

# Lab2-System-Calls

## Using gdb

### 实验目的

- 学习使用 gdb 调试工具来帮助理解和定位 xv6 内核中的问题和错误
- 理解函数调用关系，使用 gdb 进行内核级别调试

### 实验步骤

#### 1. 启动调试环境：

- 运行 `make qemu-gdb` 启动 qemu 模拟器并启用 gdb。

```
(base) root@876277deb135:~/workspace/personal_data/zwc/Xv6/Lab2-System-Calls# make qemu-gdb
*** 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=false -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0 -S -gdb tcp::25000
```

- 在另一个窗口中运行 `gdb-multiarch` 启动 gdb。

```
(base) root@876277deb135:~/workspace/personal_data/zwc/Xv6/Lab2-System-Calls# gdb-multiarch
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.2) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word".
warning: File "/root/workspace/personal_data/zwc/Xv6/Lab2-System-Calls/.gdbinit" auto-loading has been declined by your `auto-load safe-path'
set to "$debugdir:$datadir/auto-load".
To enable execution of this file add
    add-auto-load-safe-path /root/workspace/personal_data/zwc/Xv6/Lab2-System-Calls/.gdbinit
line to your configuration file "/root/.gdbinit".
To completely disable this security protection add
    set auto-load safe-path /
line to your configuration file "/root/.gdbinit".
For more information about this security protection see the
"Auto-loading safe path" section in the GDB manual.  E.g., run from the shell:
    info "(gdb)Auto-loading safe path"
(gdb) █
```

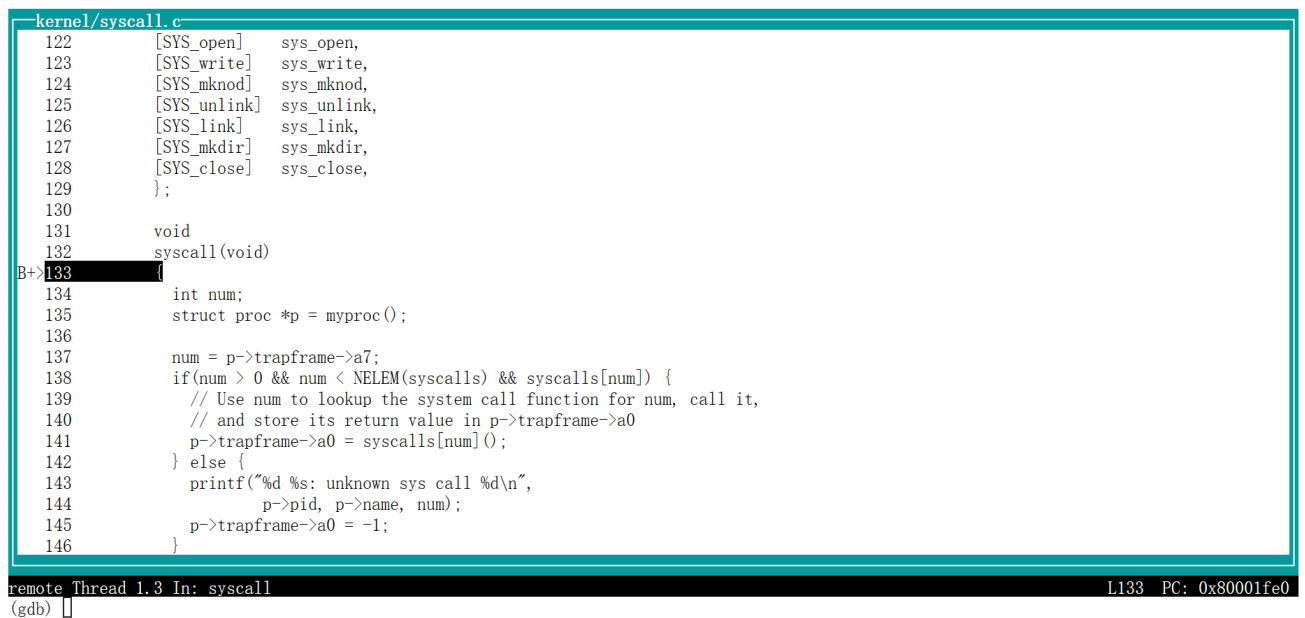
#### 2. 设置断点并继续执行：

- 在 gdb 中设置断点： `(gdb) b syscall` 。
- 继续执行程序： `(gdb) c` 。

```
(gdb) b syscall
Breakpoint 1 at 0x80001fe0: file kernel/syscall.c, line 133.
(gdb) c
Continuing.
[Switching to Thread 1.3]
```

```
Thread 3 hit Breakpoint 1, syscall () at kernel/syscall.c:133
133 {
(gdb) 
```

### 3. 当断点命中时，使用 `layout src` 命令分割窗口以显示源代码



```
kernel/syscall.c
122     [SYS_open]    sys_open,
123     [SYS_write]   sys_write,
124     [SYS_mknod]   sys_mknod,
125     [SYS_unlink]  sys_unlink,
126     [SYS_link]    sys_link,
127     [SYS_mkdir]   sys_mkdir,
128     [SYS_close]   sys_close,
129     };
130
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,
140             // and store its return value in p->trapframe->a0
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         }
}

remote Thread 1.3 In: syscall
(gdb) 
```

