NAME

syscall − indirect system call

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/syscall.h> /* Definition of **SYS_*** constants */
#include <unistd.h>

long syscall(long *number***, ...);**

Feature Test Macro Requirements for glibc (see **feature_test_macros**(7)):

syscall():
Since glibc 2.19:
_DEFAULT_SOURCE
Before glibc 2.19:
_BSD_SOURCE || _SVID_SOURCE

DESCRIPTION

syscall() is a small library function that invokes the system call whose assembly language interface has the specified *number* with the specified arguments. Employing **syscall**() is useful, for example, when invoking a system call that has no wrapper function in the C library.

syscall() saves CPU registers before making the system call, restores the registers upon return from the system call, and stores any error returned by the system call in **errno**(3).

Symbolic constants for system call numbers can be found in the header file<*sys/syscall.h*>.

RETURN VALUE

The return value is defined by the system call being invoked. In general, a 0 return value indicates success. A −1 return value indicates an error, and an error number is stored in *errno*.

NOTES

syscall() first appeared in 4BSD.

Architecture-specific requirements

Each architecture ABI has its own requirements on how system call arguments are passed to the kernel. For system calls that have a glibc wrapper (e.g., most system calls), glibc handles the details of copying arguments to the right registers in a manner suitable for the architecture. However, when using **syscall**() to make a system call, the caller might need to handle architecture-dependent details; this requirement is most commonly encountered on certain 32-bit architectures.

For example, on the ARM architecture Embedded ABI (EABI), a 64-bit value (e.g., *long long*) must be aligned to an even register pair. Thus, using**syscall**() instead of the wrapper pro vided by glibc, the **readahead**(2) system call would be invoked as follows on the ARM architecture with the EABI in little endian mode:

```
syscall(SYS_readahead, fd, 0,
        (unsigned int) (offset & 0xFFFFFFFF),
        (unsigned int) (offset >> 32),
        count);
```

Since the offset argument is 64 bits, and the first argument (*fd*) is passed in *r0*, the caller must manually split and align the 64-bit value so that it is passed in the *r2*/*r3* register pair. That means inserting a dummy value into *r1* (the second argument of 0). Care also must be taken so that the split follows endian conventions (according to the C ABI for the platform).

Similar issues can occur on MIPS with the O32 ABI, on PowerPC and parisc with the 32-bit ABI, and on Xtensa.

Note that while the parisc C ABI also uses aligned register pairs, it uses a shim layer to hide the issue from user space.

The affected system calls are **fadvise64_64**(2), **ftruncate64**(2), **posix_fadvise**(2), **pread64**(2), **pwrite64**(2), **readahead**(2), **sync_file_range**(2), and **truncate64**(2).

This does not affect syscalls that manually split and assemble 64-bit values such as **_llseek**(2), **preadv**(2), **preadv2**(2), **pwritev**(2), and **pwritev2**(2). Welcome to the wonderful world of historical baggage.

Architecture calling conventions

Every architecture has its own way of invoking and passing arguments to the kernel. The details for various architectures are listed in the two tables below.

The first table lists the instruction used to transition to kernel mode (which might not be the fastest or best way to transition to the kernel, so you might have to refer to **vdso**(7)), the register used to indicate the system call number, the register(s) used to return the system call result, and the register used to signal an error.

Arch/ABI	Instruction	System call #	Ret val	Ret val2	Error	Notes
alpha	callsys	v0	v0	a4	a3	1, 6
arc	trap0	r8	r0	-	-	
arm/OABI	swi NR	-	r0	-	-	2
arm/EABI	swi 0x0	r7	r0	r1	-	
arm64	svc #0	w8	x0	x1	-	
blackfin	excpt 0x0	P0	R0	-	-	
i386	int $0x80	eax	eax	edx	-	
ia64	break 0x100000	r15	r8	r9	r10	1, 6
loongarch	syscall 0	a7	a0	-	-	
m68k	trap #0	d0	d0	-	-	
microblaze	brki r14,8	r12	r3	-	-	
mips	syscall	v0	v0	v1	a3	1, 6
nios2	trap	r2	r2	-	r7	
parisc	ble 0x100(%sr2, %r0)	r20	r28	-	-	
powerpc	sc	r0	r3	-	r0	1
powerpc64	sc	r0	r3	-	cr0.SO	1
riscv	ecall	a7	a0	a1	-	
s390	svc 0	r1	r2	r3	-	3
s390x	svc 0	r1	r2	r3	-	3
superh	trapa #31	r3	r0	r1	-	4, 6
sparc/32	t 0x10	g1	o0	o1	psr/csr	1, 6
sparc/64	t 0x6d	g1	o0	o1	psr/csr	1, 6
tile	swint1	R10	R00	-	R01	1
x86-64	syscall	rax	rax	rdx	-	5
x32	syscall	rax	rax	rdx	-	5
xtensa	syscall	a2	a2	-	-	

