

How system calls work in Linux

Nasser M. Abbasi

May 29, 2000

Compiled on October 27, 2025 at 5:09pm

Contents

These are notes I wrote while learning how system calls work on a Linux system.

To help show this how system call works, I show flow of a typical system call such as `fopen()`.

`fopen()` is a function call defined in the C standard library. I use glibc-2.1 as an implementation.

From the UNIX98 standard, `fopen()` is defined as

```
#include <stdio.h>
```

```
FILE *fopen(const char *filename, const char *mode);
```

DESCRIPTION

The `fopen()` function opens the file whose pathname is the string pointed to by `filename`, and associates a stream with it.

The argument `mode` points to a string beginning with one of the following sequences:

`r` or `rb`

Open file for reading.

`w` or `wb`

Truncate to zero length or create file for writing.

`a` or `ab`

Append; open or create file for writing at end-of-file.

`r+` or `rb+` or `r+b`

Open file for update (reading and writing).

`w+` or `wb+` or `w+b`

Truncate to zero length or create file for update.

`a+` or `ab+` or `a+b`

Append; open or create file for update, writing at end-of-file.

Create the following t.c C program to use to test with:

```
#include <stdio.h>
```

```
int main(int argc, char *argv[])
```

```
{
```

```
FILE *f;
```

```
f = fopen("test.txt", "r");
```

```
return 0;
```

```
}
```

To step into `fopen()`, glibc 2.1 was build in debug and the new build `libc.a` was linked against instead of the default installed `libc` on my linux box.

To build glibc, the following are steps performed. A good reference is the glibc2 HOWTO, <http://www.linux.ps.pl/doc/other/LDP/HOWTO/Glibc2-HOWTO.html>

First, I downloaded the glibc tar file to `/usr/src/packages/SOURCES`. Extracted It and it created `glibc-2.1/` directory. Then copied the crypt tar file into `glibc-2.1/` and extracted that. It created `crypt/` directory under `glibc-2.1/`. Next I did

```
cd glibc-2.1
./configure --enable-add-ons
make
make check
```

Next, Installed the library into a direcorey called `INSTALL_LIB` under `glibc-2.1`.

```
make install install_root=/usr/src/packages/SOURCES/glibc-2.1/INSTALL_LIB
```

OK, now glibc-2.1 is compiled and ready to use. Back to the little C program we have above. Lets now compile it and link it to the above library.

```
gcc -static -g -I /usr/src/packages/SOURCES/glibc-2.1/INSTALL_LIB/usr/local/
include \
-L/usr/src/packages/SOURCES/glibc-2.1/INSTALL_LIB/usr/local/lib t.c
```

Ok, now lets step through it.

```
$gdb ./a.out
GNU gdb 4.18
(gdb) break main
Breakpoint 1 at 0x80481b6: file t.c, line 7.

(gdb) run
Starting program: /export/g/nabbasi/data/my_misc_programs/my_c/./a.out

Breakpoint 1, main (argc=1, argv=0xbffff434) at t.c:7
7      f = fopen("test.txt","r");
(gdb) list
2
3      int main(int argc, char *argv[])
4      {
5          FILE *f;
6
7          f = fopen("test.txt","r");
8
9          return 0;
10     }

(gdb) disassemble main
Dump of assembler code for function main:
0x80481b0 <main>:      push    %ebp
0x80481b1 <main+1>:    mov     %esp,%ebp
0x80481b3 <main+3>:    sub     $0x4,%esp
0x80481b6 <main+6>:    push    $0x8071ba8
0x80481bb <main+11>:   push    $0x8071baa
0x80481c0 <main+16>:   call   0x8048710 <_IO_new_fopen>
0x80481c5 <main+21>:   add     $0x8,%esp
0x80481c8 <main+24>:   mov     %eax,%eax
```

```

0x80481ca <main+26>:    mov    %eax,0xffffffff(%ebp)
0x80481cd <main+29>:    xor    %eax,%eax
0x80481cf <main+31>:    jmp    0x80481e0 <main+48>
0x80481d1 <main+33>:    jmp    0x80481e0 <main+48>
0x80481e0 <main+48>:    mov    %ebp,%esp
0x80481e2 <main+50>:    pop    %ebp
0x80481e3 <main+51>:    ret
End of assembler dump.

```

Humm... what happened to printf call? you will notice, it is now a call to `_IO_new_fopen`. But I was calling `fopen`, not `_IO_new_fopen`?

