CS3500: Operating Systems
Lab 2

Department of Computer Science and Engineering
Indian Institute of Technology Madras

Problem 1: System Call Tracing in `xv6`

Background

In operating systems, debugging kernel-level issues can be challenging. A system call tracer lets you monitor the calls a process makes to the kernel, providing insight into program behavior and helping to detect bugs or inefficiencies.

In UNIX-like systems, tools like `strace` serve this role. In this assignment, you will build a simplified tracer for `xv6-riscv`.

Suggested Reading:

- xv6 Book, Chapter 2: *Traps, interrupts, and system calls*
- xv6 Source Code: `kernel/syscall.c`, `kernel/sysproc.c`

Objective

Extend the xv6 kernel to add a new system call:

```
int trace(int mask);
```

The `mask` argument is a bitmask specifying which system calls to trace. Each bit corresponds to a system call number (defined in `kernel/syscall.h`).

Detailed Requirements

When tracing is enabled for a process:

- Before returning from a traced system call, print:

```
<pid>:  syscall <name> -> <return value>
```

- Propagate the tracing mask to all children created via `fork()`.
- Do not affect unrelated processes.

Testing and Example Output

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ trace 32 grep hello README
3: syscall read -> 1023
3: syscall read -> 965
3: syscall read -> 431
3: syscall read -> 0

$ trace 2147483647 grep hello README
5: syscall trace -> 0
5: syscall exec -> 3
5: syscall open -> 3
5: syscall read -> 1023
5: syscall read -> 965
5: syscall read -> 431
5: syscall read -> 0
5: syscall close -> 0

$ trace 2 usertests forkforkfork
usertests starting
8: syscall fork -> 9
usertrap(): unexpected scause 0xf pid=9
            sepc=0x47d8 stval=0x7ec4fff
test forkforkfork: 8: syscall fork -> 10
10: syscall fork -> 11
11: syscall fork -> 12
11: syscall fork -> 13
.
.
.
13: syscall fork -> 70
11: syscall fork -> -1
12: syscall fork -> -1
13: syscall fork -> -1
OK
8: syscall fork -> 71
usertrap(): unexpected scause 0xf pid=71
            sepc=0x47d8 stval=0x7ec4fff
ALL TESTS PASSED
```

Implementation Steps (Recommended Order)

1. Add `$U/_trace` to the `UPROGS` list in `Makefile`.
2. Add a prototype for `trace` in `user/user.h` and trace.c file to call for trace().
3. Add an entry for `trace` in `user/usys.pl` so that `user/usys.S` is generated with the correct syscall stub.
4. Assign a syscall number for `SYS_trace` in `kernel/syscall.h`.

5. Implement `sys_trace()` in `kernel/sysproc.c`, storing the `mask` in a new field (`trace_mask`) inside `struct proc` (`kernel/proc.h`).

6. Modify `fork()` in `kernel/proc.c` to copy the `trace_mask` to child processes.

7. In `syscall()` (`kernel/syscall.c`):

   - Check if the current syscall is enabled in `trace_mask`.
   - If yes, print PID, syscall name, and return value.

Hints and Common Pitfalls

- Use the existing `syscalls[]` array for syscall names.
- Remember that `trace` should affect the calling process and its descendants only.
- Avoid printing arguments — the assignment only requires syscall name, PID, and return value.
- trace.c doesn't exist!!

Grading Scheme for Output Evaluation

The following points to be checked while evaluating the output of the given and hidden test cases:

- **Correct format:** Output lines must follow the pattern `<pid>: syscall <name> -> <value>`.
- **Syscall names:** Names must correspond exactly to the syscalls enabled by the given mask.
- **Order:** System calls should be listed in the same sequence they are executed, with the final call returning value `0`.
- **No extra syscalls:** Output must not include unrelated system calls beyond those enabled by the mask.

Aspects that can differ:

- **PID variability:** The PID may differ between runs and does not need to match the sample output exactly.
- **Return value flexibility:** Return values for `read` (or other traced syscalls) may vary depending on file contents and buffering, but must be positive integers ending with a final value of `0` (EOF).

Problem 2: Kernel Backtrace in **xv6**

Background

A *backtrace* shows the function call chain at a specific point in execution. In debugging, it is critical to know *how* you reached a certain point in code.

In RISC-V, the `s0` register (frame pointer) links stack frames together, allowing traversal of the call chain.

Suggested Reading:

- xv6 Book, Chapter 3: *Page tables, traps, and context switching*
- RISC-V Stack Frame Layout: Calling convention spec

Objective

Implement:

