Name Spaces and Plan 9 Srv

Erik Quanstrom, Coraid
quanstro@labs.coraid.com

ABSTRACT

Plan 9 differs from most other OSes in that the name space is not shared among all processes; different processes have different sets of resources (files). This is well described in [1]. However if difference processes had completely disjoint sets of resources, the system would be unusable. There are three basic mechanisms that manage the differences: `/lib/namespace`, process groups, and *srv*(4).

What is a Name Space?

In typical OSes the concept of name space is not often important since all processes see the same set of files. In Unix-like systems, the mount table encodes the single name space. Each entry in the mount table is a translation from a file in the existing table to a new file system. New mounts are inserted before existing mounts.

Plan 9 name spaces work along similar principles. However, there are some notable differences. Name spaces translate names to file servers, not file systems. Name spaces are a property of a process group, not the system. If process *a* in process group *a* mounts a remote file server, process *b* in process group *b* will not see this mount.

Plan 9 also provides two additional features: bind and union mounts. The “bind” operation allows a name space to translate between two names in the existing name space. Union mounts allow a name to translate to more than one file server or bind. The canonical example for this is `/bin` which is a union mount consisting of architecture specific programs, shell scripts, and user-specific programs. This allows `/bin` to have a full set of architecture-specific executables, even in a heterogeneous environment. Execution paths are seldom used in Plan 9.

How do we Build Name Spaces?

Name spaces are built using a series of *mount*, *bind*, and *unmount*(2) system calls. *Mount* adds a name space entry that maps a name to the named channel; *bind* maps a name to another name. Channels are the kernel’s abstraction behind file descriptors, taking the same role as Unix inodes. Typically process of building a name space is automated by *newns*(2). This function uses `/lib/namespace` to construct a standard name space. Environment variables and includes are used to handle cases depending on the cpu type or system. Kernel devices exist independently of name spaces. They are referred to by *#c* where *c* is the kernel device’s character. This allows a process to create a new name space *ex nihilo*.

The relationship between name spaces is controlled by *rfork*(2). The result may be either a shared, a copied or an empty name space. It is not currently possible for unrelated processes to share a name space.

Srv

Fortunately, there is a way for unrelated processes to share a channel. The kernel file server *srv*(3) allows programs to post a channel (the kernel's idea of a file descriptor) and give it a name. Since any process can bind #s into its name space, any process can post a file descriptor or read from an already-posted file descriptor. If the file descriptor posted can speak 9p messages, it may be mounted. So, for example

```
; ramfs -S example
; cpu
cpu; mount -c /mnt/term/srv/example /n/example
cpu; touch /n/example/x
cpu; <eot>
; mount -c /mnt/term/srv/example /n/example
; lc /n/example
x
;
```

Most file servers only need a few lines of code to post themselves as a service

```
ARGBEGIN{
case 's':
     srv = EARGF(usage());
     break;
if(srv != nil){
     snprint(buf, sizeof buf, "/srv/%s", srv);
     fd = create(buf, OWRITE|ORCLOSE, 0666);
     if(fd == -1)
          sysfatal("srv: %r");
     snprint(buf, sizeof buf, "%d", fd);
     if(write(fd, buf, strlen(buf)) != strlen(buf))
          sysfatal("srv: %r");
}else if(mount(fd, -1, defmnt, MREPL|MCREATE, "") == -1)
     sysfatal("mount: %r");
```

When using *9p*(2), *postmountsrv* or *threadpostmountsrv* will post the file descriptor if given the *name* argument.

Srvfs

Sometimes, one wishes to share already-built name spaces or file servers that don't have the innate ability to post their channel to srv. In that case, *Srvfs*(4) can be used. It works by posting a mountable file descriptor to srv which it uses to serve parts of its own name space.

```
w0; ramfs
w1; srvfs example /tmp
cpu; mount -c /mnt/term/srv/example /n/example
cpu; touch /n/example/x
cpu; <eot>
; mount -c /mnt/term/srv/example /n/example
; lc /n/example
x
;
```

Exportfs and Cpu; Putting it all Together

If *srvfs* can export a file server via srv, it is not hard to imagine a variant, *exportfs*(4), which exports a file system over the network. *Exportfs* is implicitly invoked by *import*(4) on a remote machine. For example

```
; import /net.alt/tcp!minooka.coraid.com!exportfs / /n/coraid
; import aoe '#æ' /n/aoeæ
```

would import / from `minooka.coraid.com` and the kernel device #æ from `aoe`. In both cases, *exportfs* builds a name space using `/lib/namespace` to export. In the first case we import / from the name space. In the second, we ignore the name space and import a kernel device.

One can also restrict name space *export* uses by supplying a customized one with `−N` and disallow kernel devices by filtering out certain names with `−P`. For example, one can restrict an export to a home directory with the following export

```
minooka; cat > nsfile
mount −aC #s/boot /root $rootspec
bind −a $rootdir /
bind −c /usr/$user/ /
aux/listen1 −tv /net.alt/tcp!*!999 /386/bin/exportfs \
     −P <{echo '− ^#'} \
     −N /usr/quanstro/tmp/nsfile −a −A '$netdir'
```

This export can be incorporated in the local name space with

```
import tcp!minooka.coraid.com!999 / /n/q
```

Finally, it should be clear how *cpu* works. Locally, *exportfs* is used to export the local namespace. On the remote, it is imported and mounted on `/mnt/term`. The local `/dev/cons`, etc. are bound onto the remote's `/dev.`

References

[1] Plan 9 from Bell Labs, *Computing Systems*, Vol 8, #3, Summer, 1995, pp. 221—254.