System Call Tracer (strace-like) – Logging system calls of processes.

Objective:

To develop a system call tracing mechanism in the xv6 operating system that logs all system calls made by processes, including the syscall name, arguments, and return values. This allows monitoring and debugging of process behavior at the syscall level.

---

1. Introduction

This project modifies the xv6 kernel to add a syscall tracer, similar in functionality to the Unix tool strace. It involves kernel-level changes to track and log syscalls, adding a new system call to enable/disable tracing per process, and a user-level program to control the tracing

---

2. Detailed Changes Made

2.1 Modify proc.h

code

```
57
58   //this flag will track if syscall tracing is enabled for the process, the
   default value is 0 (there is no tracing)
59   int traced;            // flag: 1 if syscall tracing is enabled, else 0
60 };
```

Why we add this:
We add the traced flag inside the proc structure because syscall tracing is a per-process feature. Storing this flag in the process control block allows the kernel to efficiently check whether tracing is enabled for the currently running process.

2.2 Define New System Call trace

In syscall.h:

code

```
22 #define SYS_close  21
23 #define SYS_trace  22  //adding a new syscall
```

Why we add this:
The trace syscall allows user programs to enable or disable syscall tracing on a per-process basis. Defining it in syscall.h registers it within the syscall numbering system, enabling the kernel to identify and dispatch this new syscall correctly.

In defs.h:

System Call Tracer (strace-like) – Logging system calls of processes.

code

```
188
189 int             sys_trace(void);
190
```

Why we add this:

Declaring the function prototype in defs.h allows the kernel to reference the handler function during syscall dispatch.

2.3 update sh.c

code

```
17 int tracing = 0;
18
19 char trace_cmd[] = "trace\n";
20 char untrace_cmd[] = "untrace\n";
21
22 //used to compare strings
23 int streq(char *a, char *b) {
24    while(1) {
25      if(*a != *b) {
26        return 0;
27      }
28      // apparently commands end with '\n'
29      if(*a =='\n') return 1;
30      a++;
31      b++;
32    }
33 }
```

Why we update this:

- tracing is a global integer flag that indicates whether syscall tracing is enabled (1) or disabled (0). Initially, it is set to 0 (tracing off).
- trace_cmd and untrace_cmd are string constants that represent the commands entered by the user to enable or disable tracing. They include the newline character \n because input lines from the shell typically end with a newline.
- streq() is a helper function that compares two strings character-by-character, returning 1 if they match exactly (including the newline at the end), or 0 if they differ. This function is used to detect when the user input matches the trace or untrace commands.

System Call Tracer (strace-like) – Logging system calls of processes.

```
97    if (tracing) trace(T_TRACE | T_ONFORK); //set tracing on
```

Why we update this:

- This line checks if the tracing flag is enabled (i.e., tracing == 1).
- If yes, it calls the trace() function with flags T_TRACE | T_ONFORK.
- This instructs the kernel (or tracing mechanism) to enable tracing on the current process **and** to enable tracing on any child processes created by fork() (via T_ONFORK).
- Essentially, this ensures that syscall tracing is active not just on the current shell or process but also propagates to processes spawned from it.

```
187     if(streq(buf, trace_cmd)){
188       tracing = 1;
189       continue;
190     }
191     if(streq(buf, untrace_cmd)){
192       tracing = 0;
193       continue;
194     }
195     if(fork1() == 0) {
196       runcmd(parsecmd(buf));
197     }
198     wait();
199   }
200   exit();
201 }
```

Why we update this:

- This code snippet checks if the user input stored in buf matches either "trace\n" or "untrace\n".
- If it matches "trace\n", the tracing flag is set to 1, enabling syscall tracing for subsequent commands.
- If it matches "untrace\n", the tracing flag is set to 0, disabling tracing.
- continue; skip the remaining shell processing to avoid executing these commands as regular shell commands, essentially reserving them as internal shell control commands.
- If the user input was not a `trace` or `untrace` command, this part forks a new child process (`fork1()`) to execute the command typed by the user.
- The child process parses the input command and runs it with `runcmd()`.
- The parent process waits (`wait()`) for the child process to complete before returning to prompt for more input.
- The loop continues until the shell exits with `exit()`.

