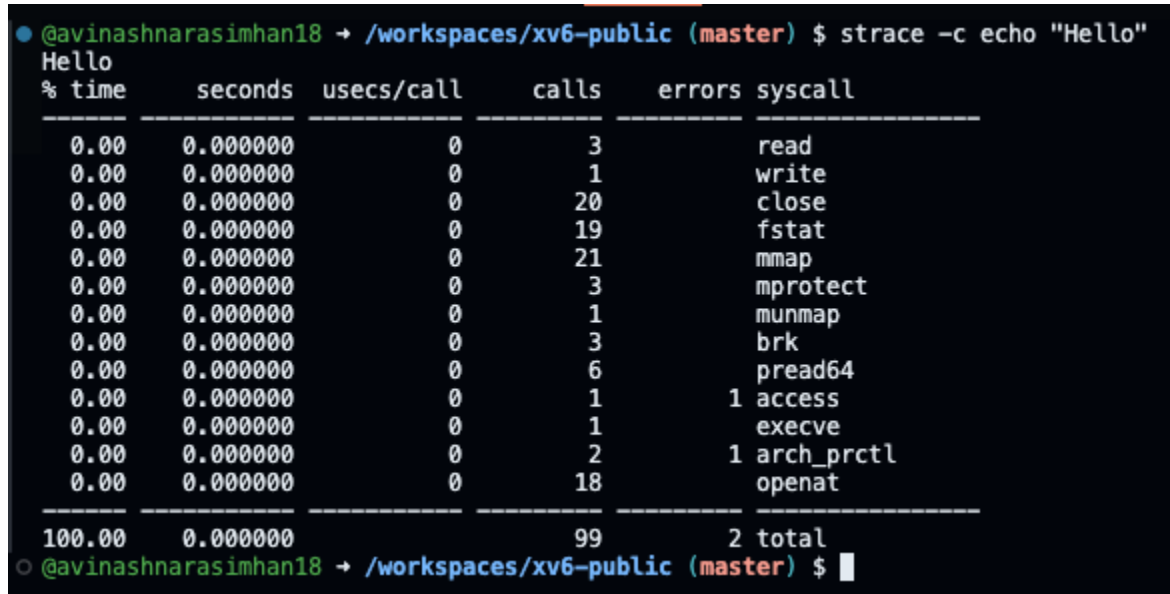


OS Final Project Report

Task 4.1.1 -

We ran the command `strace -c echo "Hello"` and got the following result showing the list of system calls made, total number of calls and time taken to run each call.



% time	seconds	usecs/call	calls	errors	syscall
0.00	0.000000	0	3		read
0.00	0.000000	0	1		write
0.00	0.000000	0	20		close
0.00	0.000000	0	19		fstat
0.00	0.000000	0	21		mmap
0.00	0.000000	0	3		mprotect
0.00	0.000000	0	1		munmap
0.00	0.000000	0	3		brk
0.00	0.000000	0	6		pread64
0.00	0.000000	0	1	1	access
0.00	0.000000	0	1		execve
0.00	0.000000	0	2	1	arch_prctl
0.00	0.000000	0	18		openat
100.00	0.000000		99	2	total

Task 4.1.2 -

We ran the same command `strace -c echo "Hello"` and detailed 4 system calls from those listed in the screenshot above.

1. Read: The read system call reads data from a file descriptor into memory. For the echo command, it might be used to read input from the shell or other resources, such as shared libraries or configuration files.
2. Write: The write system call sends data to an output file descriptor. In this case, echo writes the string "hello" followed by a new line to the standard output (file descriptor 1).
3. Mmap: The mmap system call maps files or shared libraries into memory. When running echo, mmap loads shared libraries or allocates memory for execution.
4. Execve: The execve system call executes the program specified by the path (in this case, /bin/echo). It replaces the current process image with the new program and starts its execution.

