Go 语言系统调用

64位 Linux 上的系统调用

系统调用是操作系统内核提供给用户空间程序的一套标准接口。通过这套接口，用户态程序可以**受限地**访问硬件设备，从而实现申请系统资源，读写设备，创建新进程等操作。事实上，我们常用的 C 语言标准库中不少都是对操作系统提供的系统调用的封装，比如大家耳熟能详的 `printf`, `gets`, `fopen` 等，就分别是对 `read`, `write`, `open` 这些系统调用的封装。使用 `ltrace` 来追踪调用就可以清楚地看到这一点，例如：

```c
#include <stdio.h>
/* The well-known "Hello World" */
int main(void) {
    printf("Hello World!\n");
}
```

对于上面这段代码编译后使用 `ltrace` 调试，即可得到如下输出：

```
name1e5s@asgard:~$ gcc test.c
name1e5s@asgard:~$ ltrace -S ./a.out
```

```
SYS_brk(0)
                                = 0x55eb2abba000
SYS_access("/etc/ld.so.nohwcap", 00) = -2
SYS_access("/etc/ld.so.preload", 04) = -2
SYS_openat(0xffffff9c, 0x7f2290c00428, 0x80000, 0) = 3
SYS_fstat(3, 0x7ffd2e03aa20) = 0
SYS_mmap(0, 0x21b06, 1, 2)  = 0x7f2290de4000
SYS_close(3) = 0
SYS_access("/etc/ld.so.nohwcap", 00) = -2
SYS_openat(0xffffff9c, 0x7f2290e08dd0, 0x80000, 0) = 3
SYS_read(3, "\177ELF\002\001\001\003", 832) = 832
SYS_fstat(3, 0x7ffd2e03aa80) = 0
SYS_mmap(0, 8192, 3, 34) = 0x7f2290de2000
SYS_mmap(0, 0x3f0ae0, 5, 2050) = 0x7f22907ee000
SYS_mprotect(0x7f22909d5000, 2097152, 0) = 0
SYS_mmap(0x7f2290bd5000, 0x6000, 3, 2066) = 0x7f2290bd5000
SYS_mmap(0x7f2290bdb000, 0x3ae0, 3, 50) = 0x7f2290bdb000
SYS_close(3) = 0
SYS_arch_prctl(4098, 0x7f2290de34c0, 0x7f2290de3e00, 0x7f2290de2988) = 0
SYS_mprotect(0x7f2290bd5000, 16384, 1) = 0
SYS_mprotect(0x55eb28ecf000, 4096, 1) = 0
SYS_mprotect(0x7f2290e06000, 4096, 1) = 0
```

```
24 SYS_munmap(0x7f2290de4000, 137990) = 0
25 puts("Hello World!" <unfinished ...>
26 SYS_fstat(1, 0x7ffd2e03b280) = 0
27 SYS_brk(0) = 0x55eb2abba000
28 SYS_brk(0x55eb2abdb000) = 0x55eb2abdb000
29 SYS_write(1, "Hello World!\n", 13Hello World!
30 ) = 13
31 <... puts resumed> ) = 13
32 SYS_exit_group(0 <no return ...>
33 +++ exited (status 0) +++
```

其中 `SYS_` 开头的均为系统调用，可见系统调用几乎是无处不在。在当前版本的 `amd64 Linux` 内核中有不到四百个系统调用（详见这里），我们可以使用内核提供的 C 接口或者是直接使用汇编代码来调用他们。

历史上，`x86(-64)` 上共有`int 80`, `sysenter`, `syscall`三种方式来实现系统调用。`int 80` 是最传统的调用方式，其通过中断/异常来实现。`sysenter` 与 `syscall` 则都是通过引入新的寄存器组(Model-Specific Register(MSR))存放所需信息，进而实现快速跳转。这两者之间的主要区别就是定义的厂商不一样，`sysenter` 是 Intel 主推，后者则是 AMD 的定义。到了 64位时代，因为安腾架构（IA-64）大失败，农企终于借着 `x86_64` 架构咸鱼翻身，搞得 Intel 只得兼容 `syscall`。`Linux` 在 `2.6` 的后期开始引入 `sysenter` 指令，从当年遗留下来的文章来看，与老古董 `int 80` 比跑的确实比香港记者还要快。因此为了性能，我们的 Go 语言自然也是使用

`syscall/sysenter` 进行系统调用。如果读者想要了解更多关于 LInux 系统调用的知识，还请参阅这篇文章。

Go 语言中的系统调用

尽管 Go 语言具有 cgo 这样的设施可以方便快捷地调用 C 函数，但是其还是自己对系统调用进行了封装，以 `amd64` 架构为例， Go 语言中的系统调用是通过如下几个函数完成的：

```
1 // In syscall_unix.go
2 func Syscall(trap, a1, a2, a3 uintptr) (r1, r2
  uintptr, err Errno)
3 func Syscall6(trap, a1, a2, a3, a4, a5, a6
  uintptr) (r1, r2 uintptr, err Errno)
4 func RawSyscall(trap, a1, a2, a3 uintptr) (r1,
  r2 uintptr, err Errno)
5 func RawSyscall6(trap, a1, a2, a3, a4, a5, a6
  uintptr) (r1, r2 uintptr, err Errno)
```

