Lab - System calls 目录

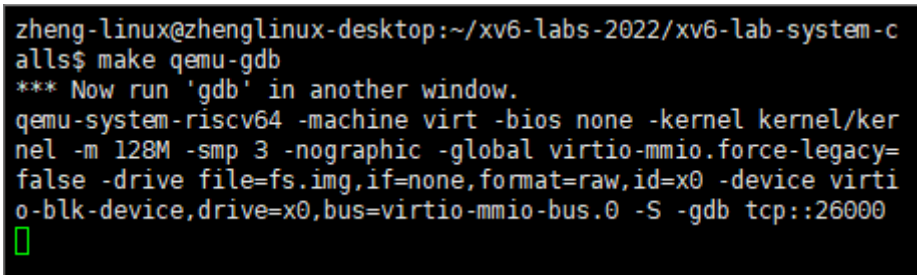
1. Using gdb (easy)

1.1 实验目的

- 学习如何使用gdb进行调试
- 学习如何在内核中进行单步调试和查看堆栈上的变量等操作

1.2 实验步骤

1. 运行`make qemu-gdb`命令启动xv6系统和GDB调试工具，结果如下：



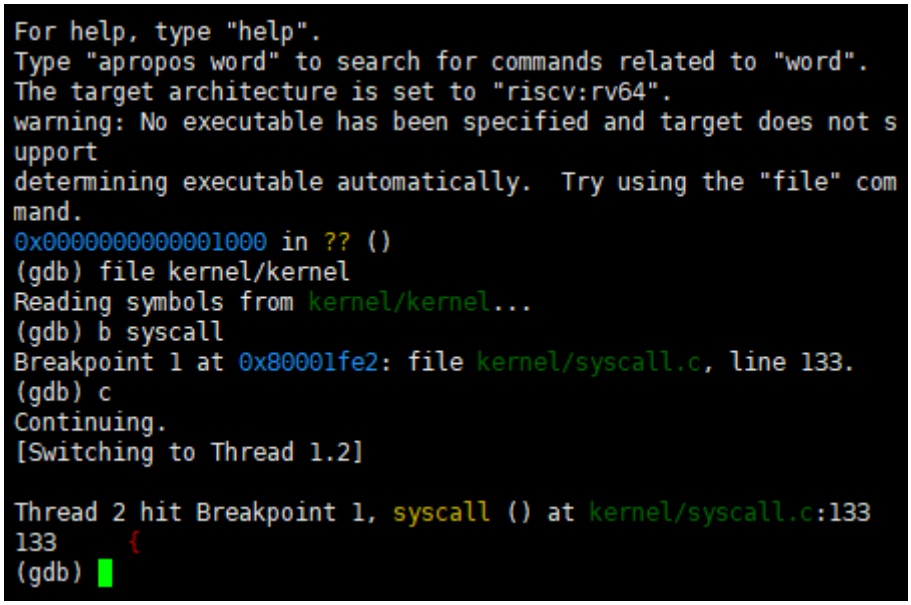
2. 打开另一个窗口，输入以下命令以开启gdb窗口：

```
gdb-multiarch
```

在gdb窗口中，输入以下命令：

```
(gdb) b syscall
Breakpoint 1 at 0x80002142: file kernel/syscall.c, line 243.
(gdb) c
Continuing.
[Switching to Thread 1.2]
```

得到结果：



3. 使用`layout src`命令将窗口分割为两部分，显示GDB在源代码中的位置：

```
kernel/syscall.c
B+>    133  {
       134    int num;
       135    struct proc *p = myproc();
       136
       137    num = p->trapframe->a7;
       138    if(num > 0 && num < NELEM(syscalls) && syscalls[
       139      // Use num to lookup the system call function
       140      // and store its return value in p->trapframe-
       141      p->trapframe->a0 = syscalls[num]();
       142    } else {
       143      printf("%d %s: unknown sys call %d\n",
       144              p->pid, p->name, num);
       145      p->trapframe->a0 = -1;
       146    }
       147  }
       148
       149
       150
       151
       152
       153
       154
remote Thread 1.2 In: syscall                 L133  PC: 0x80001fe2
```