Building Strace in xv6 -

To implement strace, we decided that the `syscall()` function in `syscall.c` is a good area to input our functionality. This is because it can be easily used to map out all the system calls currently in use/used for a program in xv6.

Please note that we have implemented Task 4.2.6 as well where we suppress the command outputs while strace is on. So, all the screenshots below will only show the trace events logged.

Task 4.2.1 (strace on) -

The strace on functionality in xv6 is implemented by enabling a per-process tracing mechanism using the `strace_enabled` flag in `struct proc`.

When a user executes `strace` on, the `strace.c` user program invokes the `sys_strace()` system call with the parameter 1, which sets `p->strace_enabled = 1` for the calling process. This activates tracing for all subsequent system calls made by the process. The `syscall()` function in `syscall.c` checks this flag during each system call execution. If tracing is enabled, it prints the system call details—such as PID, name, syscall, and return value.

```
// Log the trace event if enabled
if (p->strace_enabled && num != SYS_strace) {
    add_trace_event(p->pid, p->name, syscall_names[num], ret_val);
    cprintf("TRACE: pid = %d | command_name = %s | syscall = %s | return_value = %d\n",
           p->pid, p->name, syscall_names[num], ret_val);
}
```

The following screenshot shows a successful run of the “strace on” functionality. We first turn the strace on and all subsequent commands are traced successfully.

[illegible]

Task 4.2.2 (strace off) -

The strace off functionality in xv6 disables tracing for the calling process by setting the `strace_enabled` flag in `struct proc` to 0. When the user executes strace off, the `strace.c` user program invokes the `sys_strace()` system call with the parameter 0, which clears the `p->strace_enabled` flag for the process. This ensures that no further system calls are traced or logged. Within the `syscall()` function in `syscall.c`, the absence of tracing is confirmed by checking the `strace_enabled` flag, and if it is 0, no logging or suppression of output occurs for system calls, effectively returning the process to normal, untraced execution.

The following screenshot shows a successful run of the “strace off” functionality. We first turn on strace, run a few commands to see the trace and then turn off strace. Subsequent commands will run normally.

```
$ strace on
$ echo hi
TRACE: pid = 33 | command_name = sh | syscall = sbrk | return_value = 16384
TRACE: pid = 33 | command_name = echo | syscall = exec | return_value = 0
TRACE: pid = 33 | command_name = echo | syscall = write | return_value = 1
TRACE: pid = 33 | command_name = echo | syscall = write | return_value = 1
TRACE: pid = 33 | command_name = echo | syscall = write | return_value = 1
$ strace off
TRACE: pid = 34 | command_name = sh | syscall = sbrk | return_value = 16384
TRACE: pid = 34 | command_name = strace | syscall = exec | return_value = 0
$ echo hi
hi
$ echo hello
hello
$
```

The toggle for the strace flag is fetched from the strace user space program.

```
if (strcmp(argv[1], "on") == 0) {
    strace(1);
} else if (strcmp(argv[1], "off") == 0) {
    strace(0);
}
```

Task 4.2.3 (strace run <command>) -

The strace run functionality in xv6 enables tracing for a specific command and its child processes. When the user executes `strace run <command>`, the `strace.c` user program forks a new process. In the child process, the `sys_strace()` system call is invoked with the parameter 1 to set the `strace_enabled` flag to 1, enabling tracing for that process. The child then executes the specified command using `exec(argv[2], &argv[2])`, ensuring all system calls made during the execution of the command are traced. Meanwhile, the parent process waits for the child to complete execution using `wait()` and then disables tracing for itself by calling `sys_strace()` with the parameter 0. This mechanism ensures that tracing is confined to the command and its forked processes while maintaining the parent process's normal execution state.

