

Jacinto Gomez, Fayed Gaya
jg6243, fg2271
CS-GY 6233 Introduction to Operating Systems
Final Project (strace)
May 1st, 2022

Kindly use Jacinto's Anubis Repo

How to run these programs:

For part 2: Type “strace on” in the terminal and type a command to print the system calls. Type “strace off” in the terminal and try the same command, and the system calls will not be printed. Type “strace run” followed by a command to get the system calls one time only. Type “strace dump” to see the full contents of the ring buffer at any time. Note that we set 16 to be the default buffer size; if you would like to change it, you can do so at line 18 of syscall.h that specifies the N value which is the buffer size. To trace child process run the straceTEST program by entering ‘straceTEST’ in the terminal. For more readable format, in syscall.c comment out lines 200, 211, 219, and 224, and uncomment lines 31-51, and 209. Then try running commands as usual

For part 5: Type “strace run leak” in the terminal to run the memory leak program. It will take about 20-30 seconds for the memory to fill, and in the end will output the last few calls made to sbrk and exit.

Part 1: Get familiar with Linux Strace

A. Learning about strace through its application on ls

```
anubis@anubis-ide < master@f6f2c68 > : ~/final-project-371525eb-jg6243 [
[0] % strace -c ls -l
total 736
-rw-r--r-- 1 anubis anubis 155 May 1 15:31 compile_flags.txt
-rw-r--r-- 1 anubis anubis 156 May 1 15:31 Dockerfile
-rw-r--r-- 1 anubis anubis 512000 May 1 15:54 fs.img
drwxr-xr-x 2 anubis anubis 4096 May 1 15:54 kernel
-rw-r--r-- 1 anubis anubis 368 May 1 15:31 launch.json.tpl
-rw-r--r-- 1 anubis anubis 1174 May 1 15:31 LICENSE
-rw-r--r-- 1 anubis anubis 6464 May 1 15:36 Makefile
drwxr-xr-x 2 anubis anubis 4096 May 1 15:54 mkfs
-rw-r--r-- 1 anubis anubis 3678 May 1 15:31 README.md
drwxr-xr-x 2 anubis anubis 4096 May 1 15:54 user
-rw-r--r-- 1 anubis anubis 198144 May 1 15:54 xv6.img
```

% time	seconds	usecs/call	calls	errors	syscall
43.78	0.000095	7	12		write
30.88	0.000067	3	21	4	openat
8.76	0.000019	1	19		fstat
8.29	0.000018	0	23		close
5.99	0.000013	0	16		read
2.30	0.000005	1	3		lseek
0.00	0.000000	0	34		mmap
0.00	0.000000	0	9		mprotect
0.00	0.000000	0	2		munmap
0.00	0.000000	0	3		brk
0.00	0.000000	0	2		rt_sigaction
0.00	0.000000	0	1		rt_sigprocmask
0.00	0.000000	0	2		ioctl
0.00	0.000000	0	2	2	access
0.00	0.000000	0	4		socket
0.00	0.000000	0	4	4	connect
0.00	0.000000	0	1		execve
0.00	0.000000	0	2	2	statfs
0.00	0.000000	0	1		arch_prctl
0.00	0.000000	0	14	14	getxattr
0.00	0.000000	0	11	11	lgetxattr
0.00	0.000000	0	1		futex
0.00	0.000000	0	2		getdents64
0.00	0.000000	0	1		set_tid_address
0.00	0.000000	0	1		set_robust_list
0.00	0.000000	0	1		prlimit64
0.00	0.000000	0	11		statx
100.00	0.000217	1	203	37	total

As displayed in the above screenshots we ran strace on the ls command. Since we used the -c option a neat table is output containing information about the system calls being made, the total number of system calls, and the time it took for strace to run on the ls command.

B. 4 system calls and how they work for “echo hi”

We are now going to pick 4 system calls and explain their functionality within the context of the echo command. Here is a shot of us running the command echo with the argument “hi”:

4. close()

Close simply closes a file descriptor so that it can be used again. In our context close is called several times closing all of the file descriptors that were opened during the execution of echo.

