Overview

Over the next four weeks, we will look at these topics:

☞ Building Blocks

☞ Advanced Authentication Issues

☞ Security

Overview

- ☞ Storage and its abstraction
- ☞ Virtualization and appliances
- ☞ Data Replication and Disaster Recovery (BCP/COOP)
- ☞ Physical Environments

At the system level: Unix/Linux Building blocks

I want to take some time to talk about one of the fundamental toolsets that most programs that system administrators work with are built over.

At the system level: Unix/Linux Building blocks

In the Unix/Linux world, the most important of these are the system calls. When we run a program like `strace` to see exactly what a Unix/Linux process is doing, we are watching this fundamental interaction between a program and its requests to the kernel, usually for access to resources controlled by the operating system.

Requests

A Unix/Linux system call is a direct request to the kernel regarding a system resource. It might be a request for a file descriptor to manipulate a file, it might be a request to write to a file descriptor, or any of hundreds of possible operations.

Requests

These are exactly the tools that every Unix/Linux program is built upon. Even the simplest of all programs, `/bin/true` and `/bin/false`, which do nothing except provide a process return code, must make at least one system call to exit(2).

File systems as a mainstay

In some sense, the mainstay operations are those on the file system.

File systems as a “permanent” resource

Unlike many other resources which are just artifacts of the operating system and disappear at each reboot, changing a file system generally is an operation that has some permanence. Filesystems have a naming system (we call these “pathnames” in the Unix/Linux world), and it is possible to explicitly operate on elements of a filesystem via these names. We generally expect these files and directories to persist even over reboots of a system.

File systems as a “permanent” resource

Of course it is possible and even common to create “RAM” disk filesystems since they are quite fast and for items that are meant to be temporary, they are quite acceptable. As I mentioned with MailScanner setup, it is recommended to put `/var/spool/incoming` on a RAM disk since the information that it holds is inherently transient.

File systems as a “permanent” resource

But in the everyday world of system administration, the rule is that filesystems have some permanence, and it’s the exception for it to be merely a RAM disk.

Filesystem operations versus file descriptor operations

While many of the most frequent operations happen on file descriptors, file descriptors are indirect references, and with the exception of operations such as `dup(2)`, are not created from other file descriptors.

Filesystem operations

Unix/Linux offers a number of direct functions on files referred to by their names inside of a filesystem. These are important operations, such as creating a file descriptor associated with a file of a given name, where we want to be able to refer to a file or directory by name.

Fundamental filesystem operations

```
open()     -- create a new file descriptor to access a file
close()    -- deallocate a file descriptor

access()   -- returns a value indicating if a file is accessible
chmod()    -- changes the permissions on a file in a filesystem
chown()    -- changes the ownership of a file in a filesystem
```

Important filesystem operations

```
link()     -- create a hard link to a file
symlink() -- create a soft link to a file
```

Important filesystem operations

```
mkdir()    -- create a new directory
rmdir()    -- remove a directory
```

Important filesystem operations

```
stat()    -- return information about a file associated with a fd:
          -- inode, perms, hard links, uid, gid, size, modtimes
statfs()  -- return the mount information for the filesystem that
          -- the file descriptor is associated with
```

Important file descriptor calls

A file descriptor is an `int`. It provides stateful access to an i/o resource such as a file on a filesystem, a pseudo-terminal, or a socket to a tcp session.

```
open()     -- create a new file descriptor to access a file
close()    -- deallocate a file descriptor
```

Important file descriptor calls

```
dup()       -- duplicate a file descriptor
dup2()      -- improved way to duplicate a file descriptor
```

Important file descriptor calls

```
fchmod()   -- change the permissions of a file associated with a file
           -- descriptor
fchown()   -- change the ownership of a file associated with a file
fchdir()   -- change the working directory for a process via fd
```

Important file descriptor calls

These two are "kitchen sink" calls: they do a wide miscellanea of operations.

```
fcntl()     -- miscellaneous manipulation of file descriptors:
            -- duplication like dup(), set close on exec(),
            -- set to non-blocking, set to asynchronous mode,
            -- locks, signals
ioctl()     -- manipulate the underlying ``device'' parameters for
            -- a file descriptor
```

Important file descriptor calls

```
flock()    -- lock/unlock a file associated with a file descriptor
```

Important file descriptor calls

```
pipe()      -- create a one-way association between two file
            -- descriptors so that output from
            -- one goes to the input of the other
            -- very important for simple ipc such as shells
            -- provide between processes
```

Important file descriptor calls

```
select()   -- multiplex on pending i/o to or from a set of file
           -- descriptors
```

Important file descriptor calls

```
read()     -- send data to a file descriptor
write()    -- take data from a file descriptor
fsync()    -- forces a flush for a file descriptor
```

Important file descriptor calls

```
readdir() -- older ``raw'' read of directory entry from a
          -- file descriptor
getdents() -- more recent ``raw'' read of multiple directory
           -- entries from a file descriptor into a buffer
```

Important file descriptor calls

```
fstat()    -- return information about a file associated with a fd:
           -- inode, perms, hard links, uid, gid, size, modtimes
fstatfs()  -- return the mount information for the filesystem
           -- that the file descriptor is associated with
```

Signals

Unix also supports signals, a very simple IPC mechanism. These signals, while they do not carry a payload of information, are distinguishable by their number. The most important of these are SIGKILL, SIGTERM, SIGHUP, and SIGALRM, but there are usually many more, even ones reserved to user processes such as SIGUSR1 and SIGUSR2.