The following screenshot shows how the “strace run” command is run without turning strace off or on.

```
$ strace run echo hi
TRACE: pid = 46 | command_name = echo | syscall = exec | return_value = 0
TRACE: pid = 46 | command_name = echo | syscall = write | return_value = 1
TRACE: pid = 46 | command_name = echo | syscall = write | return_value = 1
TRACE: pid = 46 | command_name = echo | syscall = write | return_value = 1
$ echo hi
hi
$
```

Task 4.2.4 (strace dump) -

The strace dump functionality in xv6 allows the user to view the logged system call trace events for the current process. When the user executes strace dump, the strace.c user program invokes the sys_strace_dump() system call, which iterates through a circular buffer (strace_buffer) defined in proc.h. The add_trace_event function is responsible for logging details into the circular buffer.

```
// Log the trace event if enabled
if (p->strace_enabled && num != SYS_strace) {
    add_trace_event(p->pid, p->name, syscall_names[num], ret_val);
    cprintf("TRACE: pid = %d | command_name = %s | syscall = %s | return_value = %d\n",
           p->pid, p->name, syscall_names[num], ret_val);
}
```

This buffer stores details of traced system calls, such as PID, name, syscall, and return_value. The sys_strace_dump() implementation in sysproc.c retrieves and formats each trace event from the buffer and outputs it to the file descriptor associated with stdout. If the trace buffer contains no events, no output is generated.

```
void add_trace_event(int pid, char *name, char *syscall, int return_value) {
    struct trace_event *event = &strace_buffer[strace_index];

    event->pid = pid;
    safestrncpy(event->name, name, sizeof(event->name));
    safestrncpy(event->syscall, syscall, SYSCALL_NAME_LEN);
    event->return_value = return_value;

    strace_index = (strace_index + 1) % STRACE_BUFFER_SIZE; // Move to next index
    if (strace_count < STRACE_BUFFER_SIZE) {
        strace_count++;
    }
}
```

We define the N (STRACE_BUFFER_SIZE) latest events in proc.h.

```
#define STRACE_BUFFER_SIZE 10
```

The following screenshot shows a successful run of strace dump, displaying the 10 latest events.

```
$ strace on
$ echo hi
TRACE: pid = 4 | command_name = sh | syscall = sbrk | return_value = 16384
TRACE: pid = 4 | command_name = echo | syscall = exec | return_value = 0
TRACE: 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 = write | return_value = 1
$ echo hello
TRACE: pid = 5 | command_name = sh | syscall = sbrk | return_value = 16384
TRACE: pid = 5 | command_name = echo | syscall = exec | return_value = 0
TRACE: pid = 5 | command_name = echo | syscall = write | return_value = 1
TRACE: pid = 5 | command_name = echo | syscall = write | return_value = 1
TRACE: pid = 5 | command_name = echo | syscall = write | return_value = 1
TRACE: pid = 5 | command_name = echo | syscall = write | return_value = 1
TRACE: pid = 5 | command_name = echo | syscall = write | return_value = 1
TRACE: pid = 5 | command_name = echo | syscall = write | return_value = 1
TRACE: pid = 5 | command_name = echo | syscall = write | return_value = 1
$ strace off
TRACE: pid = 6 | command_name = sh | syscall = sbrk | return_value = 16384
TRACE: pid = 6 | command_name = strace | syscall = exec | return_value = 0
$ strace dump
TRACE EVENT: pid = 5 | command_name = sh | syscall = sbrk | return_value = 16384
TRACE EVENT: pid = 5 | command_name = echo | syscall = exec | return_value = 0
TRACE EVENT: pid = 5 | command_name = echo | syscall = write | return_value = 1
TRACE EVENT: pid = 5 | command_name = echo | syscall = write | return_value = 1
TRACE EVENT: pid = 5 | command_name = echo | syscall = write | return_value = 1
TRACE EVENT: pid = 5 | command_name = echo | syscall = write | return_value = 1
TRACE EVENT: pid = 5 | command_name = echo | syscall = write | return_value = 1
TRACE EVENT: pid = 5 | command_name = echo | syscall = write | return_value = 1
TRACE EVENT: pid = 6 | command_name = sh | syscall = sbrk | return_value = 16384
TRACE EVENT: pid = 6 | command_name = strace | syscall = exec | return_value = 0
$ █
```

Task 4.2.5 (Tracing child processes) -

We created a program called traceChild.c. This program creates multiple child processes in xv6, with the default being 3 (customizable via a command-line argument). The parent process prints its PID and uses a loop to fork child processes. Each child process prints its PID, executes the **echo hello** command using the exec system call, and exits. If exec fails, an error message is printed before exiting. The parent waits for each child to complete using wait() and after all children finish, prints a final message and exits.