Lets step into `_IO_new_fopen` and see what happened.

```

(gdb) s
_IO_new_fopen (filename=0x8071baa "test.txt", mode=0x8071ba8 "r") at iofopen.c:42
42      } *new_f = (struct locked_FILE *) malloc (sizeof (struct locked_FILE));

```

So, `_IO_new_fopen` is an entry in `iofopen.c`. Where is this file?

```

cd glibc-2.1
find . -name iofopen.c

```

will show it as `glibc-2.1/libio/iofopen.c` Lets look at it

```

#include "libioP.h"
#ifdef __STDC__
#include <stdlib.h>
#endif
_IO_FILE *
_IO_new_fopen (filename, mode)
    const char *filename;
    const char *mode;
{
    struct locked_FILE
    {
        struct _IO_FILE_plus fp;
#ifdef _IO_MTSAFE_IO
        _IO_lock_t lock;
#endif
    } *new_f = (struct locked_FILE *) malloc (sizeof (struct locked_FILE));
    if (new_f == NULL)
        return NULL;
#ifdef _IO_MTSAFE_IO
    new_f->fp.file._lock = &new_f->lock;
#endif
    _IO_init (&new_f->fp.file, 0);
    _IO_JUMPS (&new_f->fp) = &_IO_file_jumps;
    _IO_file_init (&new_f->fp.file);
    #if !_IO_UNIFIED_JUMPTABLES
    new_f->fp.vtable = NULL;
    #endif
    if (_IO_file_fopen (&new_f->fp.file, filename, mode, 1) != NULL)
        return (_IO_FILE *) &new_f->fp;
    _IO_un_link (&new_f->fp.file);
    free (new_f);
    return NULL;
}
#if defined PIC && DO_VERSIONING
strong_alias (_IO_new_fopen, __new_fopen)

```

```

default_symbol_version (_IO_new_fopen, _IO_fopen, GLIBC_2.1);
default_symbol_version (__new_fopen, fopen, GLIBC_2.1);
#else
# ifdef weak_alias
weak_alias (_IO_new_fopen, _IO_fopen)
weak_alias (_IO_new_fopen, fopen)
# endif
#endif

```

Notice at the end what it says, it says `weak_alias (_IO_new_fopen, fopen)`. This tells gcc that `_IO_new_fopen` is an alias to `fopen`. (weak alias). Let me make sure. Looking at `libc.a` now

```

cd /usr/src/packages/SOURCES/glibc-2.1/INSTALL_LIB/usr/local/lib
nm libc.a
...
iofopen.o:
          U _IO_file_fopen
          U _IO_file_init
          U _IO_file_jumps
00000000 W _IO_fopen
          U _IO_init
00000000 T _IO_new_fopen
          U _IO_un_link
          U __pthread_atfork
          U __pthread_getspecific
          U __pthread_initialize
          U __pthread_key_create
          U __pthread_mutex_destroy
          U __pthread_mutex_init
          U __pthread_mutex_lock
          U __pthread_mutex_trylock
          U __pthread_mutex_unlock
          U __pthread_mutexattr_destroy
          U __pthread_mutexattr_init
          U __pthread_mutexattr_settype
          U __pthread_once
          U __pthread_setspecific
          U _pthread_cleanup_pop_restore
          U _pthread_cleanup_push_defer
00000000 W fopen
          U free
          U malloc
...

```

Notice `fopen` has `W` next to it, meaning a Weak symbol. So, the linker when it sees a call to `fopen` will bind the call to `_IO_new_fopen`.

It is just a different name for `fopen`. This way, library can create different implementations for calls without the user program having to change.

Ok, now, lets continue to see where we will end up. back to `gdb`.

```

(gdb) disassemble fopen
Dump of assembler code for function _IO_new_fopen:
0x8048710 <_IO_new_fopen>:      push    %ebp
0x8048711 <_IO_new_fopen+1>:    mov     %esp,%ebp
0x8048713 <_IO_new_fopen+3>:    push    %ebx
0x8048714 <_IO_new_fopen+4>:    push    $0xb0
0x8048719 <_IO_new_fopen+9>:    call   0x804b020 <__libc_malloc>