Part 2: Building Strace in xv6

A. “Strace on” and “strace off”

In essence, we implemented strace on and off in two parts. The first part was a modification to the shell to detect whether “strace on” or “strace off” had been entered by the user. This was done by analyzing what had been entered by the user in the buffer and matching it to “strace on” and “strace off” that had been stored as strings within the shell.

The definition and initialization of the tracer flag along with the key phrases to set and unset it:

```
13  #define BACK 5
14
15  #define MAXARGS 10
16
17  int tracerFlag = 0;
18  char traceEnable[] = "strace on\n";
19  char traceDisable[] = "strace off\n";
20
```

These key strings would set or unset an strace flag in the shell.

The function we wrote to compare user input to the shell with the key phrases:

```
int streq(char* a, char*b) // This function will be used to compare two strings and modify the tracerFlag variable
{
    while(1)
    {
        if (*a != *b)
        {
            return 0;
        }
        if (*a == '\n') return 1;
        a++;
        b++;
    }
}
```

Modifying the getcmd() function to check if what the user entered (what ends up in the buffer) is a keyphrase to set or unset the tracer flag:

```

    if (strcmp(buf, traceEnable)) // check if strace on was entered
    {
        tracerFlag = 1;
        continue;
    }
    if (strcmp(buf, traceDisable)) //check if strace off was entered
    {
        tracerFlag = 0;
        continue;
    }
    if (fork1() == 0)
        runcmd(parsecmd(buf));
    wait();
}
exit();
}

```

And the tracer flag based system call `strace()` dependant on if the `tracerflag` was up, added to the `executemd` code:

```

case EXEC:
    ecmd = (struct execcmd *)cmd;
    if (ecmd->argv[0] == 0)
        exit();
    if (tracerFlag) strace(straceOn | straceFork); // Tracing is activated if the tracer flag has been set by "strace on";
    exec(ecmd->argv[0], ecmd->argv);
    printf(2, "exec %s failed\n", ecmd->argv[0]);
    break;

```

If the flag was set the `strace` system call would be made before a command was actually executed ensuring that process had its tracer attribute set and would be traced. If the flag was not set, the `strace` system call would simply not be made. It is important to not

The second part was what it actually meant for a process to be traced and what the `strace` system call did. This boiled down to adding a trace flag on the process structure that was initialized to off when the process was created. We also needed to create a new headerfile in the kernel space with the definitions of what the arguments passed to `strace()` meant:

straceTEST.c	proc.c	sh.c	memleak.c	syscall.c	Makefile	syscall.h	proc.h	usys.S	strace.h
1	#define	straceOn	1						
2	#define	straceFork	2						
3	#define	straceOff	0						

Modifications to the process struct, note the new attribute ‘`traceflag`’:

```

// For process state
struct proc {
    uint sz; // Size of process memory (bytes)
    pde_t* pgdir; // Page table
    char *kstack; // Bottom of kernel stack for this process
    enum procstate state; // Process state
    int pid; // Process ID
    struct proc *parent; // Parent process
    struct trapframe *tf; // Trap frame for current syscall
    struct context *context; // switch() here to run process
    void *chan; // If non-zero, sleeping on chan
    int killed; // If non-zero, have been killed
    struct file *ofile[NOFILE]; // Open files
    struct inode *cwd; // Current directory
    char name[16]; // Process name (debugging)
    int traceflag; // Indicates if a process is or is not being traced
};

```

The strace system call then went in and modified this attribute of a process to turn the flag on when it was called. The function was written in syscall.c:

```

125 int sys_strace() {
126     int x;
127     // int* cb=0;
128     // struct ringbuf* me=init(cb,16);
129     argint(0, &x); // we will use this function to grab parameters from user passed parameters
130     struct proc* curproc = proc;
131     curproc->traceflag = (x & straceOn) ? x : 0;
132     return 0;
133 }
134

```

We also made modifications to all of the files that need to be changed when a new system call is added, including proc.h, proc.c, sysproc.c, Makefile, usys.S, syscall.c, exec.c, syscall.h, defs.h, and user.h.