We implemented strace for child processes by enabling tracing during the fork system call. In syscall.c, within the syscall() function, after a fork system call is executed, the newly created child process is located using its parent's pid via a helper function like find_child_process(). If the parent process has strace enabled (strace_enabled flag in struct proc), the child's strace_enabled flag is also set to 1, ensuring that tracing is automatically enabled for the child process. This allows all system calls made by the child process to be logged in the same way as the parent. Additionally, the parent process waits for the child to finish, preserving the strace functionality throughout the execution of the forked process.

```
// Handle fork explicitly
if (num == SYS_fork && p->strace_enabled) {
    struct proc *child = find_child_process(p->pid);
    if (child) {
        child->strace_enabled = 1; // Enable tracing for the child
    }
}
}
```

The following screenshots show a successful run of tracing child processes. While strace is off, traceChild executes normally. Once strace is on, we can see the fork to create child processes, and exec to execute traceChild and its child commands. Each child process logs its exec and multiple write calls for output. The parent synchronizes using wait to ensure all children complete before exiting. We can see how the parent PID 30 spawns PID 31, 32 and 33, which execute separately and the wait call as well.

```
$ traceChild
Parent pid: 25
Child pid: 26
Child 26: Executing 'echo hello'
hello
Child pid: 27
Child 27: Executing 'echo hello'
hello
Child pid: 28
Child 28: Executing 'echo hello'
hello
All children finished. Exiting parent process.
```

Strace is on now. We can see the fork call spawning PID 31.

```
$ strace on
$ traceChild
TRACE: pid = 30 | command_name = sh | syscall = sbrk | return_value = 16384
TRACE: pid = 30 | command_name = traceChild | syscall = exec | return_value = 0
TRACE: pid = 30 | command_name = traceChild | syscall = getpid | return_value = 30
TRACE: pid = 30 | command_name = traceChild | syscall = write | return_value = 1
TRACE: pid = 30 | command_name = traceChild | syscall = write | return_value = 1
TRACE: pid = 30 | command_name = traceChild | syscall = write | return_value = 1
TRACE: pid = 30 | command_name = traceChild | syscall = write | return_value = 1
TRACE: pid = 30 | command_name = traceChild | syscall = write | return_value = 1
TRACE: pid = 30 | command_name = traceChild | syscall = write | return_value = 1
TRACE: pid = 30 | command_name = traceChild | syscall = write | return_value = 1
TRACE: pid = 30 | command_name = traceChild | syscall = write | return_value = 1
TRACE: pid = 30 | command_name = traceChild | syscall = write | return_value = 1
TRACE: pid = 30 | command_name = traceChild | syscall = write | return_value = 1
TRACE: pid = 30 | command_name = traceChild | syscall = write | return_value = 1
TRACE: pid = 30 | command_name = traceChild | syscall = write | return_value = 1
TRACE: pid = 30 | command_name = traceChild | syscall = write | return_value = 1
TRACE: pid = 30 | command_name = traceChild | syscall = write | return_value = 1
TRACE: pid = 30 | command_name = traceChild | syscall = write | return_value = 1
TRACE: pid = 30 | command_name = traceChild | syscall = fork | return_value = 31
TRACE: pid = 31 | command_name = traceChild | syscall = write | return_value = 1
```