```

```

0x804871e <_IO_new_fopen+14>: mov    %eax,%ebx
0x8048720 <_IO_new_fopen+16>: add    $0x4,%esp
0x8048723 <_IO_new_fopen+19>: test   %ebx,%ebx
0x8048725 <_IO_new_fopen+21>: jne     0x8048730 <_IO_new_fopen+32>
0x8048727 <_IO_new_fopen+23>: xor     %eax,%eax
0x8048729 <_IO_new_fopen+25>: jmp     0x8048782 <_IO_new_fopen+114>
0x804872b <_IO_new_fopen+27>: nop
0x804872c <_IO_new_fopen+28>: lea     0x0(%esi,1),%esi
0x8048730 <_IO_new_fopen+32>: lea     0x98(%ebx),%edx
0x8048736 <_IO_new_fopen+38>: mov     %edx,0x48(%ebx)
0x8048739 <_IO_new_fopen+41>: push    $0x0
0x804873b <_IO_new_fopen+43>: push    %ebx
0x804873c <_IO_new_fopen+44>: call    0x804a030 <_IO_init>
0x8048741 <_IO_new_fopen+49>: movl    $0x807a360,0x94(%ebx)
0x804874b <_IO_new_fopen+59>: push    %ebx
0x804874c <_IO_new_fopen+60>: call    0x80487a0 <_IO_new_file_init>
0x8048751 <_IO_new_fopen+65>: push    $0x1
0x8048753 <_IO_new_fopen+67>: mov     0xc(%ebp),%eax
0x8048756 <_IO_new_fopen+70>: push    %eax
0x8048757 <_IO_new_fopen+71>: mov     0x8(%ebp),%eax
0x804875a <_IO_new_fopen+74>: push    %eax
0x804875b <_IO_new_fopen+75>: push    %ebx
0x804875c <_IO_new_fopen+76>: call    0x80488e0 <_IO_new_file_fopen>
0x8048761 <_IO_new_fopen+81>: add     $0x1c,%esp
0x8048764 <_IO_new_fopen+84>: test    %eax,%eax
0x8048766 <_IO_new_fopen+86>: jne     0x8048780 <_IO_new_fopen+112>
0x8048768 <_IO_new_fopen+88>: push    %ebx
0x8048769 <_IO_new_fopen+89>: call    0x80497a0 <_IO_un_link>
0x804876e <_IO_new_fopen+94>: push    %ebx
0x804876f <_IO_new_fopen+95>: call    0x804b9f0 <__libc_free>
0x8048774 <_IO_new_fopen+100>: xor     %eax,%eax
0x8048776 <_IO_new_fopen+102>: jmp     0x8048782 <_IO_new_fopen+114>
0x8048778 <_IO_new_fopen+104>: nop
0x8048779 <_IO_new_fopen+105>: lea     0x0(%esi,1),%esi
0x8048780 <_IO_new_fopen+112>: mov     %ebx,%eax
0x8048782 <_IO_new_fopen+114>: mov     0xffffffffc(%ebp),%ebx
0x8048785 <_IO_new_fopen+117>: mov     %ebp,%esp
0x8048787 <_IO_new_fopen+119>: pop     %ebp
0x8048788 <_IO_new_fopen+120>: ret
End of assembler dump.

```

The call I am interested in is `_IO_new_file_fopen`. The earlier calls were calls that create and initialize data structures. I am interested in finding the call that will result in interrupt 0x80. So, let's step to `_IO_new_file_fopen`.

```

(gdb) break _IO_new_file_fopen
Breakpoint 3 at 0x80488ec: file fileops.c, line 204.
(gdb) continue
Continuing.

Breakpoint 3, _IO_new_file_fopen (fp=0x807c838, filename=0x8071baa "test.txt",
    mode=0x8071ba8 "r", is32not64=1) at fileops.c:204
204      int oflags = 0, omode;
(gdb)

```

The file `fileops.c` is located in `glibc-2.1/libio/`, let's look at the source code for `_IO_file_fopen()` in that file:

```

_IO_FILE *

```

```

_IO_new_file_fopen (fp, filename, mode, is32not64)
    _IO_FILE *fp;
    const char *filename;
    const char *mode;
    int is32not64;
{
    int oflags = 0, omode;
    int read_write;
    int oprot = 0666;
    int i;
    if (_IO_file_is_open (fp))
        return 0;
    switch (*mode)
    {
        case 'r':
            omode = O_RDONLY;
            read_write = _IO_NO_WRITES;
            break;
        case 'w':
            omode = O_WRONLY;
            oflags = O_CREAT|O_TRUNC;
            read_write = _IO_NO_READS;
            break;
        case 'a':
            omode = O_WRONLY;
            oflags = O_CREAT|O_APPEND;
            read_write = _IO_NO_READS|_IO_IS_APPENDING;
            break;
        default:
            __set_errno (EINVAL);
            return NULL;
    }
    for (i = 1; i < 4; ++i)
    {
        switch (*++mode)
        {
            case '\0':
                break;
            case '+':
                omode = O_RDWR;
                read_write &= _IO_IS_APPENDING;
                continue;
            case 'x':
                oflags |= O_EXCL;
                continue;
            case 'b':
            default:
                /* Ignore. */
                continue;
        }
        break;
    }

    return _IO_file_open (fp, filename, omode|oflags, oprot, read_write, ----> step here
                          is32not64);
}

```

Let us assume the file is not allready open, the next call will be `_IO_file_open()`

Setting a break point there. But notice, looking at source code in `fileops.c`, the above call to `_IO_file_open` is inlined (for performance?)

```
#if defined __GNUC__ && __GNUC__ >= 2
__inline__
#endif
_IO_FILE *
_IO_file_open (fp, filename, posix_mode, prot, read_write, is32not64)
    _IO_FILE *fp;
    const char *filename;
    int posix_mode;
    int prot;
    int read_write;
    int is32not64;
{
    int fdesc;
#ifdef _G_OPEN64
    fdesc = (is32not64
        ? open (filename, posix_mode, prot)
        : _G_OPEN64 (filename, posix_mode, prot));
#else
    fdesc = open (filename, posix_mode, prot);
#endif
    if (fdesc < 0)
        return NULL;
    fp->_fileno = fdesc;
    _IO_mask_flags (fp, read_write, _IO_NO_READS+_IO_NO_WRITES+_IO_IS_APPENDING);
    if (read_write & _IO_IS_APPENDING)
        if (_IO_SEEKOFF (fp, (_IO_off64_t)0, _IO_seek_end, _IOS_INPUT|_IOS_OUTPUT)
            == _IO_pos_BAD && errno != EPIPE)
            return NULL;
    _IO_link_in (fp);
    return fp;
}
```