```
void backtrace(void); // in kernel/printf.c:
```

which prints all return addresses from the current function up the call chain, stopping at the top of the current kernel stack.

Implementation Notes

- Read the frame pointer: In `kernel/riscv.h`, inside the section:

  ```
  #ifndef __ASSEMBLER__
  ...
  #endif
  ```

  add:

  ```
  static inline uint64
  r_fp()
  {
    uint64 x;
    asm volatile("mv␣%0,␣s0" : "=r" (x) );
    return x;
  }
  ```

  This uses inline assembly to copy the `s0` register into a C variable.

- Each frame:
  - Return address at `fp - 8`
  - Previous frame pointer at `fp - 16`
- Implement backtrace() inside kernel/printf.c Stop when the frame pointer moves outside the current kernel stack page (`PGROUNDDOWN(fp)` check).
- Add a prototype for `backtrace()` in `kernel/defs.h`.
- Call `backtrace()` inside:

  - `sys_sleep()` (to test with `bttest`).
  - `panic()` in `kernel/printf.c` (to test during kernel panic).

Helper Program for Testing

Place this in `user/bttest.c` and add `$U/_bttest` to `UPROGS` in the `Makefile`.

```
// user/bttest.c
#include "kernel/types.h"
#include "user/user.h"

int
main(int argc, char *argv[])
{
  // Call sleep(1) so sys_sleep() runs in kernel and triggers
     backtrace()
  sleep(1);
  printf("bttest:␣returned␣from␣sleep\n");
  exit(0);
}
```

Testing and Example Output

Backtrace from `sys_sleep()` using `bttest`

Inside the xv6 shell:

```
$ bttest
backtrace:
0x8000299a
0x80002856
0x800025e2
bttest: returned from sleep
```

To translate these addresses to function names and source lines, exit QEMU(CTRL+a , then x) and run on the same docker container:

```
$ riscv64-unknown-elf-addr2line -f -C -e kernel/kernel
# paste address one by one which bttest have given
0x8000299a
sys_sleep
/home/os-iitm/xv6-riscv/kernel/sysproc.c:69
0x80002856
syscall
/home/os-iitm/xv6-riscv/kernel/syscall.c:171
0x800025e2
usertrap
/home/os-iitm/xv6-riscv/kernel/trap.c:81
[CTRL+D]
# Names should be same
```

Grading Scheme for Output Evaluation

The following points to be checked while evaluating the output of the given and hidden test cases:

- When each printed address is passed to `addr2line` with the appropriate kernel binary:
  - The **function names** must exactly match the expected ones in the reference sequence.
  - The **file names** (after the `kernel/` directory) must exactly match the reference.
  - The **order** of the resolved functions and files must match the reference backtrace sequence.
- The following variations are acceptable and will not be penalized:
  - Memory address values (`0x8000xxxx`) may differ between runs or builds.
  - Source code line numbers may differ due to formatting or comment changes.
  - Absolute file paths may vary; only the filename after `kernel/` is required to match.

Submission

You must submit:

- xv6-riscv complete zip folder. We will test your implementation with hidden test cases.
- A **detailed report** explaining:
  - Which files you modified.
  - What changes you made.
  - How your code works internally.
- Screenshots showing the **successful execution** of all the above test commands with the required output format.