) = 0
munmap(0x7f55d06f0000, 34804)           = 0
brk(NULL)                               = 0x55aa210aa000
brk(0x55aa210cb000)                     = 0x55aa210cb000
openat(AT_FDCWD, "/usr/lib/locale/locale-archive", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=3041456, ...}) = 0
mmap(NULL, 3041456, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f55d0242000
close(3)                                = 0
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0), ...}) = 0
write(1, "hello\n", 6hello
)                  = 6
close(1)                                = 0
close(2)                                = 0
exit_group(0)                           = ?
+++ exited with 0 +++
anubis@anubis-ide ‹ master@6c4b2dc › : ~/final-project-371525eb-cec595
[0] %
```

- On Anubis, we used strace on "echo hello" and got the above results.

part 2) Pick 4 random system call (ex: nmap, write, open, etc) of your choice and explain its functionality for one command that you run (ex: echo, ls, cd, etc) in 2-3 sentences.

- Using the above example, we have “strace echo hello”. So we are running the linux command of “echo” with “hello” as an argument.
- System call #1 -> “execve(“/bin/echo”, [“echo”, “hello”], …)
    - execve() system call runs the function “echo” with argument “hello” pointed to by the filename “/bin/echo”.
- System call #2 -> “openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3”
    - openat() system call opens related libraries.
- Sytem call #3 -> “fstat(3, {st_mode=S_IFREG|0644, st_size=34804, ...}) = 0”
    - fstat() system call now gets the information about the open file from the file descriptor.
- System call #4 -> “write(1, "hello\n", 6hello”
    - Now “write” system call writes “hello” so that it displays that text on the command line.

2. Building strace in xv6

- Command: strace on
    - Explanation: I implemented a system call called “**setTraceState**” to set the **cpu->isTraceOn** value to 1 (meaning ON) since we need to keep tracing after the command is entered.

```
init: starting sh
$ strace on
$ echo hello
Trace: pid = 4 | command_name = sh | syscall = sbrk | return_value = 16384
Trace: pid = 4 | command_name = echo | syscall = exec | return_value = 0
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
```

- Command: strace off
    - Explanation: This time, we call the system call, “**setTraceState**” to set the **cpu->isTraceOn** value to 0 (meaning OFF).

```
init: starting sh
$ strace on
$ echo hello
Trace: pid = 4 | command_name = sh | syscall = sbrk | return_value = 16384
Trace: pid = 4 | command_name = echo | syscall = exec | return_value = 0
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
$ strace off
Trace: pid = 5 | command_name = sh | syscall = sbrk | return_value = 16384
Trace: pid = 5 | command_name = strace | syscall = exec | return_value = 0
$ echo hello
hello
$
```

- Command: strace run <command>
    - Explanation: we added a process-level variable "**isProcTraceOn**" and added a new system call to manipulate its value. We also added a per-process variable called **isProcTracing** to specify which process we are going to trace because we are doing a one-time tracing and directly point tracing to the current process. This works with the per-process variable because I added code to clear this property by resetting it to 0 in **allocproc()** function in **proc.c** file so that it won't keep tracing.

```
$ echo hello
hello
$ strace run echo hello
Trace: pid = 8 | command_name = echo | syscall = exec | return_value = 0
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
$ echo hello
hello
$
```

- Command: strace dump
    - Explanation: we added a few more variables to per-cpu state in proc.h file. We added an array of struct called **record** and initialized it to size of N. This struct has 3 fields, pid, return_value, and syscall for each tracing event. We also added an array of char array called "**command_name_arr**" to store the command_name for each tracing event. When a system call event happens, I added the above four information to each respective variable. Then, I updated the **cpu-> write_idx** to keep track of when to overwrite, and where to read from. Also, we added the **traceDump** system call where it loops N times over

the **record** and **command_name_arr** to dump the N last recorded events. We also added N to be a compile time constant defined as 16 in our code.