Setting a break point at the call to `open` above.

```
(gdb) where
#0  _IO_new_file_fopen (fp=0x807c838, filename=0x8071baa "test.txt", mode=0x8071ba8 "r", is32not64=0) at iofopen.c:174
#1  0x8048761 in _IO_new_fopen (filename=0x8071baa "test.txt", mode=0x8071ba8 "r") at iofopen.c:174
#2  0x80481c5 in main (argc=1, argv=0xbffff434) at t.c:7
(gdb) list
174         int read_write;
175         int is32not64;
176     {
177         int fdesc;
178         #ifdef _G_OPEN64
179             fdesc = (is32not64
180                 ? open (filename, posix_mode, prot) -----> This is call we need
181                 : _G_OPEN64 (filename, posix_mode, prot));
182         #else
183             fdesc = open (filename, posix_mode, prot);
(gdb) break open
Breakpoint 2 at 0x804df80
(gdb)
```

Since `_IO_file_fopen` is inlined inside `_IO_new_file_fopen`, we can look at the assembler call to `open` above by disassembly of `_IO_new_file_fopen()`.

I'll show only the part where the call to `open` is made

```
(gdb) disassemble _IO_new_file_fopen
Dump of assembler code for function _IO_new_file_fopen:
...
0x80489e4 <_IO_new_file_fopen+260>:    push    $0x1b6
0x80489e9 <_IO_new_file_fopen+265>:    push    %eax
0x80489ea <_IO_new_file_fopen+266>:    mov     0xc(%ebp),%edi
0x80489ed <_IO_new_file_fopen+269>:    push    %edi
0x80489ee <_IO_new_file_fopen+270>:    call   0x804df80 <__libc_open>  ---->
    this is open
...
```

Ok, back to gdb, setting a breakpoint at `open` and stepping into it

```
(gdb) where
#0  _IO_new_file_fopen (fp=0x807c838, filename=0x8071baa "test.txt", mode=0
    x8071ba8 "r", is32not64=1) at fileops.c:179
#1  0x8048761 in _IO_new_fopen (filename=0x8071baa "test.txt", mode=0x8071ba8 "r"
    ") at iofopen.c:55
#2  0x80481c5 in main (argc=1, argv=0xbffff434) at t.c:7

Breakpoint 2, 0x804df80 in __libc_open ()
(gdb) disassemble
Dump of assembler code for function __libc_open:
0x804df80 <__libc_open>:    push    %ebx
0x804df81 <__libc_open+1>:    mov     0x10(%esp,1),%edx
0x804df85 <__libc_open+5>:    mov     0xc(%esp,1),%ecx
0x804df89 <__libc_open+9>:    mov     0x8(%esp,1),%ebx
0x804df8d <__libc_open+13>:   mov     $0x5,%eax
0x804df92 <__libc_open+18>:   int     $0x80  -----> to kernel mode.
0x804df94 <__libc_open+20>:   pop     %ebx
0x804df95 <__libc_open+21>:   cmp     $0xfffff001,%eax
0x804df9a <__libc_open+26>:   jae     0x804e450 <__syscall_error>
0x804dfa0 <__libc_open+32>:   ret
End of assembler dump.
(gdb)
```

We are finally there. The `open()` call being made from `_IO_file_open()`, is translated to `__libc_open()` and `__libc_open()` will issue the interrupt `0x80`, which will turn the processor to run in kernel more, and the interrupt handler will locate the kernel system call to process `open()`.

But before jumping into kernel mode, lets see how did the call to `open()` become a call to `__libc_open()` It turns out that when building glibc-2.1, there is a file called `glibc-2.1/sysdeps/unix/syscalls.list`

This file is used by the glibc build system to generate the wrapper for `open()` and call it `__libc_open`.

```
>cat glibc-2.1/sysdeps/unix/syscalls.list
# File name      Caller  Syscall name    # args  Strong name    Weak names

access           -      access          2       __access       access
acct             -      acct            1       acct           acct
chdir            -      chdir           1       __chdir        chdir
chmod            -      chmod           2       __chmod        chmod
chown            -      chown           3       __chown        chown
```