Next we added conditionals in the syscall function within syscall.c, the part of the kernel that actually issues system calls. If the system calls were being issued by a process that was actively being traced (As earlier set by the strace system call) a set of conditionals were written to print out information to the terminal using `cprintf()` about the system call that was issued including the process id, the process name, and the the name of the system call issued.

```

void syscall(void) {
    int num, j;

    struct proc* currentprocess = proc;
    int isTraced = (currentprocess->traceflag & straceOn);
    char processName[16];

    for (j = 0; currentprocess->name[j] != 0; j++) // We copy over the process name
    {
        processName[j] = currentprocess->name[j];
    }
    processName[j] = currentprocess->name[j];

    num = currentprocess->tf->eax;

    addtobuf(currentprocess->pid, processName, syscall_names[num]);
    // char* exit="exit";
    // printf("process name %s\n", processName);
    // if (syscall_names[num] == exit) {
    //     printf("exit\n");
    // }
    if (num == SYS_exit && isTraced)
    {
        // addtobuf(buf, currentprocess->pid);
        // printf("exit\n");
        printf("\e[35mTRACE: pid = %d | process name = %s | syscall = %s\n", currentprocess->pid, processName, syscall_names[num]);
    }

    if (num > 0 && num < NELEM(syscalls) && syscalls[num])
    {
        proc->tf->eax = syscalls[num];
        if (isTraced) // Colouring the output to differentiate it from regular output. Successful exit system calls are discarded.
        {
            printf((num == SYS_exec && currentprocess->tf->eax == 0) ? "\e[35mTRACE: pid = %d | process name = %s | syscall = %s\n" : "\e[35mTRACE: pid = %d | process name = %s | syscall = %s | return val = %d\n", currentprocess->pid, processName, syscall_names[num], currentprocess->tf->eax);
        }
    }
    else {
        printf("pid %d: unknown sys call %d\n", proc->pid, proc->name, num);
        proc->tf->eax = -1;
    }
}

```

As you can notice, since the `exit()` sys call would not allow for any printing after its execution, when an `exit` syscall was detected in the `eax` buffer, that system call was printed out first. Otherwise syscalls were printed when they were found to exist. If the value in `eax` was `-1`, this was not a sys call that was recognized and an appropriate error message was printed instead.

For us to print the syscall names and not the integer values in the `eax` register, we created a new array called `syscall_names` that was indexed to the integer value of a system call from which the `printf()` function drew.

```

153 //Adding an array of the actual names of the system calls so that we can print out names instead of numbers
154 static char *syscall_names[NELEM(syscalls)] = {
155     [SYS_fork]    "fork", [SYS_wait]    "wait",
156     [SYS_pipe]    "pipe", [SYS_kill]    "kill",
157     [SYS_exec]    "exec", [SYS_chdir]    "chdir",
158     [SYS_dup]     "dup", [SYS_getpid]    "getpid", [SYS_sbrk]    "sbrk",
159     [SYS_sleep]   "sleep", [SYS_uptime]  "uptime", [SYS_open]    "open",
160     [SYS_write]   "write", [SYS_mknod]   "mknod", [SYS_unlink]   "unlink",
161     [SYS_link]    "link", [SYS_mkdir]    "mkdir", [SYS_close]    "close",
162     [SYS_strace]   "strace", [SYS_stracedump] "stracedump",
163 };

```

Here are some screenshots of `strace` on and `strace` off in action:


```

181 }
182 //if the buffer starts with "strace run " set the strace flag to 2, and then chop the buffer then run what remains
183 if (buf[0] == 's' && buf[1] == 't' && buf[2] == 'r' && buf[3] == 'a' && buf[4] == 'c' && buf[5] == 'e' && buf[6] == ' ' && buf[7] == 'r' && buf[8] == 'u' && buf[9] == 'n' && buf[10] == ' '){
184     tracerFlag = 2; // set tracerflag
185     for (int i = 0; i < (sizeof(buf) - 11); i++){
186         buf[i] = buf[i + 11];
187     }
188     if (fork1() == 0) // the child process runs the command
189     {
190         runcmd(parsecmd(buf));
191     }
192     tracerFlag = 0; // Then reset the tracer flag
193     wait();
194     continue;
195 }