We can see the fork call spawning PID 32.

```
TRACE: pid = 31 | command_name = echo | syscall = exec | return_value = 0
TRACE: pid = 31 | command_name = echo | syscall = write | return_value = 1
TRACE: pid = 31 | command_name = echo | syscall = write | return_value = 1
TRACE: pid = 31 | command_name = echo | syscall = write | return_value = 1
TRACE: pid = 31 | command_name = echo | syscall = write | return_value = 1
TRACE: pid = 31 | command_name = echo | syscall = write | return_value = 1
TRACE: pid = 31 | command_name = echo | syscall = write | return_value = 1
TRACE: pid = 30 | command_name = traceChild | syscall = wait | return_value = 31
TRACE: pid = 30 | command_name = traceChild | syscall = fork | return_value = 32
TRACE: pid = 32 | command_name = traceChild | syscall = getpid | return_value = 32
TRACE: pid = 32 | command_name = traceChild | syscall = write | return_value = 1
```

We can see the fork call spawning PID 33.

```
TRACE: pid = 32 | command_name = echo | syscall = write | return_value = 1
TRACE: pid = 30 | command_name = traceChild | syscall = wait | return_value = 32
TRACE: pid = 30 | command_name = traceChild | syscall = fork | return_value = 33
TRACE: pid = 33 | command_name = traceChild | syscall = getpid | return_value = 33
TRACE: pid = 33 | command_name = traceChild | syscall = write | return_value = 1
```

Then the final print message is displayed.

```
TRACE: pid = 33 | command_name = echo | syscall = write | return_value = 1
TRACE: pid = 30 | command_name = traceChild | syscall = wait | return_value = 33
TRACE: pid = 30 | command_name = traceChild | syscall = write | return_value = 1
TRACE: pid = 30 | command_name = traceChild | syscall = write | return_value = 1
TRACE: pid = 30 | command_name = traceChild | syscall = write | return_value = 1
```

Task 4.2.6 (Formatting Readable Output) -

We made our command outputs more readable by suppressing the command outputs. We implemented this by intercepting the write system call in syscall.c. When strace is enabled (strace_enabled flag in struct proc), the syscall() function checks for the write system call and the file descriptor being stdout (1) or stderr (2). If these conditions are met, it skips the actual write and the system only logs the write syscall details (trace) (e.g., pid, name, syscall, and return_value). In the case of an error, only the exec system call prints its output.

Task 4.3 (Implementing options in xv6) -

The implementation of the -e, -f, -s, and -c options for strace in xv6 involved several nuanced changes to both the kernel and user-space code, as these options required dynamic filtering and tracking of system call activity..

For the -e option, which filters logs to display only specific system calls, the approach began by introducing a mechanism in the process structure (struct proc) to store the desired system call name and activate a corresponding filter flag. This required defining a field in struct proc, such as char syscall_filter[16] for the system call name and int filter_enabled to indicate whether filtering was active. A user-space call like strace -e <syscall> sets these fields using a new handleflags system call, passing the name of the desired system call. In the kernel, the syscall() function was modified to check whether filter_enabled was set for the calling process and, if so, compare the name of each invoked system call against the stored syscall_filter. If the names did not match, the system call proceeded without logging. For efficiency, the comparison leveraged a string match operation with short-circuit logic to minimize overhead. After the targeted command is completed, a resetflags system call clears the filter, resetting the process state for subsequent commands.

For the -f option, which logs only failed system calls, the implementation leveraged the return values of system calls to determine failure. In xv6, a negative return value typically indicates an error. To implement this, an additional flag, such as int log_failures, was added to the process structure. When the user executed strace -f, the log_failures flag was set using the handleflags system call. Inside the syscall() function, the tracing logic was augmented to check the return value of each system call. If the return value was negative and log_failures was active, the system call was logged using the existing tracing infrastructure. This selective logging required careful integration with the add_trace_event() function to ensure that failed calls were captured while maintaining normal execution flow for successful calls. As with -e, a resetflags system call cleared the flag after the command was completed, ensuring subsequent commands were traced without filtering.

