# Linux内核系统调用在glibc中实现

在MIPS体系结构中，用户进程通过“syscall”指令实现系统调用，进入内核空间运行。在glibc库函数中封装了系统调用，用户程序以函数调用的形式实现系统调用。

本文介绍glibc如何封装系统调用，读者可从glibc官网http://ftp.gnu.org/gnu/glibc/下载glibc源码。

## 1底层实现

每种体系结构实现系统调用的指令不同，因此最底层的实现必定是与体系结构相关的。MIPS体系结构通过syscall指令引发异常进入CPU核心态，以实现系统调用。

glibc在**/sysdeps/unix/sysv/linux/mips/mips32/sysdep.h**头文件实现了系统调用底层的封装函数：

# define **INTERNAL\_SYSCALL**(name, nr, args...) \

**internal\_syscall##nr** ("**li\t%0, %2\t\t\t#** " #name "\n\t", \ /\*“##”符号表示字符串拼接\*/

"IK" (SYS\_ify (name)), \

SYS\_ify (name), err, args)

各参数语义如下：

●**name**：系统调用名称，如open、fork等。

●**nr**：系统调用参数数量，0-7。

●**arg**：其它参数

internal\_syscall##nr ()宏中SYS\_ify (name)宏定义如下：

#define SYS\_ify(syscall\_name) \_\_NR\_##syscall\_name /\*系统调用编号\*/

\_\_NR\_##syscall\_name系统调用编号定义在**/sysdeps/unix/sysv/linux/mips/mips32/arch-syscall.h**头文件。

internal\_syscall##nr ()宏定义在**/sysdeps/unix/sysv/linux/mips/mips32/sysdep.h**头文件，对于不同参数的系统调用有不同的实现。例如下面是对一个参数系统调用实现的宏：

#define internal\_syscall1(v0\_init, input, number, err, **arg1**) \ /\*number：系统调用编号，arg1：参数\*/

({ \

long int \_sys\_result; \

\

{ \

long int \_arg1 = (long int) (arg1); \

register long int \_\_s0 asm ("$16") \_\_attribute\_\_ ((unused)) \

= (**number**); \

register long int \_\_v0 asm ("$2"); \

register long int \_\_a0 asm ("$4") = **\_arg1**; \ /\*a0保存第一个参数值\*/

register long int \_\_a3 asm ("$7"); \

\_\_asm\_\_ volatile ( \

".set\tnoreorder\n\t" \

**v0\_init** \ /\*系统调用编号存入v0\*/

"**syscall**\n\t" \ /\*syscall指令\*/

".set reorder" \

: "=r" (**\_\_v0**), "=r" (\_\_a3) \

: input, "r" (\_\_a0) \

: \_\_SYSCALL\_CLOBBERS); \

\_sys\_result = \_\_a3 != 0 ? -\_\_v0 : \_\_v0; \ /\*v0保存返回值\*/

} \

\_sys\_result; \ /\*返回值\*/

})

其它参数数量对应的internal\_syscallx()宏请读者自行阅读源代码。

## 2通用接口

前面介绍的是系统调用与体系结构相关的底层实现。glibc在**/sysdeps/unix/sysdep.h**定义了调用系统调用的通用接口。

#define **INTERNAL\_SYSCALL\_CALL**(...) \ /\*调用系统调用\*/

\_\_INTERNAL\_SYSCALL\_DISP (**\_\_INTERNAL\_SYSCALL**, \_\_VA\_ARGS\_\_)

#define \_\_INTERNAL\_SYSCALL\_DISP(b,...) \

\_\_SYSCALL\_CONCAT (**b**,**\_\_INTERNAL\_SYSCALL\_NARGS(\_\_VA\_ARGS\_\_)**)(\_\_VA\_ARGS\_\_)

在以上宏中b是“**\_\_INTERNAL\_SYSCALL”**字符串，\_\_INTERNAL\_SYSCALL\_NARGS()表示参数\_\_VA\_ARGS\_\_的数量值，例如，没有参数为0，一个参数为1，依此类推。