4. 使用`backtrace`命令打印出堆栈回溯信息。

```
(gdb) backtrace
#0  syscall () at kernel/syscall.c:133
#1  0x0000000080001d16 in usertrap () at kernel/trap.c:67
#2  0x0505050505050505 in ?? ()
```

5. 使用`n`命令几次以跳过`struct proc *p = myproc();`的执行。执行完这条语句后，输入`p /x *p`，以十六进制格式打印当前进程的`proc`结构。

```
(gdb) p /x*p
$1 = {lock = {locked = 0x0, name = 0x80008178, cpu = 0x0},
  state = 0x0, chan = 0x0, killed = 0x0, xstate = 0x0,
  pid = 0x0, parent = 0x0, kstack = 0x3ffffc1000, sz = 0x0,
  pagetable = 0x0, trapframe = 0x0, context = {ra = 0x0,
    sp = 0x0, s0 = 0x0, s1 = 0x0, s2 = 0x0, s3 = 0x0,
    s4 = 0x0, s5 = 0x0, s6 = 0x0, s7 = 0x0, s8 = 0x0,
    s9 = 0x0, s10 = 0x0, s11 = 0x0}, ofile = {
    0x0 <repeats 16 times>}, cwd = 0x0, name = {
    0x0 <repeats 16 times>}}
```

6. 同理，进行其他调试操作。

1.3 实验中遇到的问题和解决办法

- **问题：** 出现警告：`warning: File ".../.gdbinit" auto-loading has been declined', edit ~/.gdbinit to add "add-auto-load-safe-path..."`

  解决方法：使用的是不被信任的文件。 在`.config/gdb/gdbinit`中添加命令`set auto-load safe-path /home/zheng-linux/xv6-labs-2022/xv6-lab-system-calls/.gdbinit`，若没有则新建。

- **问题：** 出现警告：

  ```
  warning: No executable has been specified and target does not support
  determining executable automatically.  Try using the "file" command.
  ```

  解决方法：阅读Guidance，在gdb窗口中执行`file kernel/kernel`命令。

- **问题：** 无法找出内核崩溃的原因和位置

解决方法： 查看GDB输出的堆栈回溯信息，寻找内核崩溃时的堆栈帧和函数调用关系，使用`backtrace`命令打印出完整的堆栈回溯信息。查找GDB输出中的错误提示和具体的代码位置，以确定内核崩溃的原因和位置。

1.4 实验心得

在本次实验中，我学会了如何使用GDB调试工具在xv6内核中进行单步调试和查看变量的值，了解了GDB的基本命令和调试技巧，以及如何使用GDB来跟踪和定位内核崩溃的原因和位置。GDB作为一个强大的调试工具，对于定位和解决内核中的问题非常有帮助。在实践中我更好地掌握了内核调试的技能。

2. System call tracing (moderate)

2.1 实验目的

- 实现`trace`系统调用函数
- 熟悉操作系统如何从用户态发送请求调用内核态程序

2.2 实验步骤

1. 阅读`user/trace.c`，了解其逻辑。
2. 在`Makefile`的`UPROGS`中添加`$U/_trace`
3. 在`user/user.h`中添加系统调用的原型

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

4. 在`user/usys.pl`中添加存根

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

该文件会被汇编成`usys.S`

5. 在`kernel/syscall.h`中添加系统调用编号

```
#define SYS_trace 22
```

6. 在`kernel/proc.h`中定义的`proc`结构(用于维护进程状态)中添加`mask`变量用于记录掩码

```
int mask; // Trace mask
```

7. 在`kernel/sysproc.c`中添加从用户态到内核态系统调用的参数传递入口，使用`argint()`函数获取用户态的系统调用命令的参数，将参数存储在`proc`的新变量中

```
uint64 sys_trace(void)
{
    int mask;
    argint(0, &mask);
    myproc()->mask = mask;
    return 0;
}
```

