A File Server Directory Tree Hierarchy

WRITTEN BY

***Thomas Lundgren* - TN/ETX/TX/DDU - ERICSSON TELECOM**

ABSTRACT

Today there exists Sun workstations of different machine architectures and they are not all binary compatible. This article explains a strategy which can be used by the Local System Administrator (LSA) when creating the directory tree on a file server.

The strategy has been developed for a heterogeneous file server supporting multiple architectures, multiple OS versions, multiple window systems and multiple revisions of each software product.

A Bourne Shell script has been written to automate the creation of the directory hierarchy

INTRODUCTION

This article is based on Ericsson Telecom document `ETX/TX/DD-91:110 Uen Rev.B`.

Most of the solutions presented here are not new and has been in use for some time. But many of them are not well documented and they only exists as conventions, only known to experienced system administrators. You learn the different conventions as you get experience, therefore beginning system administrators seldom have any knowledge about them.

This article therefore tries to supply inexperienced system administrators with techniques and solutions to begin with. These solutions are interesting for experienced system administrators too if they don't have similar or better solutions.

BACKGROUND

As a beginning system administrator I found that the system manuals doesn't say much about what improvements to the system you could make when you got the system up and running. I mean, when all you have is an *out of the box system*. All our system had that could be called additions to an out of the box system, was a small file tree below `/usr/local`.

And at the time, spring 1989, all that had been seen at our site so far, was a homogeneous environment (all Sun3). But at that time the SPARC systems and also the new graphical window interface Open Windows appeared. So I had to decide where to place the new files without causing any "collisions" with the existing software.

The department I work at is developing software tools for workstations and need to have several versions of the products that we are developing installed simultaneously. We developed a technique for this and the same technique was also used with some of the more important third party software packages used daily.

This technique came to use when we installed a new release of a documentation tool used at our department. The new release wasn't compatible with the previous release. And therefore it was necessary to supply both versions at the same time. We then had a period when we converted all the different documents. And finally we could remove the old version.

We also compiled and maintained a lot of *free-ware*. I found that most free-ware didn't support heterogeneous environments, and was more suited to be installed in an homogeneous environment.

Also, it had to be a system that was forward compatible as much as possible, so that it didn't need to be changed in the future. And for the free-ware that only managed homogeneous environments, it needed to be backwards compatible. So from an application point of view (and user's) there should be as few differences from a homogeneous environment as possible.

GOALS

The following goals was set up:

- Use of existing conventions as far as possible.
- Support of multiple machine architectures.
- Support of multiple operating system releases.
- Support of multiple graphical window systems.
- Being able to have multiple releases of a single software product installed simultaneously.
- Backwards compatible. The new environment should be opaque at the application level. “Old” software designed to be executed in an homogeneous environment should still be able to work in the new environment.
- Forwards compatible. It should be easy to add future architectures, operating system releases, and window systems without needing to change the strategy itself.
- There should not exist any duplicate information in the system. Architecture independent files that can be shared between architectures should be shared to save disk space.
- It should not become difficult or complex to install software.
- It should not be difficult or cause much work to introduce the new strategy.
- The environment should be identical on all hosts, NFS clients and NFS servers.

Supporting multiple architectures

First the different types of files must be defined. Some files are architecture independent and some files are architecture dependent. Files can be architecture dependent at different levels, either at the *application level* or the *kernel level*.

Machines which run different kernels due to underlying hardware differences, may yet be able to run the same application programs. For example the Sun 3/60 (MC68020 based) and the Sun 3/80 (MC68030 based) run different SunOS kernels, but can execute the same applications. One describe these machines as having the same *application architecture* but different *kernel architectures*.

The application architecture defines the environment an application sees, and the kernel architecture defines the environment the SunOS kernel sees. Kernel architectures are transparent to users and developers of user level applications. All binary executables are at least application architecture dependent and programs reading kernel data structures, such as `ps`, are kernel architecture dependent.

This results in three types of files, architecture independent files, application architecture dependent files and kernel architecture dependent files.

The different types of files will be separated into three different types of *minor file sets* as follows:

- Architecture independent files will be placed in a file set shared between all architectures, the *shared file set*.
- Application architecture dependent files will be placed in the *application file set*.
- Kernel architecture dependent files will be placed in the *kernel file set*.

For each of these minor file sets there exists a directory tree structure with a top level directory. All files belonging to a specific file set will be found below the top level directory of the directory tree representing the minor file set. This way you can easily select the file set by using the `mount`(8) command and mount the top level directory. With that you have grabbed all files in one single step. As described there are three types of file sets, so there is three mounts that have to be made.

For each type of minor file set there may exist one or more *variants*, one for each supported application architecture or kernel architecture.

From the system management point of view, it’s most convenient if the top level directories of these file sets have names based on the architecture type they contain. You can get the application architecture and the kernel architecture by giving the `arch(1)` command. Then the output from the `arch(1)` command can be used to build different paths.

Only one shared file set is needed and the top level directory will be named share. Three application architectures exist and the name of the application file set variant’s top level directories are `sun386`, `sun3` and `sun4`. And finally there are six possible kernel architectures and the name of the kernel file set variant’s top level directories are, `sun386`, `sun3`, `sun3x`, `sun4`, `sun4c`, and `sun4m`.

It’s very easy to support future architectures. The only thing that has to be done is to create new file set variants.

The different minor file sets will contain certain major directories specified below. When a minor file set is created, it is originally created with these subdirectories in the minor FS top level directory. They should suffice for the future and one should think twice before adding any directories at this level. That would almost be the same as adding new directories to the `/` directory.

When a minor file set is created these original directories will be empty indicating that the minor file set is empty. Anything that’s added to the minor file set is

placed within these major directories. In this way the topology of the top level directory is always the same, just like the topology of the `/` and `/usr` directories.

Application file set

This file set contains files that only can be shared within an *architecture family*. For example the sun3 family with the kernel architecture pair sun3 and sun3x. This file set can also contain files that themselves are not architecture dependent but rather the information they contain. For example some information or programs may not exist for both the sun3 architecture and the sun4 architecture. The following are the directories in the top level directory of the application file set:

`adm` By convention this directory contains system administrative information such as documents describing how the system has been configured etc.

`bin` Architecture dependent executables. This is the main directory where user commands are located.

`data` This directory maybe not exist by convention but I like database files to be by themselves. This directory contains database files that are architecture dependent (perhaps because differences in memory alignment between architectures).

`demo` Demonstration programs.

`etc` Programs for system administration and their auxiliary files. Having a special directory for these files you can set the directory permission so that no other than authorized users can access the contents of the directory.

`games` Games. If the games need a `lib` directory for auxiliary files, place the lib directory within the game major directory. Whatever opinions exists on games, this is just a directory that by convention exists.

`kvm` Mount point for the *kernel file set*.

`lib` Contains different object file libraries and runtime libraries and other architecture dependent data files.

`share` Mount point for the *shared file set*.

Kernel file set

Normally the `kvm` directory doesn't contain any major directories for binary executables and object file libraries. But this strategy will consistently place at least the following two directories in the top level directory of each minor file set.

`bin` Kernel architecture dependent binary executables.

`lib` Kernel architecture dependent object file libraries.

Most sites does not need the kernel file set.

Shared file set

The shared file set contains all files that can be shared across architectures. A list of directories found in the top level directory of the shared file set follows:

`adm` Architecture independent system information, logfiles, databases, etc.

`bin` The `bin` directory in the shared file set contains architecture independent executables. That is, programs that are read by interpreters, such as shell scripts, BASIC programs, LISP programs etc, are placed here. They must not contain any commands or information causing them to become architecture dependent.

`data` Architecture independent database files.

`dict` Word lists etc, maintained locally.

`doc` Documents.

`etc` Architecture independent executables performing system administrative tasks (shell scripts). They must not contain any commands or information making them architecture dependent.