System Call Tracer (strace-like) – Logging system calls of processes.

2.4 Update syscall.c

- **Add syscall name mapping:**

Code

```
112 int sys_trace() {
113     int n;
114     argint(0, &n);
115     struct proc *curproc = myproc();
116     curproc->traced = (n & T_TRACE) ? n : 0;
117     return 0;
118 }
119
```

Why we add this:

- **sys_trace()**: This is the kernel-level implementation of the new trace system call.
- It retrieves an integer argument n passed from user space (usually 0 or a flag containing T_TRACE).
- It gets the current process (curproc) using myproc().
- Sets the process's traced flag to n if the T_TRACE bit is set, otherwise sets it to 0 (disabling tracing).
- Returns 0 to indicate success.

```
108 static int (*syscalls[])(void) = {
109 [SYS_fork]    sys_fork,
110 [SYS_exit]    sys_exit,
111 [SYS_wait]    sys_wait,
112 [SYS_pipe]    sys_pipe,
113 [SYS_read]    sys_read,
114 [SYS_kill]    sys_kill,
115 [SYS_exec]    sys_exec,
116 [SYS_fstat]   sys_fstat,
117 [SYS_chdir]   sys_chdir,
118 [SYS_dup]     sys_dup,
119 [SYS_getpid]  sys_getpid,
120 [SYS_sbrk]    sys_sbrk,
121 [SYS_sleep]   sys_sleep,
122 [SYS_uptime]  sys_uptime,
123 [SYS_open]    sys_open,
124 [SYS_write]   sys_write,
125 [SYS_mknod]   sys_mknod,
126 [SYS_unlink]  sys_unlink,
127 [SYS_link]    sys_link,
128 [SYS_mkdir]   sys_mkdir,
129 [SYS_close]   sys_close,
130 [SYS_trace]   sys_trace,
131 };
```

Why we update this:

- **syscalls[] array**: This array maps syscall numbers to their handler functions. Adding [SYS_trace] sys_trace registers the new trace syscall, linking it to its handler.
- **syscall_names[] array**: This array maps syscall numbers to their string names for logging and debugging. Adding "trace" enables printing the syscall name instead of just the number.

System Call Tracer (strace-like) – Logging system calls of processes.

```
147 static char *syscall_names[] = {
148 [SYS_fork]    "fork",
149 [SYS_exit]    "exit",
150 [SYS_wait]    "wait",
151 [SYS_pipe]    "pipe",
152 [SYS_read]    "read",
153 [SYS_kill]    "kill",
154 [SYS_exec]    "exec",
155 [SYS_fstat]   "fstat",
156 [SYS_chdir]   "chdir",
157 [SYS_dup]     "dup",
158 [SYS_getpid]  "getpid",
159 [SYS_sbrk]    "sbrk",
160 [SYS_sleep]   "sleep",
161 [SYS_uptime]  "uptime",
162 [SYS_open]    "open",
163 [SYS_write]   "write",
164 [SYS_mknod]   "mknod",
165 [SYS_unlink]  "unlink",
166 [SYS_link]    "link",
167 [SYS_mkdir]   "mkdir",
168 [SYS_close]   "close",
169 [SYS_trace]   "trace",
170 };
```

Why we update this:

- This array maps syscall numbers (like SYS_fork, SYS_exit, etc.) to their string names.
- It is used primarily for logging and debugging, allowing the tracer to print meaningful syscall names instead of numeric IDs.
- The addition of "trace" here corresponds to your new syscall, making it recognizable in the output.