```

If the buffer began with that triggering key phrase then the shell itself would turn the traceflag on, chop the buffer to remove ‘strace run ‘, / the first 11 characters, execute what ever was remaining in the buffer, including commands that would fail, and then covertly unset the trace flag before resuming the shell program. This allowed for A successful limited application of strace on the command it was called with through the use of ‘strace run <command>’. Here is an example of it working.

```

Problems 42 make qemu x
SeaBIOS (version 1.14.0-2)

iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8F4C0+1FECF4C0 CA00

Booting from Hard Disk..xv6...
cpu0: starting
ssb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
ini: starting sh
$ echo hi
hi
$ strace run echo hi
TRACE: pid = 4 | process name = sh | syscall = exec
hTRACE: pid = 4 | process name = echo | syscall = write | return val = 1
iTRACE: pid = 4 | process name = echo | syscall = write | return val = 1

TRACE: pid = 4 | process name = echo | syscall = write | return val = 1
TRACE: pid = 4 | process name = echo | syscall = exit
$ echo hi
hi
$ █

```

C. Strace dump

In order to implement strace dump we first modified the shell so as to recognize the strace dump command. This was done by implementing a hardcoded check for the string “strace dump\n”.

A screenshot of the modiciation to the shell’s getcommand():

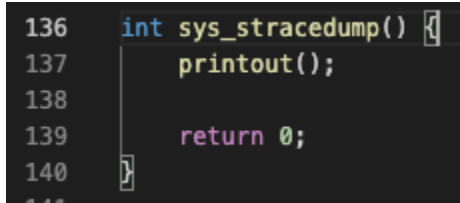
```

// if the buffer starts with "strace dump" call the strace_dump system call
if (buf[0] == 's' && buf[1] == 't' && buf[2] == 'r' && buf[3] == 'a' && buf[4] == 'c' && buf[5] == 'e' && buf[6] == ' ' && buf[7] == 'd' && buf[8] == 'u' && buf[9] == 'm' && buf[10] == 'p' && buf[11] == '\n'){
    stracedump();
    continue;
}

```

If the key string was detected, a system call to `stracedump` was made. The `stracedump` `syscall()` simply called the `printout` method that was defined as part of our implementation of a circular buffer. Of course the implementation of another system call involved modifying the usual files and adding the name of this system call to our new array of system call names.

A screenshot:

A screenshot of a code editor showing the implementation of the `sys_stracedump()` function. The code is as follows:

```
136 int sys_stracedump()  
137     printout();  
138  
139     return 0;  
140
```

So unlike `strace on` and `strace off`, `strace dump` had to dump a collection of the `N` most recent events, or system calls. In order to do this we implemented a circular buffer and functionality that allowed for us to continuously collect system calls as our instance of `xv6` runs.

`Strace` by default uses a 2 dimensional array of `N` `buf` structs. `Buf` is a struct containing the process `pid`, process name and `syscall` name. The function `printout()` prints the `N` most recent system calls that were stored in the buffer `buf`. The variable `N` is defined in our implementation as 16 since that was what was mentioned in class, but it can be changed to different integer amounts. The counter variable “`place`” tracks the index of the buffer, and resets to 0 once it is about to go over `N`, that way the oldest buffer entry always gets deleted for a new one. The counter variable “`track`” is used for printing to output; it essentially does the same thing as “`place`” but is separate so that its changing doesn't affect buffer indexing.

D. Trace child processes

In order to implement the tracing of child processes we modified the `fork()` function in `proc.c` to initialize the trace flag value of the forked process depending on the fork value of the process that called `fork`!

