Pathname Space and Search

Everyone knows that in a client-server system like QNX, the server (resource manager) provides services by registering a pathname. So, when a client gives out a name, how does the operating system find the corresponding server? This is actually very deep, let's dive in.

Pathname Manager (pathmgr)

First of all, registration, search, and cancellation of names are not managed by the kernel. Yes, you're right, it's not in the kernel. Instead, it's managed by a resource manager called pathmgr. Pathmgr manages the 'pathname space' resource, which is essentially the root directory "/".

Indeed, pathmgr is part of the procnto process, but it's not running in kernel mode.

> procnto is QNX's first process, but that doesn't mean it's a microkernel. The procnto process is actually a combination of memory manager, process manager, and pathname manager. These external managers will definitely used in ordinary QNX system. However, in extreme cases, QNX can also achieve a system that only uses the microkernel and customized processes, like early VxWorks systems. But such a system is equivalent to completely canceling QNX's advantages, so although it's technically possible, it's not popular.

Alright, let's go back to our file name search. The simplest way is to register a resource manager with '*/dev/md5*', and then the client program can *open('/dev/md5')*. You might wonder what's so complicated about this?

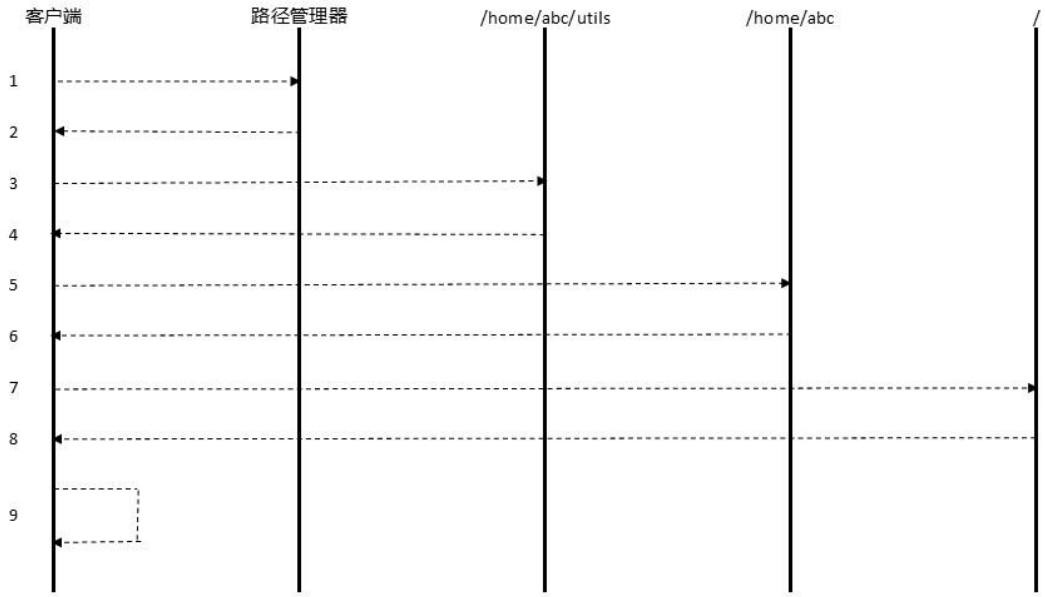
However, the resource manager not only can registers a file, but also a directory. The easiest thing to think of is the file system. When you '*mount a certain hard disk partition to /home*', the file system resource manager will register '*/home/*', meaning that '/home/' and all its subdirectories are under the resource manager's control from now on. If you then run '*ls -l /home/user/.bashrc*', it's actually sending the entire path name to the

registered resource manager, which will go to the actual hard disk partition to find if there is a '.bashrc' file, and then return the status.

If the resource manager can register a subdirectory, things suddenly become interesting. For example, manager A registers '*/home/abc*', and manager B registers '*/home/abc/utils*'. Then when the client *opens('/home/abc/utils/readme')*, which manager will serve the client?

This process of searching for the corresponding resource manager based on the file name is called path name search. It's basically letting the client contact each possible manager, and then the manager returns to the client.

Let's take a look at the timeline diagram



1. The client opens ('/home/abc/utils/readme') in the libc open function, where the first step is to send the entire path name to the path manager (pathmgr). The path manager searches for registered resource managers based on the input path name and returns *a list of all possible service providers* for the path '/home/abc/utils/readme'. In our example, this list includes the '/home/abc/utils' manager, the '/home/abc' manager, and the '/' manager.