`include` Header files.

`lib` Contains subdirectories with program utilities, fonts, raster files, icons, "sound files" etc. It also contains `Cshrc`, `Profile`, `Login` files etc which the user can read in their `.cshrc` and `.login` files by for example using the `csh` internal `source` command to get the environment set up. This directory does not contain any binary object file libraries.

`man` Manual pages.

`src` Contains program source code. This strategy assumes that source code is architecture independent, meaning that only one copy of the source code is necessary. This requires that differences between architectures must be considered when writing source files, to make them compilable on all of the target architectures.

Location of minor file sets

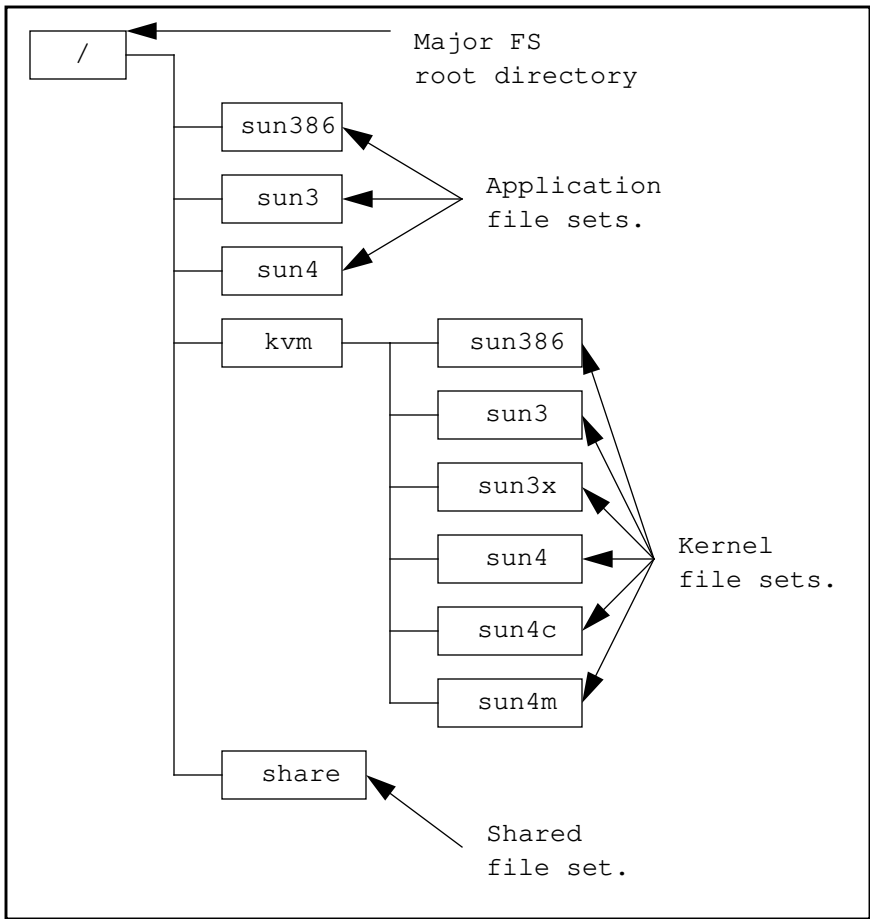
All these minor file sets are placed together as one single directory tree called a *major file set*. An example of a major file set is the file tree below the `/export/exec` directory on a file server. The top level directories of the minor file sets are placed in the top level directory of the major file set. The name of the major file set's top level directory depends on the purpose for which it exists. The name can be `local` (`/usr/local`), `project` etc.

But since some of the kernel architectures have the same name as some of the application architectures, their top level directories can not coexist in the same directory. They have to be placed in a different directory. The existing convention used in the `/export/exec` directory is used, where the kernel architecture dependent files reside in a subdirectory named `kvm`.

Where the minor file sets are located in a major file set is presented in Picture 1. The sun386 architecture has been omitted from the rest of this article.

Most sites do not need the kernel file sets. Normally such files only exists in the `/usr` file system. But if you have installed software that reads kernel data structures such as `top`, then you need to implement the kernel file sets.

Picture 1 Minor file set's location within a Major file set



Supporting multiple SunOS versions

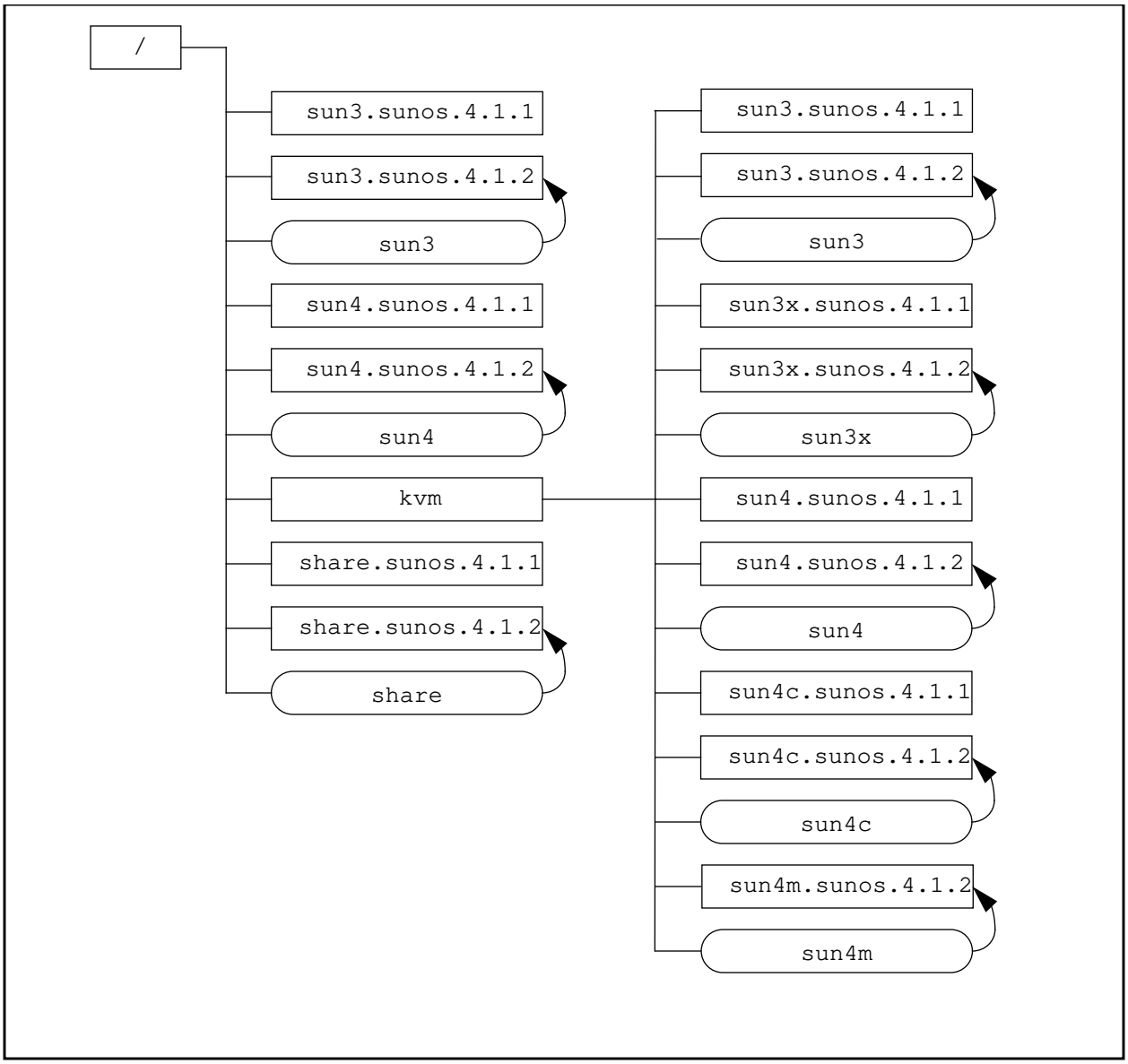
The different SunOS versions will be separated with the same method introduced with SunOS 4.1. There will be one file set of each type and variant listed above for each supported SunOS version.

