568

JOHN R. GRAHAM

SOLARIS 2.X

INTERNALS & ARCHITECTURE

DISK Included

Solaris 2.X

Internals and Architecture

John R. Graham

McGraw-Hill, Inc.
New York San Francisco Washington, D.C. Auckland Bogotá
Caracas Lisbon London Madrid Mexico City Milan
Montreal New Delhi San Juan Singapore
Sydney Tokyo Toronto

Library of Congress Cataloging-in-Publication Data

Graham, John R.
Solaris 2.x : internals and architecture / by John R. Graham.
p. cm.
Includes index.
ISBN 0-07-911876-3
1. Operating systems (Computers) 2. Solaris (Computer file)
I. Title.
QA76.76.O63G72 1995
005.4'469—dc20 95-10087
CIP

pbk 1 2 3 4 5 6 7 8 9 0 DOC/DOC 9 9 8 7 6 5

ISBN 0-07-911876-3

The sponsoring editor for this book was Gerald Papke. The book editor was Jim Gallant, and the managing editor was Susan W. Kagey. The director of production was Katherine G. Brown. This book was set in ITC Century Light. It was composed by TAB Books.

Printed and bound by R. R. Donnelley & Sons Company of Crawfordsville, Indiana.

MH95
9118763

- The trap handler will invoke the routine **syscall( )** to handle the system call.
- When the system call returns, the wrapper examines the registers for return values and returns to the user.

The real work of the system call is done in the internal routine **syscall( )**. **Syscall** is called with two arguments: the trap type and a copy of the registers. For system calls, the trap type should be type 0 (**ST_OSYSCALL**) or type 8 (**ST_SYSCALL**). These trap types are defined in **/usr/include/sys/trap.h**. The only time type 0 will be used is when trying to run an old SunOS 4.x (BSD) type program. Trap 0 is used to indicate that the system call number is old and will have to be remapped to the new system call number. Trap type 8 is the usual way SunOS 5.x calls **syscall( )**.

System Call Number

The system call number is an index into the system call entry table, the sysent array, for short. The actual table is stored at the kernel symbol **sysent[ ]**. The table is an array of struct sysent. The declaration of a struct sysent can be found in **/usr/include/sys/systm.h**

```
struct sysent {
        char sy_narg;          /* number of arguments *
        char sy_flags;         /* flags */
        int (*sy_call)();      /* the actual function address */
        krwlock_t sy_lock;     /* lock for loadable calls */
}
```


The system call number assigned to a particular system call can be found in **/usr/include/sys/syscall.h**.[8] When the wrapper for a system call is entered, the system call number, or index into the sysent array, is placed in one of the registers and extracted by the **syscall( )** routine to fetch the arguments and invoke the code for the system call. The pseudo-code for the **syscall( )** routine follows.

```
syscall ( type,rp) {
   fetch the address of the code from the sysent[] entry;
   fetch the number of arguments from the sysent[] entry;
   copy the correct number of arguments from the user stack frame;
   make the actual call;
   Check for errors;
   if error was due to interrupt or signal {
          check restart flag to restart call;
           - OR -
          return error (EINTR);
   else /* some other error */
          set return value and error;
   }
   check for signals and process (ISSIG);
   check for preempts (cpu_runrun);
}
```

[8]There is also a file, **/etc/name_to_sysnum**, that is used to map system calls to system call numbers. Code for a sample system call module and the modifications needed for this file are shown in Appendix B.

2. The search will continue using the lookup routine in the vnode for **/usr**. Again, the lookup routine will discover the next component of the path **/usr/openwin** is a mount point.
3. Following the **v_vfsmountedhere** pointer, the search will continue on the remote file system.
4. The search on the remote file system is successful and a vnode is created on the local system representing **/usr/openwin/fubar**.