As evidenced by this screenshot:

```

int fork(void) {
    int i, pid;
    struct proc *np;

    // Allocate process.
    if ((np = allocproc()) == 0)
        return -1;

    // Copy process state from p.
    if ((np->pgdir = copyuvm(proc->pgdir, proc->sz)) == 0) {
        kfree(np->kstack);
        np->kstack = 0;
        np->state = UNUSED;
        return -1;
    }
    np->traceflag = (proc->traceflag & straceFork) ? proc->traceflag : straceOff;
    np->sz = proc->sz;
    np->parent = proc;
    *np->tf = *proc->tf;

    // Clear %eax so that fork returns 0 in the child.
    np->tf->eax = 0;
}

```

This is evidenced by calling strace on straceTEST which spawns child processes. Our straceTEST program send a message notifying the caller that the strace system call is about to get turned on, then it calls strace(), forks the process, continues to trace, and then exits. Here it is in action:

Problems 42 make qemu x

SeaBIOS (version 1.14.0-2)

iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8F4C0+1FECF4C0 CA00

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

\$ straceTEST

The process is now being traced!

TRACE: pid = 3 | process name = straceTEST | syscall = fork | return val = 4

TRACE: pid = 3 | process name = straceTEST | syscall = wait | return val = 4

TRACE: pid = 3 | process name = straceTEST | syscall = strace | return val = 0

Process and its forked processes are now being traced!

TRACE: pid = 3 | process name = straceTEST | syscall = fork | return val = 5

TRACE: pid = 5 | process name = straceTEST | syscall = open | return val = -1

TRACE: pid = 5 | process name = straceTEST | syscall = close | return val = -1

TRACE: pid = 5 | process name = straceTEST | syscall = exit

TRACE: pid = 3 | process name = straceTEST | syscall = wait | return val = 5

TRACE: pid = 3 | process name = straceTEST | syscall = strace | return val = 0

Process no longer being traced!

\$ █

Extra credits: Formatting more readable output

We decided to make the output more readable by printing command results first then showing all strace commands after. This way, the user can view the results of a command in proper form, then be able to debug by looking at the steps after. We implemented this by storing all commands in an array called “pretty” and outputting these results after the system calls finish executing. Two examples are shown below:

```

kill          2 9 14760
leak          2 10 14840
ln            2 11 14656
ls            2 12 17132
mkdir         2 13 14808
ring          2 14 13784
rm            2 15 14792
sh            2 16 29436
stressfs      2 17 15356
usertests     2 18 65220
wc            2 19 16276
zombie        2 20 14328
straceTEST    2 21 15212
console       3 22 0
TRACE: pid = 0 | process name = (null) | syscall = (null) | return val = 0
TRACE: pid = 0 | process name = (null) | syscall = (null) | return val = 0
TRACE: pid = 0 | process name = (null) | syscall = (null) | return val = 0
TRACE: pid = 0 | process name = (null) | syscall = (null) | return val = 0

```

```

$ strace run echo hello
hello
TRACE: pid = 0 | process name = (null) | syscall = (null) | return val = 0
TRACE: pid = 0 | process name = (null) | syscall = (null) | return val = 0
TRACE: pid = 0 | process name = (null) | syscall = (null) | return val = 0
TRACE: pid = 0 | process name = (null) | syscall = (null) | return val = 0
TRACE: pid = 0 | process name = (null) | syscall = (null) | return val = 0
TRACE: pid = 0 | process name = (null) | syscall = (null) | return val = 0
TRACE: pid = 0 | process name = (null) | syscall = (null) | return val = 0
TRACE: pid = 0 | process name = (null) | syscall = (null) | return val = 0
TRACE: pid = 0 | process name = (null) | syscall = (null) | return val = 0
TRACE: pid = 0 | process name = (null) | syscall = (null) | return val = 0
TRACE: pid = 0 | process name = (null) | syscall = (null) | return val = 0
TRACE: pid = 0 | process name = (null) | syscall = (null) | return val = 0
TRACE: pid = 0 | process name = (null) | syscall = (null) | return val = 0
TRACE: pid = 0 | process name = (null) | syscall = (null) | return val = 0
TRACE: pid = 0 | process name = (null) | syscall = (null) | return val = 0