The -s option, which logs only successful system calls, mirrored the approach used for -f but focused on positive or zero return values. A separate flag, such as int log_successes, was added to the process structure and set via handleflags when strace -s was invoked. In the kernel, the syscall() function was modified to evaluate the return value of each system call. If the return value was greater than or equal to zero and log_successes was active, the system call was logged. Similar to the other options, the flag was reset after the command was completed, maintaining consistent behaviour for subsequent commands.

The -c option was designed to provide a summary of system call activity. To achieve this, additional counters were added to the kernel to record the frequency of each system call. These counters were implemented as an array, indexed by the system call number, and updated during each system call invocation in syscall(). When strace -c was executed, the counters were initialized for the current process, and the system call

handling logic incremented the relevant counter for each call made by the process during the traced command execution. After the command finished, a system call or user-space function was used to retrieve the counters and display a formatted summary, showing the count of each system call invoked. This required extending the `strace_dump` functionality to include the new counter data in its output.

Although we knew our theory was correct, we were unable to implement these features. We tried debugging through our approach, but we did not produce a working code.

Task 4.4 (Write output of strace to file) -

We implemented the functionality to redirect `strace` output to a file using the `-o` `<filename>` option in `strace.c`. When the user runs `strace -o <filename>`, the program opens the specified file in write mode (using `open` with `O_WRONLY | O_CREATE`) and redirects `stdout` to the file by closing the default `stdout` (file descriptor 1) and duplicating the file descriptor for the opened file using `dup(fd)`. The `strace_dump()` system call is then invoked to dump the trace logs into the redirected `stdout`. So we can see the latest N events in that output file. Finally, the program closes the file descriptor after completing the dump.

```
} else if (strcmp(argv[1], "-o") == 0) {
    if (argc < 3) {
        printf(2, "Usage: strace -o <filename>\n");
        exit();
    }

    // Open the specified file for appending
    int fd = open(argv[2], O_WRONLY | O_CREATE);
    if (fd < 0) {
        printf(2, "Error: Cannot open file %s\n", argv[2]);
        exit();
    }

    // Redirect stdout to the file
    close(1); // Close the current stdout
    dup(fd);  // Duplicate fd so it becomes the new stdout
    close(fd); // Close the original fd

    // Dump trace logs to the file
    strace_dump();

    printf(2, "Trace logs appended to %s\n", argv[2]);
    exit();
}
```