A variation on this search is when the file name starts with "**..**" as is **../home/fileb**. The search mechanism is similar except when the "**..**" is encountered in the path. If we are currently looking at the root of a mounted file system, the "**..**" will mean we have to go to the parent of the file system. We will know this is a root for a file system because the flags field of the vnode will be set to VROOT (see **vnode.h**). In this case, follow the **v_vfsp** pointer to continue the search in the parent file system.

Local Structures and Links

So far we have seen how the kernel manages files and file systems. The real objective with an **open(2)** call, is to create a local access to the file. The return value from the **open(2)** is a file descriptor. In the case of **fopen(3)**, a file pointer is returned that is a pointer to a file descriptor. This section will examine how a file descriptor at the local level is linked into the kernel structures we have just examined.

File Descriptors

The first structure of interest is the file descriptor structure. When a file is opened, a file descriptor is returned to the user. The file descriptor is of type **int** and is used as an index into a table of open file descriptors. The table is stored in the user area portion of the proc structure. In previous releases, the file descriptors were stored in a statically sized table within the user area. The problem with this was that the table could get full and the user could not open more files. In SunOS 5.x, this restriction no longer applies. File descriptors are now allocated in *chunks* of 24 (**#define NFPCHUNK 24 /* <sys/user.h> */**). The file descriptor table is a list of *chunks* linked together. The start of this list is pointed to by the **u_file** field in the user structure. Each entry in a chunk is a field of type struct file that is defined in **/usr/include/sys/file.h**. The first 24 file descriptors are stored in the user area itself. If there are more than 24, the **uf_next** pointer in the ufchunk structure is used to find the next chunk.

Following is a partial listing of **file.h**:

```
struct ufchunk {
        struct file *uf_ofile[NFPCHUNK];
        char uf_pofile[NFPCHUNK];
        struct ufchunk *uf_next;
};

/*
 * One file structure is allocated for each open/creat/pipe call.
 * Main use is to hold the read/write pointer associated with
```

The oper

```
 * each open file.
 */
typedef struct file {
      struct file     *f_next;        /* pointer to next entry */
      struct file     *f_prev;        /* pointer to previous entry */
      ushort_t        f_flag;
      cnt_t           f_count;        /* reference count */
      struct vnode    *f_vnode;       /* pointer to vnode structure */
      offset_t        f_offset;       /* read/write character pointer */
      struct cred     *f_cred;        /* credentials of user who opened it */
      caddr_t         f_audit_data;   /* file audit data */
      kmutex_t        f_tlock;        /* short term lock */
      kcondvar_t      f_done;
      int             f_refcnt;
} file_t;
```

The **file_t** structures are allocated as needed in kernel memory. For the purposes of our discussion, the most important fields in the file structure are:

- **f_next** and **f_prev**, pointers to the next and previous file descriptors.
- **f_offset**, an offset in bytes from the beginning of the file where the next read or write will take place. The offset will change each time a read or write takes place or when the **lseek(2)** call is used.
- **f_vnode**, a vnode pointer that completely describes the open file.
- **f_count**, a count that is incremented when using the **dup(2)** or **dup2(3)** call and decremented when using a **close(2)** call.

Figure 14.4 illustrates file descriptor components.

There is another advantage to allocating file descriptors in this manner. Under previous releases (before 5.x), the total number of files open by all processes at a given time was limited to the size of a static table known as the *System Open File Table* (SOFT). The SOFT was a table of file structures. With SunOS 5.x, there is no static table and the number of file structures can grow dynamically as needed. Since the file structures are allocated in kernel virtual memory, the only limit is the size of kernel virtual memory not in use, a very large number.

The open( ) System Call

With all of the pieces in place, we can now examine how the open system call uses the pieces to do its job. The following steps are completed in opening a file:

- Allocate an entry in the local file descriptor table.
- Allocate an entry for the file in the ufchunk structure.
- Using the lookup scheme described earlier, search the vfs and vnode structure until the file to be opened is located.
- Allocate a vnode and point to it through the file structure.
- Using the **v_op** routines verify permission (**VOP_ACCESS**) and then open (**VOP_OPEN**) the file.
- Return a file descriptor index to the user.