Notes:

- On a few architectures, a register is used as a boolean (0 indicating no error, and −1 indicating an error) to signal that the system call failed. The actual error value is still contained in the return register. On sparc, the carry bit (*csr*) in the processor status register (*psr*) is used instead of a full register. On powerpc64, the summary overflow bit (*SO*) in field 0 of the condition register (*cr0*) is used.
- *NR* is the system call number.
- For s390 and s390x, *NR* (the system call number) may be passed directly with *svc NR* if it is less than 256.
- On SuperH additional trap numbers are supported for historic reasons, but **trapa**#31 is the recommended "unified" ABI.
- The x32 ABI shares syscall table with x86-64 ABI, but there are some nuances:

- In order to indicate that a system call is called under the x32 ABI, an additional bit, **__X32_SYSCALL_BIT**, is bitwise-ORed with the system call number. The ABI used by a process affects some process behaviors, including signal handling or system call restarting.
- Since x32 has different sizes for *long* and pointer types, layouts of some (but not all; *struct timeval* or *struct rlimit* are 64-bit, for example) structures are different. In order to handle this, additional system calls are added to the system call table, starting from number 512 (without the **__X32_SYSCALL_BIT**). For example, **__NR_readv** is defined as 19 for the x86-64 ABI and as *__X32_SYSCALL_BIT* | **515** for the x32 ABI. Most of these additional system calls are actually identical to the system calls used for providing i386 compat. There are some notable exceptions, however, such as **preadv2**(2), which uses *struct iovec* entities with 4-byte pointers and sizes ("compat_iovec" in kernel terms), but passes an 8-byte *pos* argument in a single register and not two, as is done in every other ABI.

- Some architectures (namely, Alpha, IA-64, MIPS, SuperH, sparc/32, and sparc/64) use an additional register ("Retval2" in the above table) to pass back a second return value from the **pipe**(2) system call; Alpha uses this technique in the architecture-specific **getxpid**(2), **getxuid**(2), and **getxgid**(2) system calls as well. Other architectures do not use the second return value register in the system call interface, even if it is defined in the System V ABI.

The second table shows the registers used to pass the system call arguments.

Arch/ABI	arg1	arg2	arg3	arg4	arg5	arg6	arg7	Notes
alpha	a0	a1	a2	a3	a4	a5	-	
arc	r0	r1	r2	r3	r4	r5	-	
arm/OABI	r0	r1	r2	r3	r4	r5	r6	
arm/EABI	r0	r1	r2	r3	r4	r5	r6	
arm64	x0	x1	x2	x3	x4	x5	-	
blackfin	R0	R1	R2	R3	R4	R5	-	
i386	ebx	ecx	edx	esi	edi	ebp	-	
ia64	out0	out1	out2	out3	out4	out5	-	
loongarch	a0	a1	a2	a3	a4	a5	a6	
m68k	d1	d2	d3	d4	d5	a0	-	
microblaze	r5	r6	r7	r8	r9	r10	-	
mips/o32	a0	a1	a2	a3	-	-	-	1
mips/n32,64	a0	a1	a2	a3	a4	a5	-	
nios2	r4	r5	r6	r7	r8	r9	-	
parisc	r26	r25	r24	r23	r22	r21	-	
powerpc	r3	r4	r5	r6	r7	r8	r9	
powerpc64	r3	r4	r5	r6	r7	r8	-	
riscv	a0	a1	a2	a3	a4	a5	-	
s390	r2	r3	r4	r5	r6	r7	-	
s390x	r2	r3	r4	r5	r6	r7	-	
superh	r4	r5	r6	r7	r0	r1	r2	
sparc/32	o0	o1	o2	o3	o4	o5	-	
sparc/64	o0	o1	o2	o3	o4	o5	-	
tile	R00	R01	R02	R03	R04	R05	-	
x86-64	rdi	rsi	rdx	r10	r8	r9	-	
x32	rdi	rsi	rdx	r10	r8	r9	-	
xtensa	a6	a3	a4	a5	a8	a9	-	

Notes:

- The mips/o32 system call convention passes arguments 5 through 8 on the user stack.

Note that these tables don’t cover the entire calling convention—some architectures may indiscriminately clobber other registers not listed here.

EXAMPLES

```
#define _GNU_SOURCE
#include <signal.h>
#include <sys/syscall.h>
#include <unistd.h>

int
main(void)
{
    pid_t tid;

    tid = syscall(SYS_gettid);
    syscall(SYS_tgkill, getpid(), tid, SIGHUP);
}
```

SEE ALSO

_syscall(2), **intro**(2), **syscalls**(2), **errno**(3), **vdso**(7)