8. 在`fork()`函数中加入代码，使得`fork()`在复制时同时把掩码进行复制传递，父进程将自己的mask复制给子进程，以跟踪子进程的特定系统调用

```
np->mask = p->mask;
```

9. 在`kernel/syscall.c`中新建一个系统调用号到名称的索引，实现在`syscall()`函数中输出`trace`的信息

```
char *syscalls_name[30] = {
    "",     "fork",  "exit",   "wait", "pipe",  "read",  "kill",   "exec",
"fstat", "chdir", "dup",   "getpid",
    "sbrk", "sleep", "uptime", "open", "write", "mknod", "unlink", "link",
"mkdir", "close", "trace", "sysinfo",
};
```

```
extern uint64 sys_trace(void);
extern uint64 sys_sysinfo(void);

// An array mapping syscall numbers from syscall.h
// to the function that handles the system call.
static uint64 (*syscalls[])(void) = {
    [SYS_fork] sys_fork,   [SYS_exit] sys_exit,     [SYS_wait] sys_wait,
[SYS_pipe] sys_pipe,   [SYS_read] sys_read,       [SYS_kill] sys_kill,
    [SYS_exec] sys_exec,   [SYS_fstat] sys_fstat,   [SYS_chdir] sys_chdir,
[SYS_dup] sys_dup,     [SYS_getpid] sys_getpid,   [SYS_sbrk] sys_sbrk,
    [SYS_sleep] sys_sleep, [SYS_uptime] sys_uptime, [SYS_open] sys_open,
[SYS_write] sys_write, [SYS_mknod] sys_mknod,     [SYS_unlink] sys_unlink,
    [SYS_link] sys_link,   [SYS_mkdir] sys_mkdir,   [SYS_close] sys_close,
[SYS_trace] sys_trace, [SYS_sysinfo] sys_sysinfo,
};
```

10. 在`syscall()`函数中实现输出逻辑，需要输出的值有`pid`，系统调用名和返回值，其中返回值存储在`trapframe->a0`中

```
void syscall(void)
{
    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,
        // and store its return value in p->trapframe->a0

        p->trapframe->a0 = syscalls[num]();
        if (p->mask & (1 << num)) {
            printf("%d: syscall %s -> %d\n", p->pid, syscalls_name[num], p-
>trapframe->a0);
        }
    }
    else {
        printf("%d %s: unknown sys call %d\n", p->pid, p->name, num);
        p->trapframe->a0 = -1;
    }
}
```

11. 使用`trace 2147483647 grep hello README`命令运行测试，结果如下：

```
init: starting sh
$ trace 2147483647 grep hello README
3: syscall trace -> 0
3: syscall exec -> 3
3: syscall open -> 3
3: syscall read -> 1023
3: syscall read -> 961
3: syscall read -> 321
3: syscall read -> 0
3: syscall close -> 0
$
```

12. 运行`grade`测试程序，结果如下：

```
== Test trace 32 grep ==
$ make qemu-gdb
trace 32 grep: OK (2.5s)
== Test trace all grep ==
$ make qemu-gdb
trace all grep: OK (0.6s)
== Test trace nothing ==
$ make qemu-gdb
trace nothing: OK (0.9s)
== Test trace children ==
$ make qemu-gdb
trace children: OK (13.1s)
```

2.3 实验中遇到的问题和解决方法

- **问题：** 无法复制父进程的跟踪掩码到子进程

  解决方法： 需要在`fork()`函数中修改代码，使子进程继承父进程的跟踪掩码。

- **问题：** 题目要求不理解、逻辑不通顺

  解决方法： 边读xv6文档中的内容，边读源代码，理解已实现功能的逻辑，同时深刻理解题目要求。

2.4 实验心得

在本次实验当中，我学会了如何添加系统调用跟踪功能，以便在调试后续实验时进行辅助，了解了如何在xv6内核中添加新的系统调用、如何获取系统调用的名称和如何实现跟踪输出。所有的系统调用都首先经过`kernel/syscall.c`的`syscall()`系统调用入口函数，通过此处调用系统调用，转到`syscall()`，后转到`kernel/sysproc.c`的`sys_**()`相关函数。对操作系统的系统调用流程有了深入的了解。