2. The open function then sends an io_open message to the '*/home/abc/utils*' resource manager in the list, which contains the relative path name (*'readme'*). This means 'I want to open the readme file. Are you responsible for it?'
3. The resource manager judges whether it is responsible for providing services based on the input path name and its own resources. If so, it can call the io_open callback function to handle the result and end the open function; otherwise, if the resource manager determines that the path name in io_open is not under its responsibility, it needs to return ENOENT (POSIX defines this error as 'file not found'). In our example, we assume the resource manager returns ENOENT.
4. After the client-side open() function receives ENOENT from the first resource manager, it knows that the first resource manager does not manage the path "/home/abc/utils/readme", so it will send an io_open message to the second resource manager (/home/abc) with a relative path ("utils/readme")
   Assume the second resource manager also returns ENOENT
5. The client-side open() function sends a query request to the third resource manager for the relative path (home/abc/utils/readme), asking if the third manager recognizes this name.
6. If the third resource manager still returns ENOENT
7. At this point, the client-side open() function has searched all possible resource managers for the path "/home/abc/utils/readme" and found none that can provide service, so it finally writes ENOENT to errno and returns -1 to indicate that the open() call failed.

This process may seems long, but note that in steps 4, 6, and 8, the path name search will only continue if the resource manager returns ENOENT; if the resource manager returns EOK, it means 'yes, I manage this, and I'm ready' and the path name search ends. There is also a situation where the resource manager returns another error. EPERM, that means 'yes, I manage this name, but you don't have the permission'; at this time, the path name search will also end and return this error directly to the user.
This design sends a path name request to a series of resource managers in sequence. The consideration behind this is that in normal cases, 'finding a file' is the majority scenario, while 'not finding it' is an error scenario that should be relatively rare; under the premise of 'finding it', how can we better arrange the list of resource managers

before and after to let the client quickly hit the correct resource manager, thereby reducing unnecessary message transmission?

QNX's path name manager has a 'longest match first' rule. Taking the above example, for the target path "/home/abc/utils/readme", there are three possible resource managers, which matches the registered path length, that is "/home/abc/utils/" ranks first, and " /home/abc" has a shorter match, so it ranks second.
If there are two managers registered with the exact same path name, then the one that registers first will be placed in front (unless the later-registered manager uses a special flag).

Let’s see some examples of path name management under QNX's unique features.

Virtual directory name

A resource manager can register a directory and then any file operation on that directory by clients will be received by this manager, so it is completely possible to provide a comprehensive service using a virtual directory name. This is also available in other Unix systems, but QNX is extremely flexible because the name no longer depends on inode, so resmgr_attach() can simply register a name (This is why on QNX, mount does not need to prepare a mountpoint beforehand). For example, you can write a resource manager that registers the */car/hvac/* directory. And *'ls /car/hvac'* is just performing opendir(), readdir(), and closedir() operations on your resource manager. As long as your server returns correct data, it can make clients 'look like' you have many subdirectories:

/car/hvac/ac/onoff
/car/hvac/ac/temperature
/car/hvac/heat/onoff
/car/hvac/fans/speed ...

These files' read-write operations will all be aggregated to your resource manager, and according to the relative directory entered, you can also correctly judge whether the user wants to operate on "*fans/speed*" or "*ac/onoff*".

Similarly, you can use a resource manager to register /proc/sys/, then provide a Linux-like /proc/sys/ service.

Alternatively, you can register */mirror/*; then handle requests like *ls -l /mirror/ftp.sjtu.edu.cn/ubuntu-cd/* and implement a file system based on FTP.

The concept of union directories is also unique to QNX. For example, there are two hard disk partitions, one with directories "b/" and "d/", and the other with directories "a/" and "c/", which can be separately mounted to /home1; /home2; to form a unified directory structure.

/home1/b/
/home1/d/
/home2/a/
/home2/c/

However, if you mount them all to /home afterwards, you will get:

/home/a/
/home/b/
/home/c/
/home/d/

In other words, opendir()/readdir() operations are not just finding the first server and stopping. This is actually what most people would expect.

What's the problem when two hard drives are merged, and both have a file called '/readme'? What will happen then? Should there be two of them?

/home/readme
/home/readme

Theoretically, it should be like this, but in reality, when there are duplicate files in the same directory, QNX will only show one. Another thing that often confuses users is that if you rm /home/reamde, and then ls /home, you'll find that /home/readme still exists because rm only deleted a resource manager's readme, while another resource manager's readme still exists.