### 4. 查看函数调用关系：

- 使用 `backtrace` 命令查看调用堆栈，确定调用 `syscall` 的函数。

```
(gdb) backtrace
#0  syscall () at kernel/syscall.c:133
#1  0x0000000080001d14 in usertrap () at kernel/trap.c:67
#2  0x0505050505050505 in ?? ()
```

### 5. 查看进程结构体：

- 使用 `n` 命令逐步执行，直到通过 `struct proc *p = myproc();`。
- 使用 `p /x *p` 命令打印当前进程的 `proc` 结构体的十六进制表示。

```
(gdb) p /x *p
$1 = {lock = {locked = 0x1, name = 0x80008178, cpu = 0x800089b0}, state = 0x4, chan = 0x0, killed = 0x0, xstate = 0x0, pid = 0x1,
parent = 0x0, kstack = 0x3fffffd000, sz = 0x1000, pagetable = 0x87f73000, trapframe = 0x87f74000, context = {ra = 0x80001466,
sp = 0x3fffffd0, s0 = 0x3fffffd0, s1 = 0x80008d30, s2 = 0x80008900, s3 = 0x1, s4 = 0x0, s5 = 0x3, s6 = 0x800199d0, s7 = 0x8,
s8 = 0x80019af8, s9 = 0x4, s10 = 0x1, s11 = 0x0}, ofile = {0x0 <repeats 16 times>}, cwd = 0x80016e40, name = {0x69, 0x6e, 0x69, 0x74,
0x63, 0x6f, 0x64, 0x65, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}}
```

## 实验中遇到的问题和解决办法

当运行 `gdb-multiarch` 时出现以下警告：

```
warning: File "/root/workspace/personal_data/zwc/Xv6/Lab2-System-Calls/.gdbinit" auto-
loading has been declined by your 'auto-load safe-path' set to "$debugdir:$datadir/auto-
load".
```

这表明 GDB 拒绝自动加载位于 `/root/workspace/personal_data/zwc/Xv6/Lab2-System-Calls/.gdbinit` 的 `.gdbinit` 文件。`.gdbinit` 文件通常包含一些预定义的调试配置和命令。要解决这个问题，需要更新 GDB 的配置以允许加载这个文件。步骤如下：

### 1. 打开或创建 `.gdbinit` 文件：

```
vi ~/.gdbinit
```

### 2. 进入插入模式：

在 `vi` 中，按 `i` 键进入插入模式。

### 3. 添加以下行：

```
add-auto-load-safe-path /root/workspace/personal_data/zwc/Xv6/Lab2-System-
Calls/.gdbinit
```

### 4. 保存并退出：

- 按 `Esc` 键退出插入模式。
- 输入 `:wq` 并按回车键保存并退出。

### 5. 重启 GDB

## 实验心得

在这次实验中，我深刻体会到内核调试工具的重要性。作为计算机科学的学生，过去我们更多的是在用户空间中进行编程和调试，但这次深入到内核层次，让我对操作系统的底层机制有了更直观和深刻的理解。

通过使用 GDB 调试 xv6 内核，我学会了如何设置断点、查看调用堆栈、检查进程结构体，以及分析和解决内核 panic 的问题。实验的每一步都让我感受到调试内核的复杂性和挑战性。特别是在实验过程中，解决 GDB 符号表加载的问题让我意识到，细节的处理和配置是多么关键。没有正确

加载符号表，GDB 就无法识别内核中的符号，导致无法有效调试。这让我明白，调试不仅仅是找到错误和修复代码，更重要的是配置环境和工具的使用。

---

## System call tracing

---

### 实验目的

- 实现一个新的 `trace` 系统调用，以便在调试后续实验时使用
- 修改 xv6 内核，以便在每个系统调用即将返回时，打印出必要信息
- 熟悉操作系统如何从用户态发送请求调用内核态程序

### 实验步骤

#### 1. 添加系统调用原型和存根：

- 在 `user/user.h` 中添加系统调用原型：

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

- 在 `user/usys.pl` 中添加系统调用存根：

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

- 在 `kernel/syscall.h` 中添加系统调用编号：

```
#define SYS_trace 22
```

- 修改 `Makefile`，将 `$U/_trace` 添加到 `UPROGS`

#### 2. 实现系统调用功能：

- 在 `kernel/proc.h` 中为进程结构添加新的变量：

```
struct proc {  
    ...  
    int tracemask; // 用于存储跟踪掩码  
    ...  
};
```

- 在 `kernel/sysproc.c` 中实现 `sys_trace` 函数：

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

- 在 `kernel/proc.c` 中修改 `fork` 函数，将 trace mask 从父进程复制到子进程：

```
int fork(void) {
    ...
    np->tracemask = p->tracemask;
    ...
}
```

- 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"
};
extern uint64 sys_trace(void);
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,
};
```

- 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->tracemask & (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;
    }
    }>a0 = -1;
    }
}