chroot	-	chroot	1	chroot	
close	-	close	1	__libc_close	__close close
dup	-	dup	1	__dup	dup
dup2	-	dup2	2	__dup2	dup2
fchdir	-	fchdir	1	__fchdir	fchdir
fcntl	-	fcntl	3	__libc_fcntl	__fcntl fcntl
fstatfs	-	fstatfs	2	__fstatfs	fstatfs
fsync	-	fsync	1	__libc_fsync	fsync
getdomain	-	getdomainname	2	getdomainname	
getgid	-	getgid	0	__getgid	getgid
getgroups	-	getgroups	2	__getgroups	getgroups
getitimer	-	getitimer	2	__getitimer	getitimer
getpid	-	getpid	0	__getpid	getpid
getpriority	-	getpriority	2	getpriority	
getrlimit	-	getrlimit	2	__getrlimit	getrlimit
getuid	-	getuid	0	__getuid	getuid
ioctl	-	ioctl	3	__ioctl	ioctl
kill	-	kill	2	__kill	kill
link	-	link	2	__link	link
lseek	-	lseek	3	__libc_lseek	__lseek lseek
mkdir	-	mkdir	2	__mkdir	mkdir
open	-	open	3	__libc_open	__open open
profil	-	profil	4	profil	
ptrace	-	ptrace	4	ptrace	
read	-	read	3	__libc_read	__read read
readlink	-	readlink	3	__readlink	readlink
readv	-	readv	3	__readv	readv
reboot	-	reboot	1	reboot	
rename	-	rename	2	rename	
rmdir	-	rmdir	1	__rmdir	rmdir
select	-	select	5	__select	select
setdomain	-	setdomainname	2	setdomainname	
setegid	-	setegid	1	__setegid	setegid
seteuid	-	seteuid	1	__seteuid	seteuid
setgid	-	setgid	1	__setgid	setgid
setgroups	-	setgroups	2	setgroups	
setitimer	-	setitimer	3	__setitimer	setitimer
setpriority	-	setpriority	3	setpriority	
setrlimit	-	setrlimit	2	setrlimit	
setsid	-	setsid	0	__setsid	setsid
settimeofday	-	settimeofday	2	__settimeofday	settimeofday
setuid	-	setuid	1	__setuid	setuid
sigsuspend	-	sigsuspend	1	sigsuspend	
sstk	-	sstk	1	sstk	
statfs	-	statfs	2	__statfs	statfs
swapoff	-	swapoff	1	swapoff	
swapon	-	swapon	1	swapon	
symlink	-	symlink	2	__symlink	symlink
sync	-	sync	0	sync	
sys_fstat	fxstat	fstat	2	__syscall_fstat	
sys_mknod	xmknod	mknod	3	__syscall_mknod	
sys_stat	xstat	stat	2	__syscall_stat	
umask	-	umask	1	__umask	umask
uname	-	uname	1	uname	
unlink	-	unlink	1	__unlink	unlink
utimes	-	utimes	2	__utimes	utimes
write	-	write	3	__libc_write	__write write
writew	-	writew	3	__writew	writew

I extraced open.o from libc.a and dumped the open.o

```
use ar -x libc.a, in some temp dir.
```

```
>objdump --show-raw-insn open.o
open.o:      file format elf32-i386
>objdump --disassemble open.o
open.o:      file format elf32-i386
Disassembly of section .text:
00000000 <__libc_open>:
   0:  53                pushl   %ebx
   1:  8b 54 24 10       movl    0x10(%esp,1),%edx
   5:  8b 4c 24 0c       movl    0xc(%esp,1),%ecx
   9:  8b 5c 24 08       movl    0x8(%esp,1),%ebx
  d:  b8 05 00 00 00    movl    $0x5,%eax
 12:  cd 80            int     $0x80
 14:  5b                popl    %ebx
 15:  3d 01 f0 ff ff    cmpl    $0xfffff001,%eax
1a:  0f 83 fc ff ff    jae     1c <__libc_open+0x1c>
 20:  c3                ret
```

How does the glibc build system generate the wrapper call to open()? It happens when the glibc-2.1/io directory is build. This is the output where it happens:

```
make[1]: Entering directory `/export/g/src/packages/SOURCES/glibc-2.1/io'
(echo '#include <sysdep.h>'; \
echo 'PSEUDO (__libc_open, open, 3)'; \
echo ' ret'; \
echo 'PSEUDO_END(__libc_open)'; \
echo 'weak_alias (__libc_open, __open)'; \
echo 'weak_alias (__libc_open, open)'; \
) | gcc -c -I../include -I. -I.. -I../libio -I../sysdeps/i386/elf
-I../crypt/sysdeps/unix -I../linuxthreads/
sysdeps/unix/sysv/linux -I../linuxthreads/sysdeps/pthread
-I../linuxthreads/sysdeps/unix/sysv -I../linuxthreads
/sysdeps/unix -I../linuxthreads/sysdeps/i386/i686 -I../linuxthreads/sysdeps/i386
-I../sysdeps/unix/sysv/linux/i
386/i686 -I../sysdeps/unix/sysv/linux/i386 -I../sysdeps/unix/sysv/linux
-I../sysdeps/gnu -I../sysdeps/unix/comm
on -I../sysdeps/unix/mman -I../sysdeps/unix/inet -I../sysdeps/unix/sysv/i386
-I../sysdeps/unix/sysv -I../sysdeps/unix/i386 -I../sysdeps/unix -I../sysdeps/
posix
-I../sysdeps/i386/i686 -I../sysdeps/i386/i486 -I../sysdeps/lib
m-i387/i686 -I../sysdeps/i386/fpu -I../sysdeps/libm-i387 -I../sysdeps/i386
-I../sysdeps/wordsize-32 -I../sysdeps/ieee754 -I../sysdeps/libm-ieee754
-I../sysdeps/generic/elf -I../sysdeps/generic -D_LIBC_REENTRANT
-include ../include/libc-symbols.h -DASSEMBLER -DGAS_SYNTAX -x assembler-
with-cpp
-o open.o -echo 'io/utime.o io/mkfifo.o io/stat.o io/fstat.o io/lstat.o
io/mknod.o io/stat64.o io/fstat64.o io/lstat64.o io/xstat.o io/fxstat.o
io/lxstat.o io/xmknod.o io/xstat64.o io/fxstat64.o io/lxstat64.o io/statfs.o io/
fstatfs.o
io/statfs64.o io/fstatfs64.o io/statvfs.o io/fstatvfs.o io/statvfs64.o
io/fstatvfs64.o io/umask.o io/chmod.o io/fchmod.o io/mkdir.o io/open.o
io/open64.o io/close.o io/read.o io/write.o io/lseek.o io/lseek64.o io/access.o
io/euidaccess.o io/fcntl.o io/flock.o io/lockf.o io/lockf64.o io/dup.o io/dup2.o
io/pipe.o io/creat.o io/creat64.o io/chdir.o io/fchdir.o io/getcwd.o
io/getwd.o io/getdirname.o io/chown.o io/fchown.o io/lchown.o io/ttyname.o
io/ttyname_r.o io/isatty.o io/link.o io/symlink.o io/readlink.o io/unlink.o
```

```
io/rmdir.o io/ftw.o io/ftw64.o io/fts.o io/poll.o' > stamp.oT
mv -f stamp.oT stamp.o
```