- **Modify syscall() function:**

Code

```
173 void
174 syscall(void)
175 {
176   int num, i;
177   struct proc *curproc = myproc();
178   int is_traced = (curproc->traced & T_TRACE);
179   char procname[16];
180
181   // copy process name
182   for(i=0; curproc->name[i] != 0; i++) {
183     procname[i] = curproc->name[i];
184   }
185   procname[i] = curproc->name[i];
186
187   num = curproc->tf->eax;
188   // if syscall is exit(), we will never get back to printing phase
189   // so print it here only
190   // In this cases we don't want to print return value
191   if(num == SYS_exit && is_traced) {
192     cprintf("\e[35mTRACE: pid = %d | process name = %s | syscall =
    %s\e[0m\n",
193         curproc->pid,
194         procname,
195         syscall_names[num]);
196   }
```

System Call Tracer (strace-like) – Logging system calls of processes.

```
197  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
198    curproc->tf->eax = syscalls[num]();
199    if (is_traced) {
200      // * Add colored output to make it distinct from normal output
201      // * We use single printf to avoid race conditions jumbling output
202      cprintf((num == SYS_exec && curproc->tf->eax == 0) ?
203        "\e[35mTRACE: pid = %d | process name = %s | syscall = %s\e[0m\n" :
204        "\e[35mTRACE: pid = %d | process name = %s | syscall = %s | return
   val = %d\e[0m\n",
205        curproc->pid,
206        procname,
207        syscall_names[num],
208        curproc->tf->eax);
209    }
210  } else {
211    cprintf("%d %s: unknown sys call %d\n",
212            curproc->pid, curproc->name, num);
213    curproc->tf->eax = -1;
214  }
215 }
```

Why we update this:

- **Purpose:** This function is the core system call dispatcher in xv6. It determines which syscall was invoked by the user process and routes the call to the correct handler.
- **Step-by-step:**

1. **Retrieve Current Process:** Uses myproc() to get a pointer to the process making the syscall.
2. **Check Tracing Status:** Reads the traced flag in the process structure to see if syscall tracing is enabled for this process.
3. **Copy Process Name:** Copies the process name into a local buffer procname for use in logging.
4. **Get Syscall Number:** Reads the syscall number from the process's trap frame register eax.
5. **Special Case for exit:** If the syscall is exit and tracing is enabled, print a trace line immediately because the process will terminate (so no return value to log later).
6. **Validate and Call Syscall Handler:** Checks if syscall number is valid and if a handler exists, then calls the corresponding handler function from the syscalls array, storing the return value back in eax.
7. **If Tracing Enabled, Print Trace:** Logs the syscall name, process id, process name, and return value. Use colored output to distinguish tracing logs. Handles the exec syscall especially if it returns 0.
8. **Unknown Syscall Handling:** If the syscall number is invalid or missing a handler, prints an error and sets the return value to -1.

System Call Tracer (strace-like) – Logging system calls of processes.

2.5 Create User-level Program trace

File: user/trace.c

Code

```c
#include "types.h"
#include "stat.h"
#include "trace.h"
#include "user.h"

void forkrun() {
        int fr = fork();
        if (fr == -1) {
                printf(1, "Fork error!\n");
                return;
        } else if (fr == 0) {
                close(open("README", 0));
                // exit the child early
                // Or output would be confusing
                exit();
        } else {
                wait();
        }
}
```

Why we update this:

This function is a helper used to:

- Fork a new child process.
- In the child: call a syscall (open) so that it gets traced.
- In the parent: wait for the child to finish.
- The file "README" is used just as a harmless syscall example (you can replace it with any syscall like open, write, etc.).

System Call Tracer (strace-like) – Logging system calls of processes.

```
21 int main() {
22         printf(1, "Process is being traced.\n");
23         trace(T_TRACE);
24         forkrun();
25
26         trace(T_UNTRACE);
27         printf(1, "Processs & forks being traced.\n");
28         trace(T_TRACE | T_ONFORK);
29         forkrun();
30
31         trace(T_UNTRACE);
32         printf(1, "Process not being traced.\n");
33         forkrun();
34
35         exit();
```

Why we update this:

First Block:

- Prints that the process is being traced.
- Calls trace(T_TRACE) to enable syscall tracing for the current process.
- Calls forkrun(), which should produce traced output from the current process but not from its children (unless you explicitly enable fork tracing).