```

### 3. 测试结果

- trace 32 grep hello README

```

$ trace 32 grep hello README
4: syscall read -> 1023
4: syscall read -> 961
4: syscall read -> 321
4: _syscall read -> 0

```

- trace 2147483647 grep hello README

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

- `grep hello README`
- `trace 2 usertests forkforkfork`

```
10: syscall fork -> 68
11: syscall fork -> 69
12: syscall fork -> -1
10: syscall fork -> -1
41: syscall fork -> -1
11: syscall fork -> -1
42: syscall fork -> -1
OK
7: syscall fork -> 70
ALL TESTS PASSED
```

### 实验中遇到的问题及解决办法

在 `sys_trace` 函数中处理参数时遇到错误。

- 确保在 `sys_trace` 函数中正确提取并存储 `mask` 参数。
- 在 `syscall` 函数中，确保正确检索当前进程的 `tracemask` 并检查对应的系统调用编号是否被设置。

### 实验心得

在这次实验中，我成功为 xv6 内核添加了系统调用跟踪功能，这使我对操作系统的系统调用机制有了更深入的理解。在实现过程中，我遇到了编译错误、参数获取以及进程继承等问题，但通过

仔细阅读文档和代码逐步解决了这些难题。特别是通过 `argint` 函数获取用户参数和在 `fork` 函数中继承 `tracemask`，让我对内核与用户空间的交互有了更清晰的认识。

## Sysinfo

### 实验目的

在操作系统中添加一个名为 `sysinfo` 的系统调用，该系统调用用于收集运行系统的信息，具体为：

- **收集空闲内存数量：** `sysinfo` 系统调用需要获取系统中当前的空闲内存的字节数。
- **统计非 UNUSED 状态的进程数量：** `sysinfo` 系统调用需要统计当前系统中状态不是 UNUSED 的进程数量。

### 实验步骤

- 在 `UPROGS` 中添加 `$U/_sysinfotest` 以确保编译测试程序。
- 在 `user/user.h` 文件中预声明 `struct sysinfo` 和 `sysinfo` 函数的原型。

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

- 在 `kernel/sysproc.c` 中添加 `sysinfo` 系统调用的实现。

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

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

- 在 `kernel/kalloc.c` 中添加一个函数用于获取系统的空闲内存数量。



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

- 在 `kernel/proc.c` 中添加一个函数用于统计系统中非 UNUSED 状态的进程数量。

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

- 在 `sysinfo.h` 中添加 `get_freemem()` 和 `get_nproc()` 的声明

```
uint64 get_nproc(void);
uint64 get_freemem(void);
```

- 测试, `make qemu` 后运行 `sysinfotest`

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

## 实验中遇到的困难及解决办法

编译中遇到报错:

```

kernel/sysproc.c: In function 'sys_sysinfo':
kernel/sysproc.c:113:20: error: implicit declaration of function 'get_freemem' [-
Werror=implicit-function-declaration]
  113 |         info.freemem = get_freemem();
      |                        ^~~~~~
kernel/sysproc.c:114:18: error: implicit declaration of function 'get_nproc' [-
Werror=implicit-function-declaration]
  114 |         info.nproc = get_nproc();
      |                        ^~~~~~

```

原因是没有将这两个函数在合适的头文件中声明，在sysinfo.h中补充两个函数的声明即可

## 实验心得

在实现 `sysinfo` 系统调用的过程中，我最大的体会是内核与用户空间之间的数据交互过程。实现一个系统调用不仅仅是简单地在内核中添加一个函数，更重要的是正确、安全地在内核空间和用户空间之间传递数据。在编写和调试 `get_freemem` 和 `get_nproc` 函数时，我对操作系统的内存管理和进程管理有了更深刻的理解。我学会了如何遍历内核数据结构（如空闲内存链表和进程表），并从中提取出有用的信息。

## 总结

本实验旨在通过实际操作来加深对操作系统系统调用机制的理解和应用。具体目标包括使用GDB调试工具进行内核调试、实现自定义系统调用、系统调用跟踪以及系统信息收集等。

- Using gdb
  - 目的：
    - 学习使用GDB调试工具来理解和定位Xv6内核中的问题和错误。
    - 理解函数调用关系，使用GDB进行内核级别调试。
  - 心得：通过使用GDB调试Xv6内核，学会了如何设置断点、查看调用堆栈、检查进程结构体，以及分析和解决内核panic的问题。解决GDB符号表加载问题，提升了对内核调试的理解。
- System call tracing
  - 目的：
    - 实现一个新的trace系统调用，以便在调试后续实验时使用。
    - 修改Xv6内核以在每个系统调用返回时打印出必要信息。
  - 心得：实现trace系统调用，加深了对系统调用机制的理解。遇到编译错误和参数获取问题，通过阅读文档和代码逐步解决，提升了内核与用户空间交互的理解。

- Sysinfo

- **目的:**

- 添加一个sysinfo系统调用，用于收集系统运行信息，包括空闲内存数量和非UNUSED状态的进程数量。

- 

- **心得:** 通过实现sysinfo系统调用，学会了如何在内核与用户空间之间安全地传递数据。实现过程中遇到的编译错误，通过添加函数声明解决，提升了对内存管理和进程管理的理解。