According to the naming conventions in SunOS 4.1 the top level directory of a minor file set will be composed of the *architecture name*, followed by the word *sunos* and the *SunOS release number*, the three separated by dots. For example the top level directory of a kernel file set for the sun4c kernel architecture and SunOS release number 4.1 will be named `sun4c.sunos.4.1`.

The shared file set doesn't belong to any particular architecture so in the position for the architecture name the word *share* is placed. With the example above we get `share.sunos.4.1`. When a file set is mounted, the type of file set, the variant (architecture), and the SunOS version is selected. Adding this to the previous schema the expanded directory structure would then look like the one in Picture 2, if for example SunOS 4.1.1 and SunOS 4.1.2 is supported (sun386 architecture omitted).

The symbolic links are used to select a default version. The System Administrator creates these symbolic links to point to the current default OS version which is SunOS 4.1.2 in this example. Normally only the type and variant of file set is specified when mounting, and the SunOS

Picture 2 Support of multiple OS versions.



version number is omitted. This causes the OS to follow the symbolic link and the default version is used. If a file set for a specific SunOS version is needed the version number is added.

Location of major file sets

I mentioned earlier that major file sets can exist for any purpose. One is the `/usr/local` file set that carry software produced locally or supplied by third party vendors. Another could be the `/usr/project` file set that is used for different kinds of projects.

According to Sun's convention file systems exported to clients is mounted below the `/export` directory on the NFS server. Following what's been said earlier, the major file sets `/usr/local` and `/usr/project` will be mounted on `/export/local` and `/export/project` on the NFS server and mounted on `/usr/local` and `/usr/project` by the NFS clients. File systems like `/home` should also be mounted below the `/export` directory, ex. `/export/home/servername`.

Mounting of file sets

The correct minor file sets in a major file set is selected in the mount phase during system boot. I use the major file set `/export/local` as an example. A sun4c NFS client makes the mounts showed in Picture 3.

These mounts will use the default SunOS version since they point to the symbolic link. If the same NFS client requires SunOS 4.1.1 it will use the mounts in Picture 4.

A Sun3 client changes the architecture name on the server side but use the same paths for the client side to create an identical file tree.

This way a Sun4 client can only access Sun4 architecture dependent files and a Sun3 client can only access Sun3 architecture dependent files. Both sun3 and sun4 clients can access the architecture independent files in the shared file set.

Special additions for file servers

Search paths are now identical on all NFS clients but not on the NFS servers. As an example I take `/usr/local`. On the server the files are accessible trough `/export/local` not through `/usr/local`. And the installed software have been configured to be executed through `/usr/local`.

If you look at a heterogeneous file server's `/usr` file system you see that Sun Microsystems has solved that by using two file systems. The application and kernel file set for the file server is installed in the `/usr` file system. And for other architectures they are installed in the `/export/exec` file system.

I think it's better to have everything in one file system. It's easier when you upgrade the OS. In the case of the `/usr` file system above you have to copy the old files in the `/usr` file system to the `/export/exec` file system if you want to support different OS versions. And then install the new OS version in the `/usr` file system and for the other architectures in the `/export/exec` file system.

There is a solution for having all minor file sets in a major file set in the `/export` directory and still be able to execute the programs through `/usr/local` on the file server. The solution is explained in the following section.

Loop back mounts

To be able to execute software placed in `/export/local` and configured to be executed through `/usr/local`, loop-back mounts of `/export/local` to `/usr/local` are made. A sample `fstab` file from the NFS server is found in Picture 5 on page 7.

Normally the `/etc/rc*` files only perform UFS (UNIX File System) and NFS (Network File System) mounts during boot. Modifications of the

Picture 3 Mounting file sets using default SunOS version

```
server:/export/local/sun4                    /usr/local
server:/export/local/kvm/sun4c               /usr/local/kvm
server:/export/local/share                   /usr/local/share
```

Picture 4 Mounting file sets from a specific SunOS version

```
server:/export/local/sun4.sunos.4.1.1        /usr/local
server:/export/local/kvm/sun4c.sunos.4.1.1   /usr/local/kvm
server:/export/local/share.sunos.4.1.1       /usr/local/share
```

`/etc/rc.local` file is necessary and part of a NFS server `/etc/rc.local` file is showed in Picture 6 on page 7.

When this is done, all files of the same type are accessible with the same paths on all of the server's NFS clients and on the server it self.

There is one problem that we have noticed. Any program using the `getwd()` (get working directory) system call, such as `csh` when it initializes, in a loop back file system mounted on top of another loop back file system, will fail and exit with the error message "getwd() failed". An example of this is the shared file set and the kernel file set which are loop back mounted on top of the loop back mounted application file set. The work around is to descend through the `/export` directory to the corresponding point if you want to edit any files. If you just want to start a program that must make the `getwd()` system call you just start it from a directory outside the loop back file system.

This isn't any problem because the only reason the loop back mounts are made is so that executables can be accessible with the correct paths and so that executables can execute auxiliary files and load library modules. If you are going to edit files or do other activities you need to remember that you should access the files through the `/export` directory.

You may wonder why LOFS is used and not NFS. This is because if you bring the file server down to single user mode, all network services are terminated and the server will hang, complaining about that it can't get any answer from itself. LOFS on the other hand works well in single user mode.

