[xv6] PROJECT 2 – System calls

University of Science - VNUHCM

- **Date**: 15/02/2025
- **Subject**: *Operating System*
- **Class**: *23CLC03*
- **Environment**: *Ubuntu-24.04*
- **Intructors:**
  - *Phan Quốc Kỳ*
  - *Lê Hà Minh*
  - *Lê Giang Thanh*

I. Group Information

No	Student's ID	Student's Full Name	Contribution
1	23127004	Lê Nhật Khôi	`33%`: Using gdb
2	23127113	Nguyễn Trần Phú Quý	`33%`: Sysinfo
3	23127165	Nguyễn Hải Đăng	`33%`: System call tracing

II. Submission’s Details

Here are all of the assignments stated in the lab.

1. Using gdb

Setting up gdb

In the Visual Studio Code window, we first create two new terminal for setting up GDB

On the first terminal window, run `make qemu-gdb` to start debugging:

```
*** Now run 'gdb' in another window.
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -global virtio-mmio.force-legacy=fals
e -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0 -S -gdb tcp::26000
```

We can see that the port where QEMU or the remote debugger is listening is 26000

On the second terminal window, run `gdb-multiarch -x ./.gdbinit`:

```
xv6@DESKTOP-20GGV0D:~/khoi_xv6/xv6-labs-2024$ gdb-multiarch -x ./.gdbinit
GNU gdb (Ubuntu 15.0.50.20240403-0ubuntu1) 15.0.50.20240403-git
```

To connect to a program running on QEMU or a remote debugger via port 26000, run `target remote localhost: 26000`:

```
(gdb) target remote localhost: 26000
Remote debugging using localhost: 26000
warning: No executable has been specified and target does not support
determining executable automatically.  Try using the "file" command.
0x0000000000001000 in ?? ()
=> 0x0000000000001000:  00000297        auipc   t0,0x0
```

Next, to debug how system calls are handled in the kernel, we type in the (gdb) window `file kernel/kernel`, `b syscall` and `c`, respectively:

```
(gdb) file kernel/kernel
Reading symbols from kernel/kernel...
(gdb) b syscall
Breakpoint 1 at 0x80001c82: file kernel/syscall.c, line 133.
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at kernel/syscall.c:133
133     {
(gdb)
```

To show the better view of source code and the call stack, we run `layout src` and `backtrace`, this splits the window in two as the image below:

```
-kernel/syscall.c-
     131 void
     132 syscall(void)
B+>  133 {
     134   int num;
     135   struct proc *p = myproc();
     136
     137   num = p->trapframe->a7;
     138   if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
     139     // Use num to lookup the system call function for num, call it,

remote Thread 1.1 (src) In: syscall                          L133   PC: 0x80001c82
(gdb) backtrace
#0  syscall () at kernel/syscall.c:133
#1  0x0000000080001a3e in usertrap () at kernel/trap.c:67
#2  0x0505050505050505 in ?? ()
(gdb)
```

Answer to the questions in CQ_Lab02

At the backtrace output, which function called syscall?

> From the backtrace output, ***usertrap() function*** called the syscall

```
(gdb) backtrace
#0  syscall () at kernel/syscall.c:133
#1  0x0000000080001a3e in usertrap () at kernel/trap.c:67
#2  0x0505050505050505 in ?? ()
```

What is the value of `p->trapframe->a7` and what does that value represent?