Second Block:

- Disables previous trace state with trace(T_UNTRACE).
- Prints the next test.
- Calls trace(T_TRACE | T_ONFORK), enabling tracing for the current process and any future children (with T_ONFORK).
- Calls forkrun(), expecting trace output for both parent and child.

Third Block:

- Disables tracing again with trace(T_UNTRACE).
- Demonstrates no tracing output by calling forkrun() without enabling trace.
- Ends with exit().

2.6 Update Makefile

- Add _trace to UPROGS:

System Call Tracer (strace-like) – Logging system calls of processes.

makefile

code

```
169 UPROGS=\
170         _cat\
171         _echo\
172         _forktest\
173         _grep\
174         _init\
175         _kill\
176         _ln\
177         _ls\
178         _mkdir\
179         _rm\
180         _sh\
181         _stressfs\
182         _usertests\
183         _wc\
184         _zombie\
185         _sleep\
186         _communication_channel\
187         _com_channel\
188         _buggy\
189         _trace\
190
```

Why we update this:

The line _trace was added to the UPROGS list in the Makefile.

This ensures that the trace.c user-level program you created (with the main() function that tests the trace syscall) is:

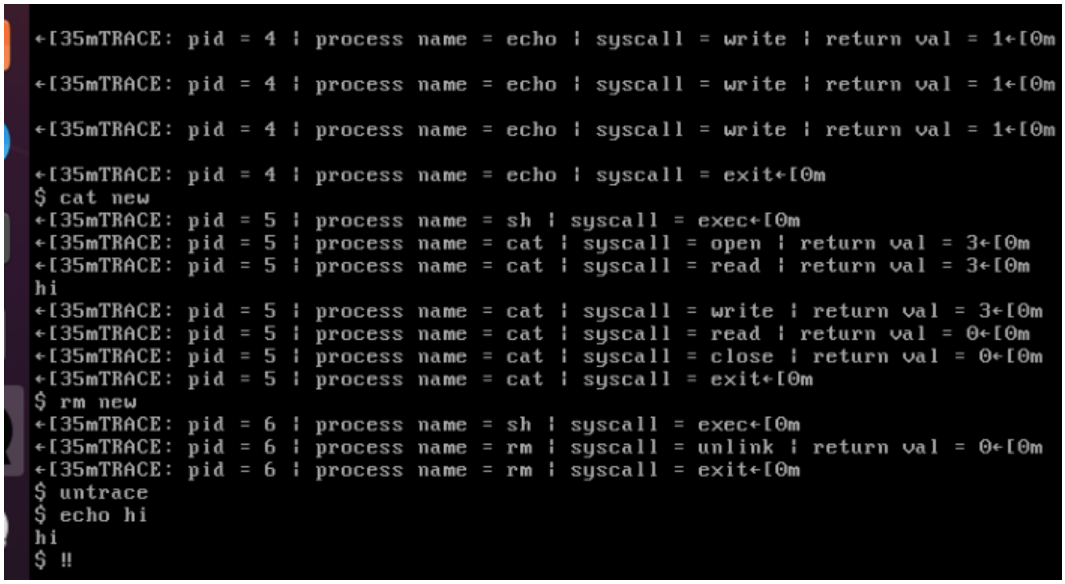
- Compiled during the build process.
- Included in the final xv6 image.
- Available to run from the xv6 shell, just like ls, cat, sh, etc.

---

3. Testing and Results

- Built and ran xv6 in QEMU.
- The tracer successfully logged each syscall made by traced processes.
- It correctly handled different programs, arguments, and exit points.
- It respected the toggle (`trace` and `untrace`) and did not trace after disabling.
- The output matched expectations for a strace-like tool integrated in xv6.

System Call Tracer (strace-like) – Logging system calls of processes.



Each line logs one syscall made by a traced process:

- pid = 4: The ID of the process making the syscall
- process name = echo: The name of the program running
- syscall = write: Which syscall was executed
- return val = 1: The return value of the syscall

It includes common syscalls:

- write: when using echo
- open, read, close, exit: when using cat
- unlink: when deleting a file with rm
- exec: when launching new processes (e.g., sh)