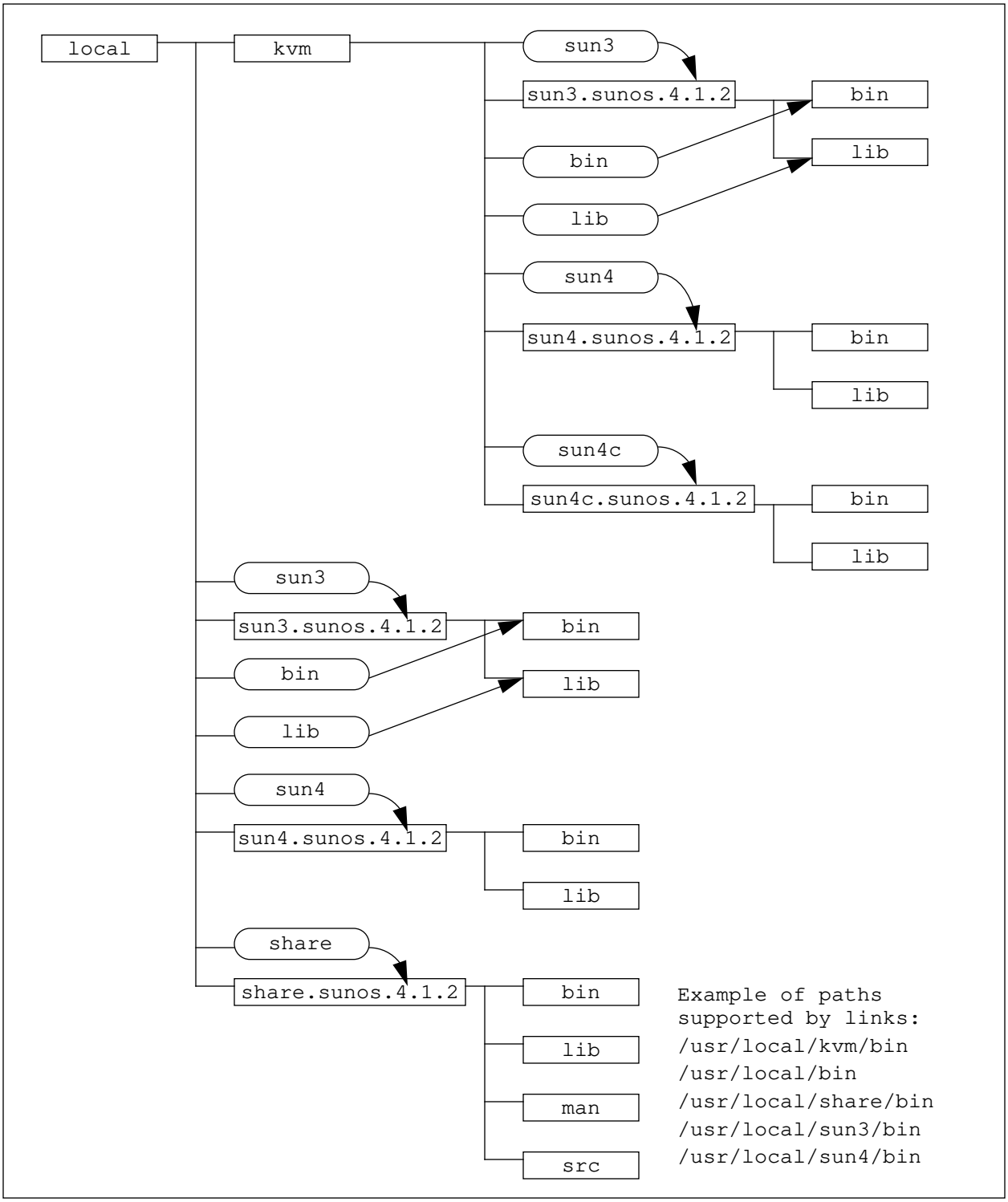
Picture 5 Sample file server `/etc/fstab` *file.*

```
/dev/id000a                  /                        4.2   rw    1 1
/dev/id001a                  /rootb                   4.2   rw    1 1
/dev/id000g                  /usr                     4.2   rw    1 2
/dev/id001g                  /export/exec             4.2   rw    1 2
/dev/id000e                  /export/root/0           4.2   rw    1 3
/dev/id001e                  /export/root/1           4.2   rw    1 3
/dev/id000d                  /export/swap/0           4.2   rw    1 3
/dev/id001d                  /export/swap/1           4.2   rw    1 3
/dev/id002c                  /export/local            4.2   rw    1 4
/dev/id003c                  /export/project/garbo-A  4.2   rw    1 4
/dev/id000h                  /export/home/garbo-A     4.2   rw    1 4
/dev/id001h                  /export/home/garbo-B     4.2   rw    1 4
/dev/id000f                  /var                     4.2   rw    1 5
/dev/id001f                  /tmp                     4.2   rw    1 5
/dev/id001b                  swap                     swap  rw    0 0
/export/local/sun4           /usr/local               lo    rw    0 0
/export/local/kvm/sun4       /usr/local/kvm           lo    rw    0 0
/export/local/share          /usr/local/share         lo    rw    0 0
```

Picture 6 Part of file server's `/etc/rc.local` *file.*

```
intr -a mount -vat lo
intr -a mount -vat nfs
echo ' '
echo 'Mounting TFS, this may take a very long time.'
echo ' '
intr -a mount -vat tfs
intr -a mount -vat tmp
intr -a mount -vat pcfs
```

Picture 7 Part of major file set installed below `/usr/local`*. To make it appear as if an application file set was mounted directly on /usr/local, symbolic links are made to the correct application file set variant's major directories.*



Soap...

Because of this you can work with your powerful favourite editor (`emacs`) and tools in single user mode and not end up using the `vi` editor (:-}).

Also, you might think that you could replace the `/usr/local` directory with a symbolic link to the `/export/local` directory. But then you forget that the NFS clients uses the server's `/usr` file system and if `/usr/local` isn't a directory they won't be able to NFS mount the application file set on `/usr/local`.

...Soap

Symbolic links

This is the second alternative if you don't want to have the file system below the `/export` directory and don't want to use LOFS. It is possible to place a major file set at the same point in the directory tree on a file server as the point the clients use. As an example I use the major file set `/usr/local`. Normally the clients would mount this file system from `server:/export/local/`*arch* on `/usr/local`. But imagine that you are configuring a very "small" system and that there are not enough disk partitions to go about. The system administrator could then place the file set in the `/usr` file system. The contents of `/usr` and `/usr/local` will then be in the same file system.

For the NFS clients this does not impose any problems. A sun4 client as an example could mount `server:/usr/local/sun4` instead of `server:/export/local/sun4`. But to be consistent they should still mount the major file set from `server:/export/local/sun4`. All that's needed on the server for this to work is the symbolic link `/export/local->/usr/local`. Below `/usr/local` you have the top level directories of the application file sets. If a sun4 client make a reference to `/export/local/sun4` it will mount the `/usr/local/sun4` directory. This works basically the same as the symbolic link `/export/exec/sun4 ->/usr`, if the file server is a sun4. And as you know this link exists even if the server is a homogenous server.

From the NFS client's point of view all NFS servers are heterogeneous.

Now, think about where the directories with the executables ends up on the file server. The path is either `/usr/local/sun4/bin` or `/usr/local/sun3/bin`. One of the goals is to make the environment identical on all hosts inclusive file severs. This is possible by using numerous symbolic links.

The principle is simple. In the top level directory of the major file set, symbolic links have to be made to the correct minor file set's major directories. These links are shown in Picture 7 on page 8.

As an example I take the `bin` directory. On a heterogeneous sun3 file server there would be two `bin` directories, `/usr/local/sun3/bin` and `/usr/local/sun4/bin`. What needs to be created is the path `/usr/local/bin`. Therefore the symbolic link `bin->sun3/bin` is created. The `sun3` component of the path is, as described earlier, a symbolic link pointing to the default SunOS version. So if you reference `/usr/local/bin`, if the default SunOS version is 4.1.2, the full completed path is `/usr/local/sun3.sunos.4.1.2/bin`.

The same thing is done with all major directories finally making files accessible with the same paths on the file server as well. This is the solution we used at our site before using loop back file systems. It's not a beautiful solution but it works.

File tree on a "small" homogenous server

If you have a server with very little disk space like the one described in the section above, and if it is homogenous, then you can implement a much more simple scheme. You don't need to implement all those symbolic links.

On a homogenous file server you need not have the major file set's mounted below the `/export` directory so you don't need a special partition for that file system. You can have them on the same location as the clients use. For example you could add the major file set `local` below the `/usr` file system's `/usr/local` directory. If the *installation path* and the *execution path* is the same it might simplify things for different software installation programs. Many such programs don't support the fact that the *installation path* could differ from the *execution path* used at runtime.

What's required is that you still support the *export-paths* since the NFS clients view the NFS servers as heterogeneous. Therefore you should create the links showed in Picture 8, if the homogenous server is a sun4.

Below `/usr/local` you skip the support of multiple architectures and SunOS versions. If the homogenous server is a sun4 the directory `/usr/local` *is the top level directory of the sun4 variant of the application file set*. And the shared file set is integrated with the application file set. `/usr/local/share` is the top

Picture 8 Symbolic links to support the `/export` *mount paths.*

```
/export/local/sun4 -> /usr/local
/export/local/kvm/sun4 -> /usr/local/kvm
/export/local/share -> /usr/local/share
```

level directory of the shared file set and not a mount point for the same. So the directory tree below `/usr/local` has the same structure as the directory tree that the NFS clients would assemble from a heterogeneous file server.

If you need to support the other kernel architecture, sun4c, the server isn't homogenous. But, the kernel file set's are often small enough to fit into the root file system. So the kernel file set for the sun4c kernel architecture could be stored in the original location `/export/local/kvm/sun4c`.