其中 `Syscall` 对应参数不超过四个的系统调用， `Syscall6` 则对应参数不超过六个的系统调用。对于 `amd64` 架构的 Linux，这几个函数的实现在 `asm_linux_amd64.s` 内，代码不是很多，摘录如下：

```
1 // func Syscall(trap int64, a1, a2, a3
  uintptr) (r1, r2, err uintptr);
2 // Trap # in AX, args in DI SI DX R10 R8 R9,
  return in AX DX
```

```
3  // Note that this differs from "standard" ABI
   convention, which
4  // would pass 4th arg in CX, not R10.
5
6  TEXT ·Syscall(SB),NOSPLIT,$0-56
7      CALL    runtime·entersyscall(SB)
8      MOVQ    a1+8(FP), DI
9      MOVQ    a2+16(FP), SI
10     MOVQ    a3+24(FP), DX
11     MOVQ    $0, R10
12     MOVQ    $0, R8
13     MOVQ    $0, R9
14     MOVQ    trap+0(FP), AX  // syscall entry
15     SYSCALL
16     CMPQ    AX, $0xfffffffffffff001
17     JLS ok
18     MOVQ    $-1, r1+32(FP)
19     MOVQ    $0, r2+40(FP)
20     NEGQ    AX
21     MOVQ    AX, err+48(FP)
22     CALL    runtime·exitsyscall(SB)
23     RET
24 ok:
25     MOVQ    AX, r1+32(FP)
26     MOVQ    DX, r2+40(FP)
27     MOVQ    $0, err+48(FP)
28     CALL    runtime·exitsyscall(SB)
29     RET
30
```

```
31 // func Syscall6(trap, a1, a2, a3, a4, a5, a6
   uintptr) (r1, r2, err uintptr)
32 TEXT ·Syscall6(SB),NOSPLIT,$0-80
33     CALL    runtime·entersyscall(SB)
34     MOVQ    a1+8(FP), DI
35     MOVQ    a2+16(FP), SI
36     MOVQ    a3+24(FP), DX
37     MOVQ    a4+32(FP), R10
38     MOVQ    a5+40(FP), R8
39     MOVQ    a6+48(FP), R9
40     MOVQ    trap+0(FP), AX  // syscall entry
41     SYSCALL
42     CMPQ    AX, $0xfffffffffffff001
43     JLS ok6
44     MOVQ    $-1, r1+56(FP)
45     MOVQ    $0, r2+64(FP)
46     NEGQ    AX
47     MOVQ    AX, err+72(FP)
48     CALL    runtime·exitsyscall(SB)
49     RET
50 ok6:
51     MOVQ    AX, r1+56(FP)
52     MOVQ    DX, r2+64(FP)
53     MOVQ    $0, err+72(FP)
54     CALL    runtime·exitsyscall(SB)
55     RET
56
57 // func RawSyscall(trap, a1, a2, a3 uintptr)
   (r1, r2, err uintptr)
```

```
58 TEXT ·RawSyscall(SB),NOSPLIT,$0-56
59     MOVQ    a1+8(FP), DI
60     MOVQ    a2+16(FP), SI
61     MOVQ    a3+24(FP), DX
62     MOVQ    $0, R10
63     MOVQ    $0, R8
64     MOVQ    $0, R9
65     MOVQ    trap+0(FP), AX  // syscall entry
66     SYSCALL
67     CMPQ    AX, $0xfffffffffffff001
68     JLS ok1
69     MOVQ    $-1, r1+32(FP)
70     MOVQ    $0, r2+40(FP)
71     NEGQ    AX
72     MOVQ    AX, err+48(FP)
73     RET
74 ok1:
75     MOVQ    AX, r1+32(FP)
76     MOVQ    DX, r2+40(FP)
77     MOVQ    $0, err+48(FP)
78     RET
79
80 // func RawSyscall6(trap, a1, a2, a3, a4, a5,
   a6 uintptr) (r1, r2, err uintptr)
81 TEXT ·RawSyscall6(SB),NOSPLIT,$0-80
82     MOVQ    a1+8(FP), DI
83     MOVQ    a2+16(FP), SI
84     MOVQ    a3+24(FP), DX
85     MOVQ    a4+32(FP), R10
```

```
86      MOVQ    a5+40(FP), R8
87      MOVQ    a6+48(FP), R9
88      MOVQ    trap+0(FP), AX  // syscall entry
89      SYSCALL
90      CMPQ    AX, $0xfffffffffffff001
91      JLS ok2
92      MOVQ    $-1, r1+56(FP)
93      MOVQ    $0, r2+64(FP)
94      NEGQ    AX
95      MOVQ    AX, err+72(FP)
96      RET
97  ok2:
98      MOVQ    AX, r1+56(FP)
99      MOVQ    DX, r2+64(FP)
100     MOVQ    $0, err+72(FP)
101     RET
```

可以看到，`Syscall` 和 `RawSyscall` 在源代码上的区别就是有没有调用 `runtime` 包提供的两个函数。这意味着前者在发生阻塞时可以通知运行时并继续运行其他协 程，而后者只会卡掉整个程序。我们在自己封装自定义调用时应当尽量使用 `Syscall`。

自己封装系统调用

Go 语言通过手写与 `Perl` 脚本自动生成相结合的方式定义了很多系统调用的函数，可以查阅文档来使用，这里只举一个直接使用 `Syscall` 函数查看当前进程 PID 的例子：

```go
package main

import (
    "fmt"
    "syscall"
)

func main() {
    pid, _, _ := syscall.Syscall(39, 0, 0, 0)
// 用不到的就补上 0
    fmt.Println("Process id: ", pid)
}
```

输出如下:

```
name1e5s@asgard:~$ go run test.go
Process id:  19184
```