I do not understand the above, as I do not see where is the C source code for the call wrapper. Maybe one day I will understand the above.

But as a result of the above, we get `open.o` in `libc.a`, with the `__libc_open` entry there as an alias for 'open'. OK, now let me look more at the code generated in `__libc_open`.

Here it is again

```
>objdump --disassemble open.o
open.o:      file format elf32-i386
Disassembly of section .text:
00000000 <__libc_open>:
   0:  53                pushl  %ebx
   1:  8b 54 24 10       movl   0x10(%esp,1),%edx
   5:  8b 4c 24 0c       movl   0xc(%esp,1),%ecx
   9:  8b 5c 24 08       movl   0x8(%esp,1),%ebx
  d:  b8 05 00 00 00    movl   $0x5,%eax
 12:  cd 80            int    $0x80
 14:  5b                popl   %ebx
 15:  3d 01 f0 ff ff    cmpl   $0xfffff001,%eax
1a:  0f 83 fc ff ff ff jae    1c <__libc_open+0x1c>
20:  c3                ret
```

Notice that `open()` takes 3 arguments

```
open (filename, posix_mode, prot)
```

Notice the assembler shows using registers `eds`, `ecx`, and `ebx` to pass the data, then it moves 5 to `eax`. What is 5? This got to be the number that kernel uses to identify which system call it is.

Actually this will end up as an index used by the interrupt handler to locate the system call. Lets look around.

```
cd glibc-2.1
>find . -name '*.h' | grep syscall
./include/syscall.h
./misc/syscall.h
./misc/syscall-list.h
./sysdeps/generic/sys/syscall.h
./sysdeps/mach/sys/syscall.h
./sysdeps/unix/sysv/linux/mips/sys/syscall.h
./sysdeps/unix/sysv/linux/sys/syscall.h
./sysdeps/unix/sysv/sco3.2.4/sys/syscall.h
./sysdeps/unix/sysv/sysv4/solaris2/sys/syscall.h
./INSTALL_LIB/usr/local/include/sys/syscall.h
./INSTALL_LIB/usr/local/include/bits/syscall.h
./INSTALL_LIB/usr/local/include/syscall.h

>more ./include/syscall.h
#include <misc/syscall.h>

>more ./misc/syscall.h
#include <sys/syscall.h>
>
```

Ok, getting closer, lets look at `/usr/include/sys/syscall.h`

```
>more /usr/include/sys/syscall.h
```

```
/* Copyright (C) 1995, 1996, 1997 Free Software Foundation, Inc.
   This file is part of the GNU C Library.

   The GNU C Library is free software; you can redistribute it and/or
   modify it under the terms of the GNU Library General Public License as
   published by the Free Software Foundation; either version 2 of the
   License, or (at your option) any later version.

   The GNU C Library is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
   Library General Public License for more details.

   You should have received a copy of the GNU Library General Public
   License along with the GNU C Library; see the file COPYING.LIB. If not,
   write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330,
   Boston, MA 02111-1307, USA. */

#ifndef _SYSCALL_H
#define _SYSCALL_H      1

/* This file should list the numbers of the system the system knows.
   But instead of duplicating this we use the information available
   from the kernel sources. */
#include <asm/unistd.h>

#ifndef _LIBC
/* The Linux kernel header file defines macros `__NR_<name>', but some
   programs expect the traditional form `SYS_<name>'. So in building libc
   we scan the kernel's list and produce <bits/syscall.h> with macros for
   all the `SYS_' names. */
# include <bits/syscall.h>
#endif

#endif
```

Ok, I am getting really close now.

```
>more /usr/include/asm/unistd.h
```

```
#ifndef _ASM_I386_UNISTD_H_
#define _ASM_I386_UNISTD_H_

/*
 * This file contains the system call numbers.
 */

#define __NR_exit          1
#define __NR_fork          2
#define __NR_read          3
#define __NR_write         4
#define __NR_open          5 /*-----> HERE IT IS !!!*/
....
```

yahoo! found it. So, 5 is moved to register eax, and interrupt 0x80 is invoked.