INTERNAL\_SYSCALL\_CALL() 宏参数包括系统调用名称和系统调用参数，不需要传递系统调用参数数量。

\_\_SYSCALL\_CONCAT (a,b)宏是把a和b两个参数进行字符串拼接。

因此，INTERNAL\_SYSCALL\_CALL(...)展开的最终效果如下：

**\_\_INTERNAL\_SYSCALLx(**\_\_VA\_ARGS\_\_**)**  /\*x表示系统调用参数个数\*/

\_\_INTERNAL\_SYSCALLx()宏定义如下（/sysdeps/unix/sysdep.h）：

#define \_\_INTERNAL\_SYSCALL0(name) \

**INTERNAL\_SYSCALL** (name, 0) /\*体系结构定义的系统调用宏，见上文\*/

#define \_\_INTERNAL\_SYSCALL1(name, a1) \

INTERNAL\_SYSCALL (name, 1, a1)

#define \_\_INTERNAL\_SYSCALL2(name, a1, a2) \

INTERNAL\_SYSCALL (name, 2, a1, a2)

...

glibc还定义了调用系统调用的INLINE\_SYSCALL\_CALL()宏，如下所示：

#define **INLINE\_SYSCALL\_CALL**(...) \ /\*参数：系统调用名称、系统调用参数（不需要数量值）\*/

\_\_INLINE\_SYSCALL\_DISP (\_\_INLINE\_SYSCALL, \_\_VA\_ARGS\_\_)

INLINE\_SYSCALL\_CALL()宏与INTERNAL\_SYSCALL\_CALL()宏类似，最后展开形式如下：

\_\_INLINE\_SYSCALLx(\_\_VA\_ARGS\_\_) /\*x表示系统调用参数个数\*/

\_\_INLINE\_SYSCALLx()宏定义如下（/sysdeps/unix/sysdep.h）：

#define \_\_INLINE\_SYSCALL0(name) \

INLINE\_SYSCALL (name, 0)

#define \_\_INLINE\_SYSCALL1(name, a1) \

INLINE\_SYSCALL (name, 1, a1)

...

INLINE\_SYSCALL ()宏定义在**/sysdeps/unix/sysv/linux/sysdep.h**头文件，如下所示：

#define INLINE\_SYSCALL(name, nr, args...) \

({ \

long int sc\_ret = **INTERNAL\_SYSCALL** (name, nr, args); \

\_\_glibc\_unlikely (INTERNAL\_SYSCALL\_ERROR\_P (sc\_ret)) \

? SYSCALL\_ERROR\_LABEL (INTERNAL\_SYSCALL\_ERRNO (sc\_ret)) \

: sc\_ret; \

})

INLINE\_SYSCALL ()宏调用INTERNAL\_SYSCALL ()宏实现系统调用。

在/sysdeps/unix/sysdep.h头文件内还定义了其它引用系统调用的宏，例如：

●**SYSCALL\_CANCEL()**

**●INTERNAL\_SYSCALL\_CANCEL()**

## 3库函数实现

内核系统调用被glibc封装成库函数，使用户程序可以函数调用的方式实现系统调用。Linux内核系统调用封装函数主要位于/sysdeps/unix/sysv/linux/目录文件内（其它目录应该也还有）。

例如，read系统调用的封装函数位于/sysdeps/unix/sysv/linux/read.c文件内，代码简列如下：

ssize\_t \_\_libc\_read (int fd, void \*buf, size\_t nbytes)

{

return **SYSCALL\_CANCEL** (**read, fd, buf, nbytes**); /\*执行系统调用\*/

}

libc\_hidden\_def (\_\_libc\_read)

libc\_hidden\_def (\_\_read)

weak\_alias (\_\_libc\_read, \_\_read)

libc\_hidden\_def (read)

**weak\_alias (\_\_libc\_read, read)** /\*函数别名read()\*/