You can't support multiple SunOS versions if your server is to be a homogenous server since if you do, the server isn't homogenous. And you will need a complete new set of minor file sets for the second SunOS version. And to store these you will need a new file system. So as the title of this section says, this scheme is just a special variant for a very "small homogenous file server" or standalone machine.

Supporting multiple window systems

Support of multiple window systems is basically done the same way as for multiple architectures, where the different types resides in separate file trees. Applications are at least application architecture dependent and therefore they are first sorted into the correct application file set. *Secondly they are grouped after which window system they require*.

So in the top level directory of the application file set and the shared file set, directories are created and named after each window system supported by the local system. For example, if you support the three window systems Sunview, Open Windows, and X11, you create the three directories `sunview`, `openwin`, and `X11`.

The window system's subdirectories will hold *micro application file sets* or *micro shared file sets* depending on in which type of minor file set they appear. Each window system will have it's own file set in both the application file set and the shared file set. To put a label on them they can be called *window application file set* and *window shared file set*.

In the *window application file set* the two directories `bin` and `lib` are created. The *window shared file set* contain the directories `bin`, `lib`, `include`, and `man`.

Open Windows is a X11 compatible NeWS server (Well, Open Windows 1.0 and 3.0 is a X11 compatible NeWS server, but Open Windows 2.0 was a NeWS compatible X11 server). One expect to be able to run X11 clients compiled on other X11 systems in Open Windows without any problems, but that isn't the case.

This because Open Windows lacks many of the popular toolkits and the different run time libraries that belong to these toolkits. This is the main reason why we bothered with installing the X11 version distributed by MIT. It was only to compile the different toolkits and libraries, to be able to compile the different clients, many distributed with the X11 source and many distributed by net news. Also, the revision of the X11 archives never seems to be the same in the latest release of the two versions. Clients compiled under the MIT version of X11 don't always run smoothly under Open Windows. So at our site we ended up separating the two versions of the X11 window system.

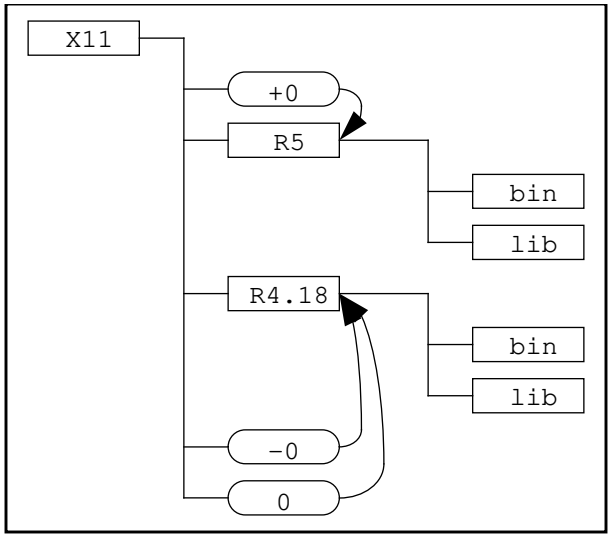
You may wonder if there exist any architecture independent executables that can use any window system. Well, I thought about this and found that I had written one myself. I had written a shell script which in turn started an X11 menu program which presented some alternatives and then returned the selection to the script. Therefore the script became not architecture dependent, but window system dependent. So watch out. You may not notice the small differences.

Supporting multiple releases of the same window system

If your department is developing software which is window system dependent, then your CASE environment must be able to support multiple releases of the window system for which you are writing the software. The programmers must be able to use Beta releases if they need to when they are programming.

To do this you do a minor correction to the file tree presented in the next section. The technique is basically the same as the one explained in the section *supporting multiple software versions*.

Picture 9 Window application file set supporting multiple versions of a window system.



What you have to do is to add a *version level* to the *window file set*. As an example you could have the release `R4.18` of X11 and you just acquired release 5. And so you might like to have it tested first. So if you add a version level to the X11 window file set it looks like Picture 9.

When using the default version, the path follows the symbolic link `0`. If the *prerelease* is going to be used the path should follow the symbolic link `+0`. For further information on managing multiple software versions se section *supporting multiple software versions*.

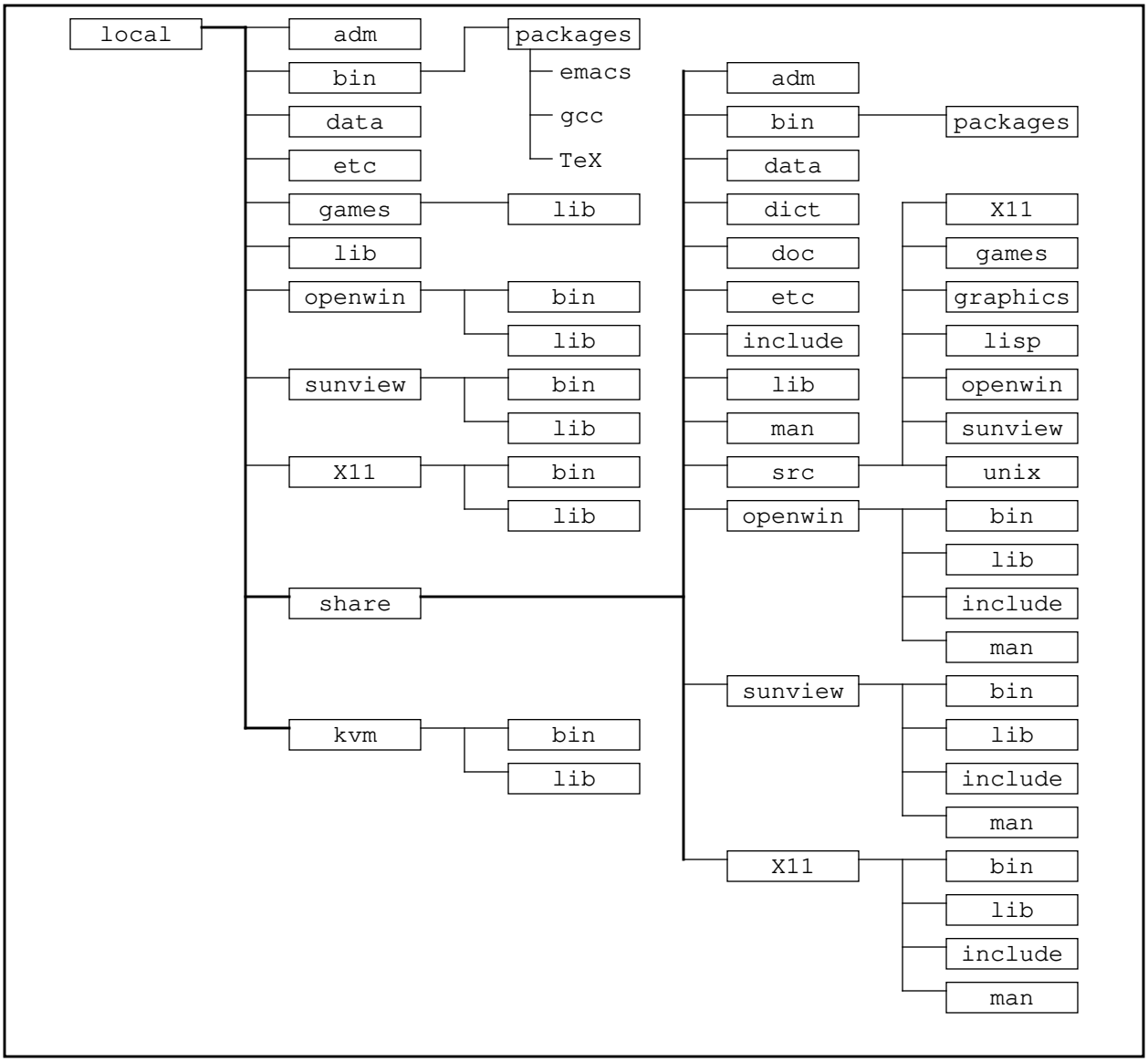
The NFS client's assembled major file set

From the user's and the application's perspective the directory tree on a Sun4 client is identical to that of a Sun3 client. How this looks on a client with the same previous example, if you don't support multiple window systems as explained above, is presented in Picture 10.