```

Part 3: Building options for strace

A. Option -e

We spent significant time in discussion and several attempted implementations of option -e. What we had was to change the way we store system calls. Instead of using conditionals to print them directly as soon as they were made in the syscall function, we would create a second buffer, one of very long length that persisted throughout the lifetime of a process. When that process made a system call that system call, along with other info, like the process name, id, and the return value would then get added to the process specific buffer. When it was time for the process to make its LAST systemcall, presumably exit(), we would dump the output of the process specific buffer to the terminal. This way we would have a list with attributes, most relevant to the implementation of option -e, was the syscall name. If option -e was entered we could filter that dump so that only the system calls that had been passed to the command 'strace -e <system call name>' where system call name == the name attribute that we stored in our process specific buffer. The pseudo code for that is something like:

```

For(int i = 0; i < processbuffersize(); i++) {
if (i[name] == passedsystemcallname) {
print(process id, process name)
}
}

```

Alternatively we could also implement option -e by setting the flags as a part of the proc structure itself. And then we would go on to modify the conditional statements in the sysprint call to factor in a proc's proc->-e flag, proc->systemcallname, and what not.

B. Option -s

Similarly to the work we put in to theorizing option -e, we use the process specific buffer approach or set additional process flags so that the conditionals in the syscall(void) function would trigger based on if a proc's proc->-s flag was up. In this case we would only print successful system calls.

C. Option -f

Similarly to the work we put in to theorizing option -e, and conversely to the implementation of the -s option, we would manipulate the conditionals in the syscall function to only trigger if the syscall was successful if directed by the proc's proc->-f flag.

D. Option runs only once

As our current theoretical options implementation is process specific, as it modifies that process's attributes, this would naturally be the case. We could call the system calls in a way such that they would be applied to the processes depending on the command they were entered with

E. Extra Credit: Combining Options

As our current theoretical options implementation is process attribute flag based, it would not be difficult to combine them, we would simply write more logic in the syscall function in syscall.c to account for multiple options being passed instead of one. So a case (if -e), case if(-e -s).....

F. Extra Credits: Implement -c Option

This would force us to use an array that collect system calls and the other called for data on a process by process basis and then dump that data at the end of a process's life in the neat format.

Part 4: Output of strace to file

Theoretically, we would go with editing the README file here as that is easier for the kernel to deal with than to create a new file from scratch. We would do this by instead printing out system calls to the terminal, opening(), writing(), and then closing the readme file!

Part 5: Application of strace

We wrote a simple program to cause a memory leak in the system. The program dynamically allocates memory using malloc in a for loop, but never frees the memory. This eventually causes the system to run out of usable memory and crash. When running STrace on this program we can see that it causes an error in the sbrk() system call. The sbrk() system call dynamically changes the amount of memory allocated for a calling process by adjusting the program's break value. In the memory leak, there was no more free memory to allocate for the calling process, causing sbrk() to return -1.

As you can see from the screenshot below, this is what happens after a short amount of time when you execute the program with strace off. The process makes successful sbrk system calls until there is no more memory to be allocated resulting in a failed sbrk system call as indicated by the -1 value. Then exit() is called to terminate the running process.

```
TRACE: pid = 3 | process name = leak | syscall = sbrk | return val = 231944192
TRACE: pid = 3 | process name = leak | syscall = sbrk | return val = 231976960
TRACE: pid = 3 | process name = leak | syscall = sbrk | return val = 232009728
TRACE: pid = 3 | process name = leak | syscall = sbrk | return val = 232042496
TRACE: pid = 3 | process name = leak | syscall = sbrk | return val = 232075264
TRACE: pid = 3 | process name = leak | syscall = sbrk | return val = 232108032
TRACE: pid = 3 | process name = leak | syscall = sbrk | return val = 232140800
TRACE: pid = 3 | process name = leak | syscall = sbrk | return val = 232173568
TRACE: pid = 3 | process name = leak | syscall = sbrk | return val = 232206336
TRACE: pid = 3 | process name = leak | syscall = sbrk | return val = 232239104
TRACE: pid = 3 | process name = leak | syscall = sbrk | return val = 232271872
TRACE: pid = 3 | process name = leak | syscall = sbrk | return val = 232304640
TRACE: pid = 3 | process name = leak | syscall = sbrk | return val = 232337408
allocvm out of memory
TRACE: pid = 3 | process name = leak | syscall = sbrk | return val = -1
TRACE: pid = 3 | process name = leak | syscall = exit
```

