OS Lab – 2

Madhav Bhardwaj (CS23B035)

August 2025

Problem 1

Files modified and changes made

- First of all, the trace is a system call and we are also testing it by calling this system call from `trace.c` file. It is not a user program by itself but a system call.
- Since we are adding `trace.c` for testing, we need to add it in `UPROGS` to build it. Therefore after building it, we can just call `trace` with suitable command line arguments.
- We have defined `int trace(int)` in the `user/user.h`. This trace will call the syscall trace internally. This is just telling the compiler there is an implementation of trace to link with somewhere and can let other user programs call this `trace(mask)`.

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

- Entry in `user/usys.pl` to generate the `usys.S` file, inside the `user/usys.pl` we have

  ```
  entry("trace");
  ```

  It expands to a small assembly code in `usys.S` file which has the system call stub (performs the ecall). When this ecall is called the traphandler is invoked which is in kernel/trap.c. In that based on the type of trap it is (here it is a systemcall), control is transfered to kernel/syscall.c. There, based on the syscall number, suitable function is called (in case of trace, it will be sys_trace).

- Every system call has a number so assign a system call number for `trace` in `kernel/syscall.h`.
- Now for the implementation part of the system call, it is in `kernel/sysproc.c`. we write:

  ```
  uint64
  sys_trace(void)
  {
    int mask;
    if(argint(0, &mask) < 0)
      return -1;
    myproc()->trace_mask = mask;
  ```

```
  return 0;
}
```

therefore when sys_trace is called, the mask is set in the process information struct `proc`.

- Modify `fork()` in `kernel/proc.c` here we need to copy the trace mask to forked processes so that they are also traced.

- Finally inside the `syscall` function in `syscall.c` the following changes are made just to check if the mask for the current system call is set and if yes print a msg.

```
if(p->trace_mask & (1 << p->trapframe->a7)) {
  printf("%d: syscall %s -> %lu\n",
         p->pid,
         syscalls[p->trapframe->a7].name,
         p->trapframe->a0);
}
```

- Then the `trace(mask)` returns to the main in trace.c and then the actual program based on the commandline arguments is executed where the systemcalls will be tracked.

Problem 2

- Modified `kernel/riscv.h`:
  - Added a static inline function `r_fp()` which uses inline assembly to read the frame pointer (the `s0` register) and return it as a `uint64`. it is placed in kernel/riscv.h because it contains all the architecture level helper functions that can read the registers directly.
  - Code:

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

- Modified `kernel/printf.c`:
  - Implemented the `backtrace()` function.
  - The function begins by reading the current frame pointer (`fp = r_fp()`).
  - The base of the stack page is computed as `fp_base = PGROUNDDOWN(fp)`.
  - For each frame:
    * The return address is stored at `fp - 8` and is printed using `printf`.
    * The previous frame pointer is stored at `fp - 16`.

* If the previous frame pointer goes outside the current stack page (`fp_base` to `fp_base + PGSIZE`), the loop terminates.
    * Otherwise, `fp` is updated to the previous frame pointer, continuing the traversal.
  - Code:

```
void
backtrace(void)
{
  uint64 fp;
  fp = r_fp();
  uint64 fp_base = PGROUNDDOWN(fp);
  while(fp >= fp_base && fp >= 0 && PGROUNDDOWN(fp) == fp_base)
  {
    printf("%p\n", (void*)*(uint64*)(fp - 8));
    if (*(uint64*)(fp - 16) >= fp_base + PGSIZE ||
        (*(uint64*)(fp - 16) < fp_base)) break;
    fp = *(uint64*)(fp - 16);
  }
}
```

- Modified `kernel/defs.h`:
  - Added a prototype for the `backtrace()` function so it can be called from other files.
- Usage of `backtrace()`:
  - Called inside `sys_sleep()` to test with `bttest`.
  - Called inside `panic()` in `kernel/printf.c` to print stack traces when the kernel panics.
- Internal Working:
  - The function works by traversing the list of stack frames maintained using the `s0` register (frame pointer).
  - Each frame stores the caller's frame pointer and return address.
  - By following the chain of previous frame pointers, the function can unwind the stack and print the sequence of return addresses that led to the current function.