Additions for backwards compatibility

If one want a directory in the shared file set to be accessible in the application file set a symbolic link has to be created.

Picture 10 NFS client's assembled Major file set directory tree.



Two cases are the `src` and `man` directories which exists on most systems, but with this approach they have been placed in the shared file set which is mounted on the share directory. For such directories just create symbolic links such as `man->share/man` and `src->share/src` in the top level directory of each application file set.

Supporting multiple software versions

Supporting multiple versions of each application simultaneously is basically done the same way as with machine architectures and operating system releases. The different program versions are placed in their own sub directories. Separating them this way makes it possible to support multiple versions.

What you have to do is to have one directory tree for each installed software version. To avoid that the path variable becomes too long symbolic links are made from the minor FS `bin` directory to the `bin` directory in the specific software's installation directory tree. This is done for small software packages or software packages with only a few executables or even only one start-up script such as Frame Maker's `imaker` executable. Packages with many executables is explained below.

In the application file set's `bin` directory you create a special directory named `packages` to hide the different software installation directory trees in. In the `packages` directory you create directories named after the software they contain.

In the directory, named after the software, you create the installation root directories named after the software version number. So if you have *Program* in versions `4.2` and `4.3` you will have the following paths from the application file set's root directory; `./bin/packages/`*Program*`/4.2` and `./bin/packages/`*Program*`/4.3`.

Now it's tiresome for users having to remember all the different version numbers of different applications that exists in the system. Especially if they write their own `~/.cshrc` and `~/.login` files, and for you as a system administrator if you write similar files or start-up scripts. And when a new version of a software is installed the scripts have to be updated for the users to be able to access the new version.

Therefore you should add symbolic links in the version level of the directory tree to the current default version. We use version number `0` as the current default version. So if version `4.3` in this example is the default version, the symbolic link `./bin/packages/`*Program*`/0->4.3` is created.

When you install a new version, say `4.4`, you may not want to make it publicly available immediately, you may need to test it first. For *prereleases* we have the symbolic link `+0`. And if some one need to run the version prior to the current version we have the version number `-0`.

When you have tested the new release you just remove the `0` link and create a new that point to the new release. `+0` is left as it is since there isn't any prerelease available, and `-0` is updated to point to the old version. This way users only have to know the versions `0`, `+0` and `-0`. And they are common to all installed software.

When this is done you create the links in the minor FS `bin` directory to the software's `bin` directory. And you reference the software's `bin` directory through the symbolic link `0`. This way you don't need to create these links or update them when you make an upgrade unless there are new programs that have been added to the product. Adding these links in the `bin` directory is simple. And it doesn't matter how many links that have to be made. In the minor file set's `bin` directory you run the command showed in Picture 11. As you probably noticed you can't just follow this strategy blindly. There exists applications with many executables (more than 50). And if you have a couple of applications with this number of executables the directory listing of the application file set's `bin` directory would become quite long. For these larger applications it would be better to just add the application's `bin` directory to the environment variable **`PATH`**.

The software's installation tree should also contain the `lib` and `man` directories. Having all files below one single directory makes it easier to remove the software if necessary. As you made symbolic links to the bin directory you also create symbolic links from `/usr/local/share/man/man?` to the `man` directory in the software's installation tree.

Picture 11 Commands to make symbolic links from the application file set's bin directory to the executables in the bin directory in the package installation directory tree.

```
foreach link (packages/Program/0/bin/*)
      ln -s $link
end
```

Picture 12 Updating the symbolic link to the default version.

```
rm 0 ; ln -s 4.4 0
```

Managing software this way the current version of the software can be accessible by users while the system administrator installs the new release. And when the system administrator decides to upgrade he or she runs the command showed in Picture 12 on page 12, in the directory at the version level if for example the new version is 4.4.

Mounting of other FS

For many of the other file system you probably don't need to support multiple architectures, SunOS versions, window systems etc. One example is the `/home` file system and also the project areas that you mount below `/usr/project`.

But I have included them in the strategy too. And some of the ideas can be used for them too. One is that you should continue to consequently place file systems exported to NFS clients below the `/export` directory on the NFS server. Therefore `/home/servername` should be placed as `/export/home/servername` and `/usr/project` should be placed as `/export/project`.

If there exists multiple major file systems or file systems without the special directory tree, place those that exists for the same purpose in a joint directory. For example if you have more than one *project area* you can place them as in picture 13 on the NFS server.

We experienced a certain confusion at our site regarding the project areas. When someone want to have a look at files that someone else has in a certain project area they have to ask to things, the name of the project area and the name of the project's top level directory. For casual users the name of the project area seems to be the hardest to remember from one time to the next.

So we supplied symbolic links in the `/usr/project` directory to the right directory in the correct project area. This way we can move projects to new project areas if necessary. All we have to do is supply a new link to the new location. All the users have to know is the name of the project directory.

The users should therefore always access their project areas through the symbolic link by using the path `/usr/project/`*projectname*.

Installing programs

Large program systems often consists of a start program or script, that the user executes. This program can also be the main program. Other functions that isn't executed frequently is compiled into different auxiliary executables executed by the main program, for example a program that format and sends the result to printer. And so the main program must have a way to find it's auxiliary programs. The main program may also need to find other files like initialization files or menu files etc.

Access using PATH variable

Program systems that only consists of executables may use the **`PATH`** variable to find the auxiliary files. This is directly supported by the `execvp(3V)` system call which mimics the shells use of the **`PATH`** variable.

Access using environment variables

Larger program systems normally uses environment variables to find the installation root directory and from there traverse the directory hierarchy to a well known place and locate any auxiliary file like executables, configuration files, menu files, help files etc.

Therefore you can install the programs anywhere and the only thing that has to be done is to update the environment variable for the program to work.

Access using hard coded paths

When you compile the programs your selves you are often required to decide and supply a path to a place where the software will be installed. And the path is then hard coded into the executable code and the programs then uses that path to find the auxiliary files. To change the path you have to recompile the programs.

This doesn't offer you much flexibility. NFS clients mounting file systems carrying programs that uses hard coded paths are not free to mount them where they like. They have to see to that it gets mounted in such a way that the required path is created for the program to work. If that isn't possible a symbolic link has to be created.

Compiling and installing software

Normally the compilation and installation is automated by supplied make files which normally have the targets `all`, `install`, and `clean`. Since the paths are identical for all architectures when using this strategy, the make file and or source files will normally need to be modified only once.

Picture 13 Location of home and project areas on the NFS server.

```
/export/project/servername-A
/export/project/servername-B
/export/home/servername-A
/export/home/servername-B
```

First log in to a host of the architecture for which you are compiling. Run `make all` and then `make install`, or just `make install` if it performs both steps.

Then run `make clean`, because there can't be object files around when the next architecture is being compiled. Therefore the source directory can't be the same as the installation directory (I have seen such cases). If non of this is automated by a make file you'll have to do this by hand.

Then login to a host of the next architecture. As described earlier, that host has another application file set and kernel file set mounted. Perform the same steps as above which will install the program system in that application or kernel file set.

To be able to compile and install files from a client, see to that the client has mounted the file system read write and that the server exports the file system read write. You also have to check that the user installing the software has permission to write files into the installation directory. You can create a special group for this, for example `swmaint` for software maintenance.

Normally the files in the `/usr/local` file system is owned by the group `staff`. But it would be better to create a special group or several groups for the purpose of software maintenance. This way you have separated the people who do software maintenance from those belonging to the group `staff`. People belonging to the group `staff` also has permission to update some files in directories such as `/etc`, `/usr/etc`, `/bin`, `/lib` etc.