The following screenshot shows a successful output redirection to a file. The strace dump events are successfully logged in the file as well.

```
$ strace on
$ echo hi
TRACE: pid = 4 | command_name = sh | syscall = sbrk | return_value = 16384
TRACE: pid = 4 | command_name = echo | syscall = exec | return_value = 0
TRACE: 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 = write | return_value = 1
$ echo hey
TRACE: pid = 5 | command_name = sh | syscall = sbrk | return_value = 16384
TRACE: pid = 5 | command_name = echo | syscall = exec | return_value = 0
TRACE: pid = 5 | command_name = echo | syscall = write | return_value = 1
TRACE: pid = 5 | command_name = echo | syscall = write | return_value = 1
TRACE: pid = 5 | command_name = echo | syscall = write | return_value = 1
TRACE: pid = 5 | command_name = echo | syscall = write | return_value = 1
$ strace off
TRACE: pid = 6 | command_name = sh | syscall = sbrk | return_value = 16384
TRACE: pid = 6 | command_name = strace | syscall = exec | return_value = 0
$ strace dump
TRACE EVENT: pid = 4 | command_name = echo | syscall = write | return_value = 1
TRACE EVENT: pid = 4 | command_name = echo | syscall = write | return_value = 1
TRACE EVENT: pid = 5 | command_name = sh | syscall = sbrk | return_value = 16384
TRACE EVENT: pid = 5 | command_name = echo | syscall = exec | return_value = 0
TRACE EVENT: pid = 5 | command_name = echo | syscall = write | return_value = 1
TRACE EVENT: pid = 5 | command_name = echo | syscall = write | return_value = 1
TRACE EVENT: pid = 5 | command_name = echo | syscall = write | return_value = 1
TRACE EVENT: pid = 5 | command_name = echo | syscall = write | return_value = 1
TRACE EVENT: pid = 6 | command_name = sh | syscall = sbrk | return_value = 16384
TRACE EVENT: pid = 6 | command_name = strace | syscall = exec | return_value = 0
$ strace -o file.txt
Trace logs appended to file.txt
$ cat file.txt
TRACE EVENT: pid = 4 | command_name = echo | syscall = write | return_value = 1
TRACE EVENT: pid = 4 | command_name = echo | syscall = write | return_value = 1
TRACE EVENT: pid = 5 | command_name = sh | syscall = sbrk | return_value = 16384
TRACE EVENT: pid = 5 | command_name = echo | syscall = exec | return_value = 0
TRACE EVENT: pid = 5 | command_name = echo | syscall = write | return_value = 1
TRACE EVENT: pid = 5 | command_name = echo | syscall = write | return_value = 1
TRACE EVENT: pid = 5 | command_name = echo | syscall = write | return_value = 1
TRACE EVENT: pid = 5 | command_name = echo | syscall = write | return_value = 1
TRACE EVENT: pid = 6 | command_name = sh | syscall = sbrk | return_value = 16384
TRACE EVENT: pid = 6 | command_name = strace | syscall = exec | return_value = 0
$
```

Task 4.5 (Application of strace) -

To demonstrate strace, we created a simple program which tries to open a non-existent file. When we run that program, it first searches for a file named “nonexistent.txt” to open it for reading. Since that file is not present in the system, an error message is printed. The following screenshot shows the implementation of this program in xv6.

```
int main() {
    int fd;

    // Attempt to open a non-existent file for reading
    fd = open("nonexistent.txt", 0_RDONLY);
    if (fd < 0) {
        printf(1, "Error: Could not open nonexistent.txt for reading\n");
    }

    exit();
}
```

We get the following trace when we run this with strace enabled. The program attempts to open a non-existent file (nonexistent.txt) in read-only mode using the open system call. Since the file does not exist, the open call returns -1, indicating failure. The program logs an error message using printf before exiting indicated by the subsequent write system calls. If we did not know the program beforehand, an open syscall returning -1 suggests the program attempted to open a file that either doesn't exist or lacks proper permissions. The subsequent write calls suggest that an error message is printed.

```
$ strace on
$ straceReadxv6
TRACE: pid = 28 | command_name = sh | syscall = sbrk | return_value = 16384
TRACE: pid = 28 | command_name = straceReadxv6 | syscall = exec | return_value = 0
TRACE: pid = 28 | command_name = straceReadxv6 | syscall = open | return_value = -1
TRACE: pid = 28 | command_name = straceReadxv6 | syscall = write | return_value = 1
TRACE: pid = 28 | command_name = straceReadxv6 | syscall = write | return_value = 1
TRACE: pid = 28 | command_name = straceReadxv6 | syscall = write | return_value = 1
TRACE: pid = 28 | command_name = straceReadxv6 | syscall = write | return_value = 1
TRACE: pid = 28 | command_name = straceReadxv6 | syscall = write | return_value = 1
TRACE: pid = 28 | command_name = straceReadxv6 | syscall = write | return_value = 1
TRACE: pid = 28 | command_name = straceReadxv6 | syscall = write | return_value = 1
```

We implemented the same program for Linux as follows.

```
int main() {
    int fd;

    // Attempt to open a non-existent file for reading
    fd = open("nonexistent.txt", O_RDONLY);
    if (fd < 0) {
        perror("Error opening nonexistent.txt");
    }

    return 0;
}
```

The strace logs are as follows.