Arbitrary signalling

```
kill          -- send an arbitrary signal to an arbitrary process
killpg        -- send an arbitrary signal to all processes in a
              -- process group
```

Signals

```
sigaction    -- interpose a signal handler (can include special
             -- ``default''  and ``ignore'' handlers)
sigprocmask -- change the list of blocked signals
```

Signals

```
alarm        -- set an alarm clock for a SIGALRM to be sent to
             -- a process time measured in seconds
setitimer    -- set an alarm clock in fractions of a second to
             -- deliver one of SIGALRM, SIGVTALRM, or SIGPROF
```

Signals

```
wait          -- check for a signal (can be blocking or non-blocking)
              -- or child exiting
waitpid       -- check for a signal from a child process
              -- (can be general or specific)
```

Modifying the current process's state

Many system calls exist to change the current process's state.

```
chdir        -- change the working directory for a process to dirname
chroot       -- change the root filesystem for a process
```

Modifying the current process's state

```
execve      -- execute another binary in this current process
fork        -- create a new child process running the same binary
clone       -- allows sharing of execution context (unlike fork(2))
exit        -- terminate the current process
```

Modifying the current process's state

```
getdtablesize  -- report how many file descriptors this process
               -- can have active simultaneously (see select()
               -- for why this is useful)
```

Finding the current process's state

```
getgid      -- return the group id of this process
getuid      -- return the user id of this process
getpgid     -- return process group id of this process
getpgrp     -- return process group's group of this process
```

Finding the current process's state

```
getpid       -- return the process id of this process
getppid      -- return parent process id of this process
getrlimit    -- get resource limits on this process (core size, cpu time,
             -- data size, stack size, and others)
getrusage    -- find amount of resource usage by this process
```

Modifying the current process's state

```
nice()              -- change the calling process's priority
setpriority()       -- arbitrarily change any process's (or group or user)
                    -- priority
getpriority()       -- get any process's priorities
setrusage           -- set maxima for resource utilization by the
                    -- current process
```

Communications and Networking

```
socket      -- create a file descriptor (can be either network or local)

bind        -- bind a file descriptor to an address, such as a tcp port
listen      -- specify willingness for some number of connections to be
            -- blocked waiting on accept()
accept      -- block until there is a new connection

connect     -- actively connect to listen()ing socket
```

Communications and Networking

```
setsockopt  -- set options on a given socket associated with fd,
            -- such as for out-of-band data, keep-alive information,
            -- congestion notification, final timeout, and so forth
            -- (see man tcp(7))
getsockopt  -- retrieve information about options enabled for a given
            -- connection from fd
```

Communications and Networking

```
getpeername -- retrieve information about other side of a
            -- connection from fd
getsockname -- retrieve information this side of a
            -- connection from fd
```

Others

```
brk           -- allocate memory for the data segment for the
              -- current process. in Linux, the first call to brk
              -- actually creates the heap; the second and subsequent
              -- calls do allocation

gethostname   -- gets a ``canonical hostname'' for the machine
gettimeofday -- gets the time of day for the whole machine
settimeofday -- sets the time of day for the whole machine
mount         -- attaches a filesystem to a directory and makes
              -- it available


sync          -- flushes all filesystem buffers, forcing changed
              -- blocks to ``drives'' and updates superblocks
```

```
futex          -- raw locking (lets a process block waiting on a change
                  to a specific memory location)
```

Others

```
sysinfo       -- provides direct access from the kernel to:
                    load average
                    total ram for system
                    available ram
                    amount of shared memory existing
                    amount of memory used by buffers
                    total swap space
                    swap space available
                    number of processes currently in proctable
```

SYS V IPC

```
msgctl      -- SYS V messaging control (uid, gid, perms, size)
msgget      -- SYS V message queue creation/access
msgrcv      -- receive a SYS V message
msgsnd      -- send a SYS V message

shmat       -- attach memory location to SYS V shared memory segment
shmctl      -- SYS V shared memory contrl (uid, gid, perms, size, etc)
shmget      -- SYS V shared memory creation/access
shmdt       -- detach from SYS V shared memory segment
```