With a buffer size of 16 we see the last few calls that the system made. The first 2 write calls are for the last 2 letters of “allocvm out of memory” then, the next few read calls are for reading “

strace dump,” the penultimate call is to the stracedump system call itself, then the final call is to exit the program.

```
allocuvm out of memory
TRACE: pid = 3 | process name = leak | syscall = sbrk | return val = -1
TRACE: pid = 3 | process name = leak | syscall = exit
$ strace dump
TRACE: pid = 2 | process name = sh | syscall = write | return val = 16
TRACE: pid = 2 | process name = sh | syscall = write | return val = 16
TRACE: pid = 2 | process name = sh | syscall = read | return val = 5
TRACE: pid = 2 | process name = sh | syscall = read | return val = 5
TRACE: pid = 2 | process name = sh | syscall = read | return val = 5
TRACE: pid = 2 | process name = sh | syscall = read | return val = 5
TRACE: pid = 2 | process name = sh | syscall = read | return val = 5
TRACE: pid = 2 | process name = sh | syscall = read | return val = 5
TRACE: pid = 2 | process name = sh | syscall = read | return val = 5
TRACE: pid = 2 | process name = sh | syscall = read | return val = 5
TRACE: pid = 2 | process name = sh | syscall = read | return val = 5
TRACE: pid = 2 | process name = sh | syscall = read | return val = 5
TRACE: pid = 2 | process name = sh | syscall = read | return val = 5
TRACE: pid = 2 | process name = sh | syscall = read | return val = 5
TRACE: pid = 2 | process name = sh | syscall = read | return val = 5
TRACE: pid = 2 | process name = sh | syscall = stracedump | return val = 23
TRACE: pid = 3 | process name = | syscall = exit | return val = 2
```

In Linux we see that running strace on a memory leak file results in the process being killed once memory is full. *This was done using the setup of HW4, since that homework already had a file with the <stdlib.h> header to use malloc for the memory leak*


```
brk(0x5627a1cd9000) = 0x5627a1cd9000
brk(0x5627a1cfc000) = 0x5627a1cfc000
brk(0x5627a1d1e000) = 0x5627a1d1e000
brk(0x5627a1d40000) = 0x5627a1d40000
brk(0x5627a1d62000) = 0x5627a1d62000
brk(0x5627a1d84000) = 0x5627a1d84000
brk(0x5627a1da7000) = 0x5627a1da7000
brk(0x5627a1dc9000) = 0x5627a1dc9000
brk(0x5627a1deb000) = 0x5627a1deb000
brk(0x5627a1e0d000) = 0x5627a1e0d000
brk(0x5627a1e2f000) = 0x5627a1e2f000
brk(0x5627a1e52000) = 0x5627a1e52000
brk(0x5627a1e74000) = 0x5627a1e74000
brk(0x5627a1e96000) = 0x5627a1e96000
brk(0x5627a1eb8000) = 0x5627a1eb8000
brk(0x5627a1eda000) = 0x5627a1eda000
brk(0x5627a1efd000) = 0x5627a1efd000
brk(0x5627a1f1f000) = 0x5627a1f1f000
brk(0x5627a1f41000) = 0x5627a1f41000
brk(0x5627a1f63000) = 0x5627a1f63000
brk(0x5627a1f86000) = 0x5627a1f86000
brk(0x5627a1fa8000) = 0x5627a1fa8000
brk(0x5627a1fca000) = 0x5627a1fca000
+++ killed by SIGKILL +++
[1] 569 killed strace ./leak
anubis@anubis-ide: ~$ ./main 27ec271 0 0 0 /home/anubis4_25777be2_jc6242
```