```
$ strace dump
==================== strace dump results ====================
PID = 4 | command = wc | syscall = write | return_value = 1
PID = 4 | command = wc | syscall = write | return_value = 1
PID = 4 | command = wc | syscall = write | return_value = 1
PID = 4 | command = wc | syscall = write | return_value = 1
PID = 4 | command = wc | syscall = write | return_value = 1
PID = 4 | command = wc | syscall = close | return_value = 0
PID = 5 | command = sh | syscall = sbrk | return_value = 16384
PID = 5 | command = echo | syscall = exec | return_value = 0
PID = 5 | command = echo | syscall = write | return_value = 1
PID = 5 | command = echo | syscall = write | return_value = 1
PID = 5 | command = echo | syscall = write | return_value = 1
PID = 5 | command = echo | syscall = write | return_value = 1
PID = 5 | command = echo | syscall = write | return_value = 1
PID = 5 | command = echo | syscall = write | return_value = 1
PID = 6 | command = sh | syscall = sbrk | return_value = 16384
PID = 6 | command = strace | syscall = exec | return_value = 0
$
```

- Trace child process
  - Explanation: We created a program called “**testTrace.c**” and added a small program that uses fork() and trace the processes shown below:

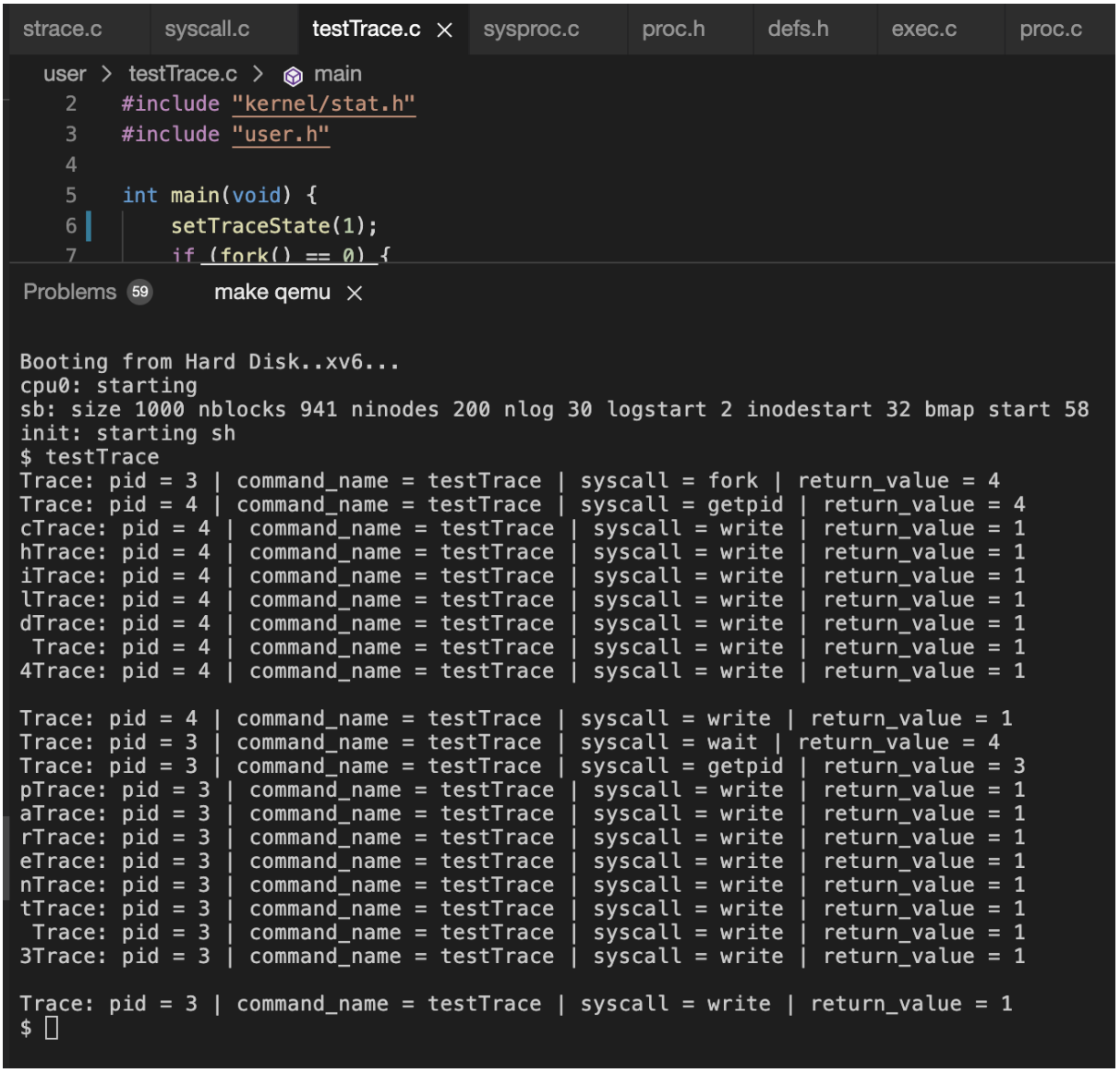
strace.c | syscall.c | testTrace.c | sysproc.c | proc.h

user > testTrace.c > main