Multiple resource managers register the same path name

You might wonder why two resource managers would register the same path name. Besides 'backup' as one reason, there could be another possibility for 'non-stop upgrade'.

For example, a server v1.0 provides car speed search services by registering /car/speed; after some time, this server has a new v2.0 version. Generally, the upgrade step would be to shut down the previous v1.0 server and then start up the v2.0 server. But this means that even for just a brief moment, the /car/speed service disappears.

Better approach is to start v2.0 server first and let it also register /car/speed. However, since v2.0 server starts later than v1.0, all accesses to /car/speed will still go to v1.0 server initially. At this point, the v2.0 server needs to use the _RESMGR_FLAG_BEFORE flag when registering. This tells the path name manager 'although I registered late, but when someone queries /car/speed, please put me first'.

This way, after v2.0 starts up, all clients that open('/car/speed') will communicate with v2.0 server first. And
v1.0 will temporarily not exit the stage, it still needs to serve existing clients that have established connections with it until these clients end their connections (once v2.0 starts up, v1.0 will no longer have new connections because new clients will connect to v2.0).

When all current clients of v1.0 end their connections, it can officially exit. This way we perform a seamless 'no-interruption upgrade' from v1.0 to v2.0.

Someone with insight may have noticed that using _RESMGR_FLAG_BEFORE can make the server registered later than others appear earlier, thus allowing 'hijacking' of certain resources. Yes, it is indeed possible to do so, and this is the basis for QNX's devc-ditto service.

Pathname Manager Naming Transmission

QNX, like other Unix systems, has a standard ln program that is typically used to establish link files, whether they are hard links or symbolic links. This has specific meanings in the file system, which I think everyone knows. However, QNX has a powerful extension -

-P Create link in process manager prefix tree.

Usually we use it with -s, Unix file linking typically has certain limitations, such as not being able to cross different partitions, but QNX's -P is directly redirecting in the pathname manager because all pathname searches start by asking the pathname manager. Essentially, -P can link any 'pathname'.

 # ln -sP /dev/shmem /tmp

This method of using shared memory to do temporary servers is very common on QNX systems without external storage. Its actual meaning is that it says
Establish a redirect in the pathname manager, any access to files under .../tmp will be automatically redirected to
/dev/shmem/..., then start pathname resolution.

 # ln -sP /net/mediacenter/dev/snd /dev/snd

If your machine doesn't have a sound card, but another mediacenter does, this link will establish it directly on the local machine.
Directory of /dev/snd, local player programs do not need any modification when they try to open ("/dev/snd/...") at that time, the path
name manager automatically replaces it with /net/mediacenter/dev/snd/..., thus establishing a connection and message transmission between the local player and the sound card resource manager on Mediacenter. This is QNX's powerful 'transparent

distributed processing' (QNET), which allows for seamless collaboration between QNX systems without requiring any additional code.
(QNET), QNX to QNX, can easily collaborate across devices. Without having to write any code for this collaboration.

Let's have some fun again

```
# /bin/hostname
localhost
# ln -s /bin/ls /bin/hostname
# /bin/hostname
.               .lastlogin      .profile
..              .ph
# rm /bin/hostname #
/bin/hostname
localhost
```

What happened here? As you can see, initially /bin/hostname is working normally; but by forcibly linking /bin/ls to /bin/hostname, or saying that /bin/hostname path name points to /bin/ls after that, executing
/bin/hostname behaves just like executing /bin/ls. And rm /bin/hostname then deletes this redirection,
/bin/hostname then executes correctly again...

Conclusion

In summary, this explanation has covered the concept of QNX path name search and some details. Perhaps you will ask, what's the use of knowing these things?

Once you know how pathnames are searched, you can avoid designing some aspects that might decrease system performance during system design. For example, try not to overlap registered pathnames. If one resource manager registers "/media/db" and another registers "/media/control", it will make pathname searching simpler; whereas if one resource manager registers "/media" and another registers "/media/player", in the future every time you perform read-write operations on "/media/player", there may be unnecessary message transmissions.

Another common one is that because pathname searching needs to query different resource managers, the entire process will take relatively longer. Therefore, try to reduce the number of `open()` calls as much as possible, which can also improve efficiency. If you need to frequently communicate with a certain resource manager, it's definitely not a good idea to `open()`/`write()`/`close()` each time. It's better to `open()` once and remember the file descriptor (fd), then send commands using the fd afterwards.