By typing `n` (new) a few times, the breakpoint now past `struct proc *p = myproc()`;

```
kernel/syscall.c
    void
    syscall(void)
B+  {
      int num;
      struct proc *p = myproc();

 >    num = p->trapframe->a7;
      if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
        // Use num to lookup the system call function for num, call it,
remote Thread 1.3 (src) In: syscall                                          L137   PC: 0x80001c94
$1 = {lock = {locked = 0x0, name = 0x800071b8, cpu = 0x0}, state = 0x4, chan = 0x0, killed = 0x0, xstate = 0x0, pid = 0x1,
  parent = 0x0, kstack = 0x3fffffd000, sz = 0x5000, pagetable = 0x87f4e000, trapframe = 0x87f56000, context = {ra = 0x800012be,
    sp = 0x3fffffda80, s0 = 0x3fffffdab0, s1 = 0x8000a660, s2 = 0x8000a230, s3 = 0x1, s4 = 0x800127a8, s5 = 0x3,
    s6 = 0x8001b300, s7 = 0x0, s8 = 0x8001b428, s9 = 0x16e, s10 = 0xb0, s11 = 0x2}, ofile = {0x0 <repeats 16 times>},
  cwd = 0x80018770, name = {0x69, 0x6e, 0x69, 0x74, 0x0, 0x0, 0x64, 0x65, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}}
(gdb)
```

Then run `p /x p->trapframe-a7` to determine the value of p->trapframe->a7:

```
(gdb) p /x p->trapframe->a7
$2 = 0xf
```

> The value of p->trapframe->a7 is ***0xf(hex) or 15(decimal)***. This value is represented for ***SYS_open***.

What was the previous mode that the CPU was in?

The processor is running in kernel mode, and we can print privileged registers such as sstatus by using `p /x $sstatus`:

```
(gdb) p /x $sstatus
$8 = 0x200000022
```

To determine the SPP bit, run `print ($sstatus >> 8) & 1`:

```
(gdb) print ($sstatus >> 8) & 1
$9 = 0
```

> The result turns out to be 0 which corresponding to ***User Mode***

Write down the assembly instruction the kernel is panicing at. Which register corresponds to the variable num?

In this subsequent part of the lab, we replace this line `num = p->trapframe->a7;` with `num = *(int *) 0;` in the `syscall.c` file to stimulate programming error that causes xv6 kernel to panic

Running QEMU again, we get the error message about sepc which holds the address of the instruction that caused the panic:

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
scause=0xd sepc=0x80001c92 stval=0x0
panic: kerneltrap
```

Then, we need to locate the instruction causing the panic by searching `sepc=0x80001c92` in the `kernel/kernel.asm` file:

```
  num = *(int *) 0;
    80001c92: 00002683           lw   a3,0(zero) # 0 <_entry-0x80000000>
```

> In conclusion, ***the kernel panicked at*** ***`0x80001c92`*** and the instruction that caused the fault is `lw a3, 0(zero)`, meaning ***num corresponds to register*** ***`a3`***.

Why does the kernel crash?

To inspect the state of the processor and the kernel at the faulting instruction, fire up gdb, and set a breakpoint at the faulting sepc by running `b *0x80001c92` and `c`:

```
(gdb) b *0x80001c92
Breakpoint 1 at 0x80001c92: file kernel/syscall.c, line 138.
(gdb) c
Continuing.
[Switching to Thread 1.3]

Thread 3 hit Breakpoint 1, syscall () at kernel/syscall.c:138
138       num = *(int *) 0;
```

```
   0x80001c8a <syscall+8>  addi    s0,sp,32
   0x80001c8c <syscall+10> jal     0x80000d66 <myproc>
   0x80001c90 <syscall+14> mv      s1,a0
B+>0x80001c92 <syscall+16> lw      a3,0(zero) # 0x0
   0x80001c96 <syscall+20> addiw   a4,a3,-1
   0x80001c9a <syscall+24> li      a5,20
   0x80001c9c <syscall+26> bltu    a5,a4,0x80001cba <syscall+56>
   0x80001ca0 <syscall+30> slli    a4,a3,0x3
   0x80001ca4 <syscall+34> auipc   a5,0x6
remote Thread 1.3 (asm) In: syscall                          L138  PC: 0x80001c92
(gdb) p $scause
$1 = 8
```

> In the `syscall()` function, the program attempted to access `address 0x0` using the statement `num = *(int *)0`. However, ***in the kernel's address space,*** ***`address 0x0`*** ***is not mapped***, so when the CPU executes the instruction `lw a3, 0(zero)`, it causes a memory access fault. And when the kernel detects an invalid memory access, it ***cannot continue execution and must halt, leading to a panic state***.

> ***The value of the*** ***`scause register is 8`***, indicating that the fault was caused by a system call from user mode

What is the name of the binary that was running when the kernel paniced? What is its process id (pid)?

To find out which user process was running when the kernel paniced, you can print out the process's name by running `p p->name`:

```
(gdb) p p->name
$2 = "initcode\000\000\000\000\000\000\000"
```

Or running `p p->id` to get its process id:

```
(gdb) p p->pid
$3 = 1
```

> The process that caused the kernel panic ***has PID = 1***, and the ***binary running is "initcode"***.

2. System call tracing (moderate)

> There are two phases: basic tracing functionality and advanced argument printing (challenge). The goal was to provide a tool for monitoring system call execution, including arguments and return values, to aid in debugging and understanding program behavior.

Phase 1: Basic Tracing

Goals

Implement a basic tracing system that allows selectively enabling or disabling tracing for specific system calls using a bitmask. Each bit in the mask corresponds to a system call number.

Implementation

The following xv6 files were modified:

- **`proc.h`**: Added a `trace_mask` field (an integer) to the `struct proc` to store the bitmask for each process.
- **`user/user.h` and `user/usys.pl`**: Declared the `trace` system call, allowing user programs to set the trace mask.
- **`kernel/syscall.h`**: Defined a new system call number for the `trace` system call (e.g., `SYS_trace`).
- **`kernel/proc.c`**: Modified `fork()` to ensure that child processes inherit the `trace_mask` from their parent.
- **`kernel/syscall.c`**: Modified the `syscall()` function to check the `trace_mask` of the current process. If the relevant bit is set, a trace message (PID and system call number) is printed. The `trace` system call's implementation (`sys_trace`) sets the `trace_mask` for the current process.

Testing

Tested with commands like `trace 32 grep hello README`. This sets the trace mask to trace system call number 5 (assuming `read` is syscall number 5) and shows trace output when `grep` executes `read`.

Phase 2: Advanced Argument Printing

Goals

Extend the tracing mechanism to display arguments and return values of traced system calls.

Key Challenges

- **Handling diverse argument types:** System calls use integers, addresses, and strings.
- **Safely retrieving arguments from user-space memory:** Avoiding kernel crashes due to invalid user-space pointers.
- **System call-specific processing:** Each system call has a unique argument signature.
- **Timing:** Arguments must be retrieved *before* execution, and the return value *after*.
- **Formatting:** Appropriate formatting for different argument types.

Implementation

1. **Additional System Call:**

- A new system call, `setargs`, was added to toggle argument printing (enabled/disabled).
- A global variable, `print_args`, in `kernel/syscall.c` controls whether argument printing is enabled.

2. **Argument Retrieval and Formatting:**

- The `syscall()` function in `kernel/syscall.c` was extended.
- A `switch` statement handles the specific arguments of each system call *before* execution.
- Helper functions (`argint`, `argaddr`, `argstr`) are used to retrieve arguments of different types. `argstr` uses a safe string copy mechanism (like a simplified `copyinstr`) to handle potential page faults and invalid user-space addresses. It would allocate a buffer in kernel space.
- Arguments are printed **before** the system call.
- The return value is printed **after** the system call executes.

3. **Error Handling:** Argument retrieval functions are designed to handle invalid user-space addresses, returning error codes to prevent kernel crashes.

Design Decisions

- `print_args` **(Global Variable):** Simple and efficient way to enable/disable argument printing.
- `switch` **Statement:** Clear and organized handling of different system call argument sets.
- **Helper Functions (** `argint` **,** `argaddr` **,** `argstr` **):** Encapsulate argument retrieval and error handling.

Testing

Thoroughly tested with various commands and scenarios:

- Different system calls (e.g., `open` , `read` , `write` , `exec` , `wait` , `exit`).
- Commands with various arguments.
- Error conditions (invalid file paths, invalid memory addresses).
- Nested system calls.

Example test and output:

```
setargs 1       # Enable argument printing
trace 22 grep hello README    # Trace syscall number 22.
4: syscall open(path="README", oflag=0) -> 3   # Example
4: syscall read(fd=3, buf=0x1000, n=1024) -> 1024
...
setargs 0 #disable
```

3. Sysinfo (with load average)

> This system call will report:
>
> - *The number of free memory pages*
> - *The number of processes currently running*
>
> CHALLENGE: *Compute the load average and export it through sysinfo.*

Implementing standard sysinfo

1. Define the `sysinfo` **system call**

- **Edit** `kernel/sysinfo.h` → Define a struct to hold system information.

2. Implement the `sysinfo` **system call**

- **Add `kfreemem()` to `kernel/kalloc.c`** → Collect free memory's size.
- **Add `countproc()` to `kernel/proc.c`** → Collect number of processes.
- **Edit `kernel/sysproc.c`** → Add the *actual function* that collects system info and fills the sysinfo struct using the above functions.

3. Modify `syscall.c` and `syscall.h` to register the new system call

- **Edit `kernel/syscall.c`** → Map `sysinfo` to the syscall function table.
- **Edit `kernel/syscall.h`** → Assign an available syscall number to `sysinfo`.

4. Update the user-space interface

- **Edit `user/user.h`** → Declare `sysinfo()` so user programs can call it.

Testing sysinfo

Successfully passed `sysinfotest`:

```
$ sysinfotest
sysinfotest: start
sysinfotest: OK
```

[Challenge] Exporting the Load average

The *load average* is computed by tracking the number of **runnable** and **running** processes at regular intervals. A history of these values is maintained, and the average is calculated over a fixed number (100) of past samples. The computed value is then exposed to user space through the `sysinfo` system call.

Key changes

- Extended the `struct sysinfo` to include a load average field.
- Several additions to `kernel/proc.c`:
    - Implemented a circular buffer to store historical data about *runnable/running* processes
    - `update_load()`: update the load history.
    - `get_load_avg()`: calculate the load average from samples.
    - Modified the `scheduler()` to call `update_load()` regularly.

Testing load average

To test this feature, we have written a custom user-level testing program `loadtest.c`. When executed, it shows how the load average responds to changes in the number of active processes.

```
$ loadtest
=== Load Test (`loadtest`) ===
[1] Initial system state:
Load Avg: 0.09 | Processes: 3 | Free Mem: 133251072 bytes
[2] Creating 10 processes...
[3] After load creation:
Load Avg: 0.41 | Processes: 13 | Free Mem: 132841472 bytes
[4] Load stabilizing:
Load Avg: 0.33 | Processes: 13 | Free Mem: 132841472 bytes
[5] Cleaning up processes...
[6] Final system state:
Load Avg: 0.32 | Processes: 3 | Free Mem: 133251072 bytes
=== Test Complete ===
```

Limitation of our average load computing

- **Scheduler-Based Update:** Load average is only updated during scheduling, which may not capture system activity with high precision.
- **Fixed Sample Size:** History buffer is limited to a fixed number of samples, potentially losing long-term trends.
- **Integer Scaling:** Stored as an integer (×100) to avoid floating-point math since `xv6` lacks `float` datatype.
- **Short-Term Focus:** Only recent history is considered, lacking exponential smoothing.
- **No Decay Factor:** Load does not gradually decrease over time.
- **Limited Precision:** Accuracy depends on sampling rate and integer scaling.

> While there are more sophisticated approaches, we believe this is the simplest solution.

III. References

- *Lab Answers from MIT*
- *GeeksforGeeks - System calls*
- *Solution from Western Steamed Buns*
- *xv6 Lab answers from yixuaz*