APPENDIX

Here follows the `mkmfs.sh` Bourne shell script which you run on the file server to build the directory tree in for example `/export/local`. Before you run the script you have to edit the parameters in the beginning so the script is properly configured for your system. An example of how to use the script is showed in Picture 14.

Picture 14 Example of how to invoke the `mkmfs.sh` *Bourne shell script.*

```
mkdir /export/local
mount /dev/id001g /export/local
cd /export/local
mkmfs.sh `pwd`
```

Picture 15 Listing of the `mkmfs.sh` *Bourne shell script.*

```
#! /bin/sh
# Current SunOS versions
OSVERSIONS="4.1.1 4.1.2 5.0"
# Current Default SunOS version
OSVERSION=4.1.2
# Current Application Architectures
ARCHS="sun3 sun4"
# Current Kernel Architectures
KVMARCHS="sun3 sun3x sun4 sun4c sun4m"
# Current Window Systems
WINSYS="sunview openwin openwin/3.0 NeWS X11 X11/R4 X11/R5"
ECHO=/usr/5bin/echo

TOP=$1
if [ "$TOP" = "" ]; then
      echo 'you must specify the top level directory'
      exit 1
fi
```

```
mkarchdirs() {
      CURRTOP=$1
      shift
      for CURRENTARCH
      do
            for CURRENTOS in $OSVERSIONS
            do
                  mkdir $CURRTOP/$CURRENTARCH.sunos.$CURRENTOS
            done
            rm -f $CURRTOP/$CURRENTARCH
            ln -s $CURRENTARCH.sunos.$OSVERSION $CURRTOP/$CURRENTARCH
      done
}
mkmantree() {
      for TYPE in man cat
      do
            for SECT in 1 2 3 4 5 6 7 8 l n
            do
                  mkdir $1/$TYPE$SECT
            done
      done
}
mkgenericfset() {
      FSETTOP=$1
      shift
      for CURRDIR
      do
            mkdir $FSETTOP/$CURRDIR
            if [ "$CURRDIR" = "man" ]; then
                  mkmantree $FSETTOP/$CURRDIR
            fi
      done
}
mkapplfset() {
      mkgenericfset $1 adm bin data etc games games/lib lib kvm share
      for CURRENTWIN in $WINSYS
      do
            mkdir -p $1/$CURRENTWIN
            mkgenericfset $1/$CURRENTWIN bin lib
      done
}
```

```
mksharedfset() {
      mkgenericfset $1 adm bin data doc dict etc games games/lib include lib\
man src
      for CURRENTWIN in $WINSYS
      do
            mkdir –p $1/$CURRENTWIN
            mkgenericfset $1/$CURRENTWIN bin lib include man
      done
}
mkmajorfset() {
      if [ –d $1 ]; then
            mkarchdirs $1 share $ARCHS
            mkdir $1/kvm
            mkarchdirs $1/kvm $KVMARCHS
            for CURRARCH in $KVMARCHS
            do
                  for CURROS in $OSVERSIONS
                  do
                        mkgenericfset $1/kvm/$CURRARCH.sunos.$CURROS bin lib
                  done
            done
            for CURRARCH in $ARCHS
            do
                  for CURROS in $OSVERSIONS
                  do
                        mkapplfset $1/$CURRARCH.sunos.$CURROS
                  done
            done
            for CURROS in $OSVERSIONS
            do
                  mksharedfset $1/share.sunos.$CURROS
            done
      else
            echo "directory $1 doesn't exist"
      fi
}

mkmajorfset $TOP
```

Using The NFS Automounter

WRITTEN BY

***Thomas Lundgren* - TN/ETX/TX/DDU - ERICSSON TELECOM**

ABSTRACT

This article explains how to make use of the automounter with the directory hierarchy explained in document ETX/TX/DD-91:110 Uen Rev. B.

"Normally" you would need to edit all your NFS clients `/etc/fstab` files so that the correct file sets and other file systems get mounted. But if you use the automounter you can leave them as they are after the clients have been created.

INTRODUCTION

This article is based on Ericsson Telecom document `TX/DD-91:111 Uen Rev. B`.

You can read about how the automounter works in the on-line manual page `automount(8)` and in the *SunOS 4.1 System & Network Administration* binder at page 439.

NFS client automount map tree

The way in which the automounter is started in `/etc/rc.local` causes the automounter to first read the contents of the NIS map `auto.master`.

When you write the contents of this map and all other maps you should always begin with referencing *file maps*, not *NIS maps*. In the end of each file map you may or may not decide to include NIS maps.

You can use environment variables when writing automount maps. This makes it possible to write maps in a general way which makes them suited to be distributed by NIS. This is explained in the next section. To cause the clients to mount differently all you have to do is to set up the environment variables differently on different clients.

And therefore the NFS clients seldom needs special entries in their maps tailored after each clients specific requirement. Instead, they can all use the same generic NIS map. So in the normal case all the client's file maps do is include a NIS map. Therefore the *file map* `/etc/auto.direct` usually looks like the one showed in Picture 1.

Since the maps always begin with referencing a file map, it is possible to add entries for one single client which you couldn't do if you only referenced NIS maps. If you only referenced NIS maps then all you can do is to change the automount's interpretation of each entry by using environment variables, but you can't change the number of entries.

So if you want to add an automount entry for one single client you could add the entry in Picture 2 to `/etc/auto.direct` on that client.

Picture 1 Normally all the automount file maps do is include a NIS map.

```
+auto.direct
```

Picture 2 Using a file map to add automount entries for a client.

```
/mnt_special server:/export/home/friend
+auto.direct
```

NFS server automount map tree

When using the automounter on a server you will run into some small complications.

When using file maps you can append entries to the NIS maps, but servers usually have some of these file systems on their own local disk and therefore you don't want them to go and automount themselves. For this there exist the possibility to nullify maps already in NIS. But my opinion is that it looks a bit messy. A more straight approach would be to write totally new server specific file maps for this purpose. You only need file server specific maps for the maps describing mounts of file systems already on the servers local disks. The rest of the NIS maps can be used as they are.

Picture 3 NFS server's `auto.master` *file map.*

```
/-                  /etc/auto.direct.server
/usr/remote         /etc/auto.remote
/usr/_project       /etc/auto.project
/home               /etc/auto.home
/net                /etc/auto.net
/nse                /etc/auto.nse
/repl               auto.repl -ro
```

Picture 4 `auto.master` *NIS map.*

```
/-                  /etc/auto.direct
/usr/remote         /etc/auto.remote
/usr/_project       /etc/auto.project
/home               /etc/auto.home
/net                /etc/auto.net
/nse                /etc/auto.nse
/repl               auto.repl -ro
```

Though the automounter doesn't NFS mount local file systems I choose not to be dependent on that the automounter is running to get file sets like `/usr/local` mounted on the server. For FS like these I use the `/etc/fstab` file.

To use the correct maps you have to get control of which maps are read from the very beginning, and the best way to do this is to write a special master map for the server. And automount must be started with an option that tells it not to go and read the `auto.master` *NIS map* but instead go and read the file servers own `/etc/auto.master` *file map*.

But on the NIS master that would reference the file that the NIS map i build from. So a NIS master needs to have one `/etc/auto.master` which NIS is build from, and a special one for the server itself. You could call this file `/etc/auto.master.server` to avoid file name collision. And to be consequent you do the same on other machines with local disks that needs special automount maps, even if they are not NIS masters and or NFS servers. Our server's `/etc/auto.master.server` file is listed in Picture 3.

This file map doesn't include the NIS map `auto.master` because we don't want the contents of that map for the server, and this was the reason for creating the file server specific map in the first place. Our `auto.master` NIS map is listed in Picture 4.