3. Sysinfo (moderate)

3.1 实验目的

- 编写系统调用函数`sysinfo()`，用于收集有关运行系统的信息
- 深入理解如何编写系统调用函数

3.2 实验步骤

1. 在`Makefile`的`UPROGS`中添加`$U/_sysinfotest`。
2. 预先声明`struct sysinfo`

```
struct sysinfo;
int sysinfo(struct sysinfo *);
```

3. 同上个实验，添加 `sysinfo` 系统调用。在 `user/user.h` 中声明 `sysinfo()` 的原型
4. 可以参考 `kernel/sysfile.c` 中的 `sys_fstat()` 和 `kernel/file.c` 中的 `filestat()`，使用 `copyout()` 来实现拷贝操作，把数据从内核栈拷贝到用户栈。其中 `sysinfo` 需要将一个 `struct sysinfo` 拷贝回用户空间。

   `copyout()` 函数如下：

```
// Copy from kernel to user.
// Copy len bytes from src to virtual address dstva in a given page table.
// Return 0 on success, -1 on error.
int
copyout(pagetable_t pagetable, uint64 dstva, char *src, uint64 len)
{
  uint64 n, va0, pa0;

  while(len > 0){
    va0 = PGROUNDDOWN(dstva);
    pa0 = walkaddr(pagetable, va0);
    if(pa0 == 0)
      return -1;
    n = PGSIZE - (dstva - va0);
    if(n > len)
      n = len;
    memmove((void *)(pa0 + (dstva - va0)), src, n);

    len -= n;
    src += n;
    dstva = va0 + PGSIZE;
  }
  return 0;
}
```

5. 编写 `sys_sysinfo()` 函数：

```
uint64 sys_sysinfo(void)
{
    uint64 addr;
    argaddr(0, &addr);
    struct sysinfo info;
    info.freemem = countMEM();
    info.nproc = countProc();

    if (copyout(myproc()->pagetable, addr, (char *)&info, sizeof(info)) < 0)
{
        return -1;
    }
    return 0;
}
```

   其中，`argaddr()` 的作用是将系统调用参数作为指针读入。

6. 为了收集可用内存的数量，在 `kernel/kalloc.c` 中添加一个函数 `countMEM()`：

```
uint64 countMEM()
{
    struct run *p = kmem.freelist;
    uint64 count = 0;
    while (p) {
        count++;
        p = p->next;
    }
    return count * PGSIZE;
}
```

7. 为了收集进程的数量，在`kernel/proc.c`中添加一个函数`countProc()`：

```
uint64 countProc()
{
    uint64 count = 0;
    for (int i = 0; i < NPROC; i++) {
        if (proc[i].state != UNUSED)
            count++;
    }
    return count;
}
```

8. 运行`grade`测试程序，结果如下：

```
== Test sysinfotest == sysinfotest: OK (2.4s)
    (Old xv6.out.sysinfotest failure log removed)
```

3.3 实验中遇到的问题和解决方法

- **问题：** 无法通过测试程序，错误信息如下：

```
== Test sysinfotest == sysinfotest: FAIL (2.3s)
    ...
        init: starting sh
        $ sysinfotest
        sysinfotest: start
        FAIL: free mem 32500 (bytes) instead of 133120000
```

解决方法： `countMEM()`函数返回的值应当是统计出的块数*页表大小。

3.4 实验心得

本次实验中，我学习了如何在内核中实现一个新的系统调用，并将相应的信息填充到对应结构体中，如何将内核空间的数据拷贝到用户空间，并了解了如何获取可用内存的数量和进程的数量。总而言之，通过本次实验，我对系统调用和内核功能有了更深入的理解。