```c
#include "kernel/stat.h"
#include "user.h"

int main(void) {
    setTraceState(1);
    if (fork() == 0) {
        printf(1, "child %d\n", getpid());
    } else {
        wait();
        printf(1, "parent %d\n", getpid());
    }
    exit();
}

```



3. Building option for strace

- Option: -e <system call name>
  - Explanation: We added a new system call **setTargetSuccess** to manipulate the variable **proc->target_success_fail**, which is a variable we added to per-process state. All the options only run once so we needed to use the process level state variable and reset that value in **allocproc()**. Here, we set **proc->target_success_fail** to 1 to instruct to only trace the system call name that was passed in along with strace -e. Since the argument (system call name) is string type, we added a function **getNumberSys()** inside strace.c to convert the system call name to integer. We also included the `"kernel/syscall.h"` file to refer to the integer assigned to each system call and then stored the resulting integer into **cpu->trace_syscall** variable for comparison.

```
init: starting sh
$ strace run echo hello
Trace: pid = 3 | command_name = echo | syscall = exec | return_value = 0
hTrace: pid = 3 | command_name = echo | syscall = write | return_value = 1
eTrace: pid = 3 | command_name = echo | syscall = write | return_value = 1
lTrace: pid = 3 | command_name = echo | syscall = write | return_value = 1
lTrace: pid = 3 | command_name = echo | syscall = write | return_value = 1
oTrace: pid = 3 | command_name = echo | syscall = write | return_value = 1

Trace: pid = 3 | command_name = echo | syscall = write | return_value = 1
$ strace -e write
$ echo hello
hTrace: pid = 5 | command_name = echo | syscall = write | return_value = 1
eTrace: pid = 5 | command_name = echo | syscall = write | return_value = 1
lTrace: pid = 5 | command_name = echo | syscall = write | return_value = 1
lTrace: pid = 5 | command_name = echo | syscall = write | return_value = 1
oTrace: pid = 5 | command_name = echo | syscall = write | return_value = 1