When interrupt returns, system call is complete. It does not seem that the syscall macros

defined in `/usr/include/asm/unistd.h` are used in glibc?

OK, so far so good, now I'll switch hats, and jump into kernel mode to see how the `open()` call is processed. I need to find the code for that processes the interrupt 0x80.

The interrupt routine that is bound to interrupt 0x80 is found in

`/usr/src/linux/arch/i386/kernel/entry.S`

the entry point is called `ENTRY(system_call)`.

Lets look at the code for the interrupt routine:

```
ENTRY(system_call)
    pushl %eax                # save orig_eax
    SAVE_ALL
    GET_CURRENT(%ebx)
    cmpl $(NR_syscalls),%eax  -----> Notice, eax is where the system
    call number is saved.
    jae badsys
    testb $0x20,flags(%ebx)   # PF_TRACESYS
    jne tracesys
    call *SYMBOL_NAME(sys_call_table)(,%eax,4) -----> Here we index into the
    sys_call_table using the above number.
    movl %eax,EAX(%esp)       # save the return value
ENTRY(ret_from_sys_call)
#ifdef __SMP__
    movl processor(%ebx),%eax
    shll $5,%eax
    movl SYMBOL_NAME(softirq_state)(,%eax),%ecx
    testl SYMBOL_NAME(softirq_state)+4(,%eax),%ecx
#else
    movl SYMBOL_NAME(softirq_state),%ecx
    testl SYMBOL_NAME(softirq_state)+4,%ecx
#endif
    jne    handle_softirq

ret_with_reschedule:
    cmpl $0,need_resched(%ebx)
    jne reschedule
    cmpl $0,sigpending(%ebx)
    jne signal_return
restore_all:
    RESTORE_ALL

    ALIGN
signal_return:
    sti                # we can get here from an interrupt
    handler
    testl $(VM_MASK),EFLAGS(%esp)
    movl %esp,%eax
    jne v86_signal_return
    xorl %edx,%edx
    call SYMBOL_NAME(do_signal)
    jmp restore_all

    ALIGN
v86_signal_return:
    call SYMBOL_NAME(save_v86_state)
    movl %eax,%esp
    xorl %edx,%edx
```

```

        call SYMBOL_NAME(do_signal)
        jmp restore_all

        ALIGN
tracesys:
        movl $-ENOSYS,EAX(%esp)
        call SYMBOL_NAME(syscall_trace)
        movl ORIG_EAX(%esp),%eax
        cmpl $(NR_syscalls),%eax
        jae tracesys_exit
        call *SYMBOL_NAME(sys_call_table)(,%eax,4)
        movl %eax,EAX(%esp)          # save the return value
tracesys_exit:
        call SYMBOL_NAME(syscall_trace)
        jmp ret_from_sys_call
badsys:
        movl $-ENOSYS,EAX(%esp)
        jmp ret_from_sys_call

        ALIGN
ret_from_exception:
#ifdef __SMP__
        GET_CURRENT(%ebx)
        movl processor(%ebx),%eax
        shll $5,%eax
        movl SYMBOL_NAME(softirq_state)(,%eax),%ecx
        testl SYMBOL_NAME(softirq_state)+4(,%eax),%ecx
#else
        movl SYMBOL_NAME(softirq_state),%ecx
        testl SYMBOL_NAME(softirq_state)+4,%ecx
#endif
        jne    handle_softirq

ENTRY(ret_from_intr)
        GET_CURRENT(%ebx)
        movl EFLAGS(%esp),%eax      # mix EFLAGS and CS
        movb CS(%esp),%al
        testl $(VM_MASK | 3),%eax   # return to VM86 mode or non-supervisor?
        jne ret_with_reschedule
        jmp restore_all

        ALIGN
handle_softirq:
        call SYMBOL_NAME(do_softirq)
        jmp ret_from_intr

        ALIGN
reschedule:
        call SYMBOL_NAME(schedule)  # test
        jmp ret_from_sys_call

ENTRY(divide_error)
        pushl $0                    # no error code
        pushl $ SYMBOL_NAME(do_divide_error)
        ALIGN
error_code:
        pushl %ds
        pushl %eax

