Pathname Resolution with a Bit on Union Mountpoints

Authored by: Thomas Fletcher
Updated by: Thomas Fletcher

Like previous QNX operating systems, Neutrino is a microkernel-based, message-passing system. The OS itself is generally built up from a number of separate server processes that communicate to each other via message passing.

In a Neutrino system, servers create channels (via **ChannelCreate**()). Clients then connect to these channels (via **ConnectAttach**()) before they can send messages.

The **ConnectAttach**() call requires a node ID (**nid**), process ID (**pid**), and channel ID (**chid**). The call then returns a connection ID (**coid**), which can then be used either directly with **MsgSend**() or indirectly as a file descriptor (**fd**) with functions such as **read**() and **write**().

The pathname space

At the heart of any message communication system is the ability to locate and connect to a service by some symbolic name. In Neutrino, one way to do this conversion from a name to a connection ID is through the pathname space.

The task of managing names/entries in the pathname space is distributed among several parties in Neutrino:

- The process manager (Proc) manages the mountpoints in the system, which are generally created with **resmgr_attach**().
- Individual servers are responsible for pathname/entry validation below their mountpoints. This applies only to servers that attach a pathname with the **RESMGR_FLAG_DIR** attribute.
- The client (indirectly through the C Library) is responsible for managing "traffic". It provides the glue for translating the server/mountpoint responses from Proc to requests targeting the individual servers.

Clear as mud? An example will hopefully help here. Let's say we have three servers:

- Server A - a single device that generates numbers. Its mountpoint is */dev/random*.
- Server B - a flash filesystem. Its mountpoint is */bin*. It contains *ls* and *echo*.
- Server C - a QNX 4 filesystem. Its mountpoint is */*. It contains *bin/true* and *bin/false*.

Mountpoint	Server
/	Server C (filesystem)
/bin	Server B (filesystem)
/dev/random	Server A (device)

At this point, the process manager's internal mount table would look like this:

```
Mountpoint Server
/ Server C (filesystem)
/bin Server B (filesystem)
/dev/random Server A (device)
```

Of course, each "Server" name is actually an abbreviation for the **nid**,**pid**,**chid** for that particular server channel.

Now, suppose our client wants to send a message to Server A. The client code will look like this:

```
    int fd;
    fd = open("/dev/random", ...);
    read(fd, ...);
    close(fd);
```

In this case, the C library will request from Proc the servers that could potentially handle the path */dev/random*. Proc will return a list of servers:

- Server A (most likely; longest path match)
- Server C (least likely; shortest path match)

From this information, the C library will then contact each server in turn and send it an "**open**" message, including the component of the path that the server should validate:

- Server A receives path ""; the request came in on the same path as the mountpoint. Server C (if Server A denies the request) receives path "*dev/random*", since its mountpoint was "*/*".

What is important to note here is that as soon as one server positively acknowledges the request, the remaining servers are not contacted.

This is fairly straightforward with single device entries, where the first server is generally the server that will handle the request. Where it becomes interesting is in the case of unioned filesystem mountpoints.

Unioned filesystem mountpoints

In a typical UNIX system, when the filesystems are mounted as above, you might have one of two configurations:

- Server B mounted after Server A (most likely):
  /
  /bin
  /bin/ls
  /bin/echo
- Server A mounted after Server B:
  /
  /bin
  /bin/true
  /bin/false

In both cases, only one of the filesystems provides a listing for the contents of the directory /bin, even though both filesystems have entries for this directory.

Under a Neutrino system, you would see the following due to the unioning of the mountpoints:

> */*
> */bin*
> */bin/ls*
> */bin/echo*
> */bin/true*
> */bin/false*

What's happening here is that the resolution for the path */bin* takes place as before, but rather than limit the return to just one connection ID, all the servers are contacted and asked about their handling for the path:

```
    DIR *dirp;
    dirp = opendir("/bin", ...);
    closedir(dirp);
```

which results in:

- Server B receives path ""; the request came in on the same path as the mountpoint.
- Server C receives path "*bin*", since its mountpoint was "*/*".

The result now is that we have a collection of file descriptors to servers who handle the path */bin* (in this case two servers); the actual directory name entries are read in turn when a **readdir**() is called. If any of the names in the directory are accessed with a regular open, then the normal resolution procedure takes place and only one server is accessed.

Why overlay mountpoints?

This overlaying of mountpoints is a very handy feature for things like field updates and servicing. It also makes for a more unified system, where pathnames result in connections to servers regardless of what services they're providing, giving us a more unified API.

How to see a server's mountpoints

You can cut through the layers and take a look at the services offered by one single server by exploring the */proc/mount* directory. While somewhat cryptic, this method lets you see which processes have mounted to which locations. And in the case of filesystems, you can access those filesystems without the unioning of the path entries.

The numbered entries correspond to the **nid**,**pid**,**chid**,**handle**,**ftype** entries . . . just in case you were wondering. =;-)