Trace: pid = 5 | command_name = echo | syscall = write | return_value = 1
$ echo hello
hello
$
```

- STRACE Option: -s

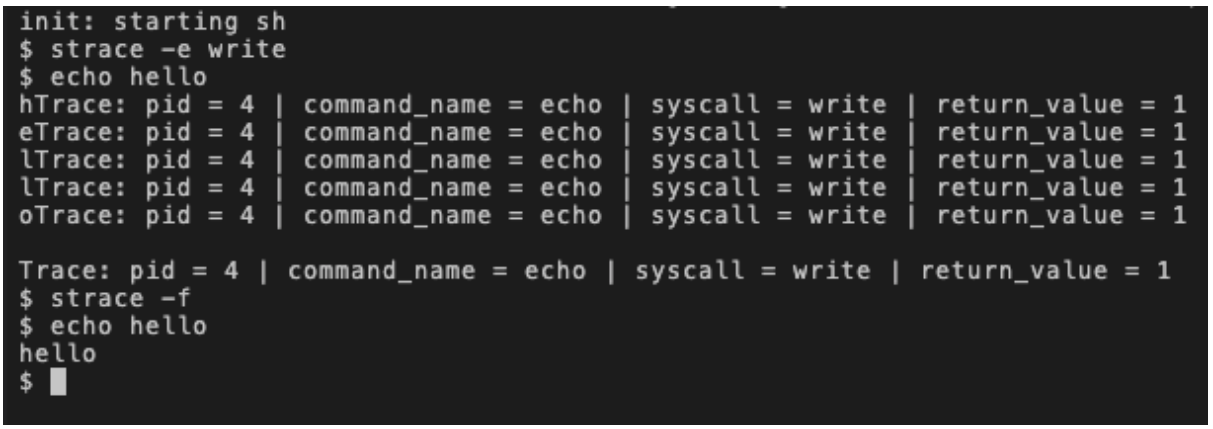
- Explanation: when a system call succeeds, it will return a value that is equal to or greater than 0. And then in syscall.c file syscall() function, we filtered out system call events by their return value. We made sure that it is a one-time tracing through maintaining a **cpu->nextProcPid** which contains the next pid to track.

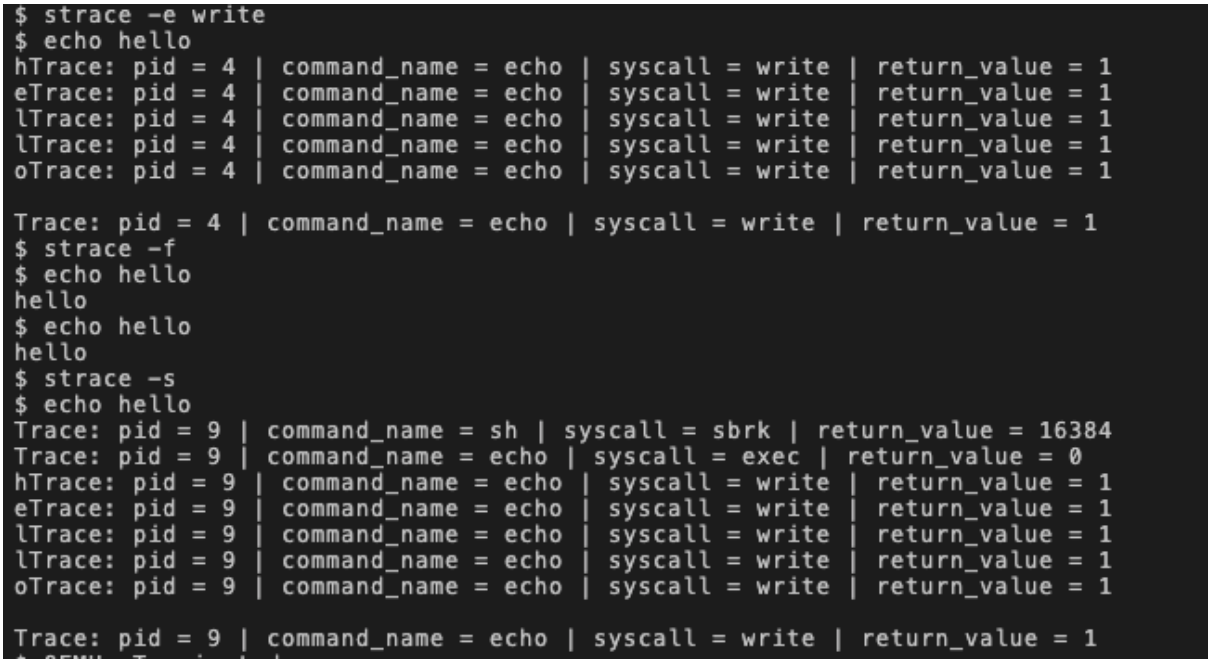
```
init: starting sh
$ strace -s
$ echo hello
Trace: pid = 4 | command_name = sh | syscall = sbrk | return_value = 16384
Trace: pid = 4 | command_name = echo | syscall = exec | return_value = 0
hTrace: pid = 4 | command_name = echo | syscall = write | return_value = 1
eTrace: pid = 4 | command_name = 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
$ echo hello
hello
$
```

- STRACE Option: -f

- Explanation: when system calls fails for whatever reason, it will return a value that is less than 0. And then in syscall.c file syscall() function, we filtered out system call events by their return value.





note: instructions on how to run can be found in README.md file.

4. Write output of strace to file