```

```

xorl %eax,%eax
pushl %ebp
pushl %edi
pushl %esi
pushl %edx
decl %eax                # eax = -1
pushl %ecx
pushl %ebx
cld
movl %es,%ecx
xchgl %eax, ORIG_EAX(%esp) # orig_eax (get the error code. )
movl %esp,%edx
xchgl %ecx, ES(%esp)      # get the address and save es.
pushl %eax                # push the error code
pushl %edx
movl $(__KERNEL_DS),%edx
movl %edx,%ds
movl %edx,%es
GET_CURRENT(%ebx)
call *%ecx
addl $8,%esp
jmp ret_from_exception

```

The `sys_call_table` itself is located in `.data` segment in `entry.S`, this is the start of the table

```

.data
ENTRY(sys_call_table)
    .long SYMBOL_NAME(sys_ni_syscall)        /* 0 - old "setup()" system
    call*/
    .long SYMBOL_NAME(sys_exit)
    .long SYMBOL_NAME(sys_fork)
    .long SYMBOL_NAME(sys_read)
    .long SYMBOL_NAME(sys_write)
    .long SYMBOL_NAME(sys_open)              /* 5 */
    .long SYMBOL_NAME(sys_mincore)
    .long SYMBOL_NAME(sys_madvise)

.....

/*
 * NOTE!! This doesn't have to be exact - we just have
 * to make sure we have _enough_ of the "sys_ni_syscall"
 * entries. Don't panic if you notice that this hasn't
 * been shrunk every time we add a new system call.
 */
    .rept NR_syscalls-219
        .long SYMBOL_NAME(sys_ni_syscall)
    .endr

```

Ok, lets follow the system call. I see from the dispatch table above, that the `open()` call is implemented in kernel using `sys_open`.

Where is `sys_open()` ? All the sys calls related to IO are located in `linux/fs/`. Looking at `linux/fs/open.c`, this is the `sys_open` function.

```

asmlinkage long sys_open(const char * filename, int flags, int mode)
{

```

```

    char * tmp;
    int fd, error;

#if BITS_PER_LONG != 32
    flags |= O_LARGEFILE;
#endif

    tmp = getname(filename);
    fd = PTR_ERR(tmp);
    if (!IS_ERR(tmp)) {
        fd = get_unused_fd();
        if (fd >= 0) {
            struct file * f;
            lock_kernel();
            f = filp_open(tmp, flags, mode);
            unlock_kernel();
            error = PTR_ERR(f);
            if (IS_ERR(f))
                goto out_error;
            fd_install(fd, f);
        }
    }

out:
    putname(tmp);
}
return fd;

out_error:
    put_unused_fd(fd);
    fd = error;
    goto out;
}
\begin{Cinline}
%
The function \verb|filp_open()| is in the same above file as \verb|sys_open()|.
Here is the function
%
\begin{Cinline}
struct file *filp_open(const char * filename, int flags, int mode)
{
    int namei_flags, error;
    struct nameidata nd;

    namei_flags = flags;
    if ((namei_flags+1) & O_ACCMODE)
        namei_flags++;
    if (namei_flags & O_TRUNC)
        namei_flags |= 2;

    error = open_namei(filename, namei_flags, mode, &nd);
    if (!error)
        return dentry_open(nd.dentry, nd.mnt, flags);

    return ERR_PTR(error);
}

```

Notice the call to `open_namei()`, this is the interface to the virtual file system. calls into VFS are named `_namei` (verify?).

open_namei() is defined in linux/fs/namei.c.

After some access checking, and pathname checking, and possibly allocating an inode, a kernel internal struct file is allocated for the file. The file struct contains a pointer to file_operations struct, which contains the address of functions to process operations on this filesystem, that must have been initialized when the file system was mounted.

```
struct file {
456     struct list_head      f_list;
457     struct dentry          *f_dentry;
458     struct vfsmount        *f_vfsmnt;
459     struct file_operations *f_op;
460     atomic_t               f_count;
461     unsigned int           f_flags;
462     mode_t                 f_mode;
463     loff_t                 f_pos;
464     unsigned long          f_reada, f_ramax, f_raend, f_ralen, f_rawin;
465     struct fown_struct      f_owner;
466     unsigned int           f_uid, f_gid;
467     int                    f_error;
468
469     unsigned long          f_version;
470
471     /* needed for tty driver, and maybe others */
472     void                   *private_data;
473 };

\begin{Cinline}%

and

\begin{Cinline}
struct file_operations {
693     loff_t (*llseek) (struct file *, loff_t, int);
694     ssize_t (*read) (struct file *, char *, size_t, loff_t *);
695     ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
696     int (*readdir) (struct file *, void *, filldir_t);
697     unsigned int (*poll) (struct file *, struct poll_table_struct *);
698     int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
699     int (*mmap) (struct file *, struct vm_area_struct *);
700     int (*open) (struct inode *, struct file *);
701     int (*flush) (struct file *);
702     int (*release) (struct inode *, struct file *);
703     int (*fsync) (struct file *, struct dentry *);
704     int (*fasync) (int, struct file *, int);
705     int (*lock) (struct file *, int, struct file_lock *);
706     ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
707     ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
708 };
```

Ok, time to go sleep now.