By default the automounter reads the contents of the `auto.master` NIS map. So on the file servers, the automounter have to be started with different options in `/etc/rc.local`. In fact, the script can decide which method that will be used depending on if the server specific file `/etc/auto.master.server` exists or not. The commands to start the automounter is showed in Picture 5.

Picture 5 Bourne shell commands to start the automounter selecting the appropriate method by checking if the file map `/etc/auto.master.server` *exists or not.*

```
if [ -f /usr/etc/automount ]; then
      /bin/rm -rf /tmp_mnt/auto*
      if [ -f /etc/auto.master.server ]; then
            automount -m -f /etc/auto.master.server &
      else
            automount &
      fi
fi
```

Automount environment

The automounter allows environment variables to be used directly in automount maps. By default it recognizes the **ARCH** variable. It doesn't matter if it is set or not. If not set, it will be given the value of the output from the `arch(1)` command.

No other directly useful variables are set in `/etc/rc.local`, as I see it, that can be used in automount maps. But that can be changed. First I list the different environment variables that I like to have in the environment:

ARCH The application architecture name.

KARCH The kernel architecture name.

OSVER The SunOS version number.

SERVER This variable holds the host name of the client's file server. It has already been set in rc.local but it hasn't been exported.

APPLSRV The host name of the application server. Maybe you can't afford having a complete set of your local applications installed on every file server. So if you have two servers and only one of the two servers have installed applications in `/export/local`, then the NFS clients of the other server needs one additional hostname in the environment.

REGSRV The hostname of the regional server. Some files like database files are not suited to be stored in many copies on several servers. Therefore you should select one server to be your regional server.

For example you may have 4 application servers but only 1 regional server. The automouter will mount the regional server's `/export/regional` on `/usr/regional`.

MAILSRV The host name of the server carrying the common mail area, the `/usr/spool/mail` directory.

CALSRV The host name of the server carrying a common area for files generated by the calendar program. If shared this way the user can login to any machine and use calendar.

NEWSSRV The host name of the server carrying the net news (`nn`) article database.

Different clients may choose different servers from where to mount files. And we don't want to have several different versions of the `/etc/rc.local` file because the environment needs to be set up differently on different machines. Therefore these values are stored in a configuration file in the `/etc` directory. I choose to name this file `/etc/automount.conf`. Example of an automount configuration file is shown in Picture 6.

If this file is the same among all clients then it can be moved to `/usr/share/lib/automount.conf` and a symbolic link created from the clients `/etc` directory. To read in these settings and export the values to the environment I've written a special shell script named `automount_boot`.

The configuration file in Picture 6 is actually Bourne shell command file which is read by the script `automount_boot`. Therefore you can use any Bourne shell commands like `if-then-else`. So there is great flexibility. The environment variable **DURATION** isn't used in the automount maps but in the `automount_boot` script when the automounter is started. I just thought that the configuration file was a

Picture 6 Sample /etc/automount.conf file.

```
# The time in seconds the FS can
# be inactive before dismounted.
autodefine DURATION 3600
# If you want to mount file sets
# from a specific SunOS version
# define the value here or leave
# undefined to get the default
# version.
autodefine OSVER ".sunos.4.1.1"
# Name of the NFS server
autodefine SERVER
# Name of the application server
# from where to get the /usr/local
# file set.
autodefine APPLSRV bergman
# Name of the regional server
autodefine REGSRV leander
# Name of the mail server
autodefine MAILSRV garbo
# Name of the calendar server
autodefine CALSRV garbo
# Name of the news server
autodefine NEWSSRV bergman
```

good place to define this value. `autodefine` is a convenience routine to set different environment variables and the routine is listed in Picture 7.

`automount_boot` makes it easier to start and stop the automounter when so many environment variables are used. Because if you kill the automounter and then want to start it again you must set up the automount environment or else the automounter will not be able to interpret the automount maps. So this script reads the configuration file `/etc/automount.conf`, sets up the environment, and starts the automounter.

The shell script `automount_boot` should be called when the system boots. The part of `/etc/rc.local` that starts the automounter should be replaced by commands shown in Picture 8.

Picture 7 Convenience routine `autodefine` *used in the automount configuration file.*

```
autodefine () {
        eval $1=\"$2\"
        export $1
}
```

Picture 8 Part of `/etc/rc.local` *which starts the automounter.*

```
if [ -f /usr/share/adm/automount_boot ]; then
        /usr/share/adm/automount_boot
fi
```

Picture 9 Automount direct NIS map "auto.direct".

```
/usr/local \
      /                 $APPLSRV:/export/local/$ARCH$OSVER \
      /kvm              $APPLSRV:/export/local/kvm/$KARCH$OSVER \
      /share            $APPLSRV:/export/local/share$OSVER \

/usr/regional \
      /                 $REGSRV:/export/regional/$ARCH$OSVER \
      /kvm              $REGSRV:/export/regional/kvm/$KARCH$OSVER \
      /share            $REGSRV:/export/regional/share$OSVER

/var/spool/mail         $MAILSRV:/var/spool/mail
/var/spool/calendar     $CALSRV:/var/spool/calendar
/var/spool/news         $NEWSSRV:/var/spool/news
```

Picture 10 Automount direct NIS map "auto.direct" after variable substitution. **OSVER** *is set to NULL.*

```
/usr/local \
      /                 bergman:/export/local/sun4 \
      /kvm              bergman:/export/local/kvm/sun4c \
      /share            bergman:/export/local/share \

/usr/regional \
      /                 leander:/export/regional/sun4 \
      /kvm              leander:/export/regional/kvm/sun4c \
      /share            leander:/export/regional/share

/var/spool/mail         garbo:/var/spool/mail
/var/spool/calendar     garbo:/var/spool/calendar
/var/spool/news         bergman:/var/spool/news
```

Now with the environment set I can explain which types of automount maps that are needed and where.

Automount point /usr/local

The mounting of the file sets below `/usr/local` is a hierarchic mount scheme. So I use a direct map and I use the special map entry format that will cause the automounter to mount all file sets in one step. You only need one direct map so I also add the other entries that will mount `/var/spool/mail`, `/var/spool/calendar`, and `/var/spool/news`. The map is listed in Picture 9 on page 20.

The variables will be substituted with their values. The values can be null. So if the variable **`OSVER`** has a null value then the default SunOS version is selected. The automount entry after variable substitution is listed in Picture 10 on page 20.

So these mount-paths will point to the symbolic links. Now, if the client wants to run a specific SunOS version all that's needed is that the environment changes. And the environment variable **`OSVER`** must have a special format so that the mount-path is legal. For example if the client should mount file sets from the SunOS release 4.1 the environment variable **`OSVER`** must be set to **`"`**`.sunos.4.1`**`"`**. Observe that it begins with a dot.

Automount point /usr/regional

The automount map entries that describes how the automounter should mount `/usr/regional` from the regional server is of the type direct entries and is also included in the `auto.direct` NIS map above.

If you don't want to have a special partition for `/export/regional` then `/export/regional` can be a symbolic link to `/export/local` on the regional server. Or it can be a symbolic link to `/export/local/regional` if you want the files to be separate from the `/export/local` files. This would be better if you later decide to have a special file system for these files.

Automount point /usr/remote

Now it would also be nice to be able to mount *any* server's `/export/local`. So in Picture 11 a real *wild card* map is listed that will mount any server's `/export/local`

This map changes to mount from whatever server you are trying to access below `/usr/remote`. For example, if you type the command `cd/usr/remote/bergman`, the server bergman's `/export/local` will be mounted on `/usr/remote/bergman`. You can try any hostname, and the map will change to mount that hosts `/export/local`.

The `auto.remote` map can be used if you have several application servers, and you have a very large software package that you only afford to have installed on one server or for some other technical reason can't have installed on more than one server.

Now, remember that many software packages have hard coded paths or have the location of their auxiliary files recorded during configuration. So you can't just reference the software directly through `/usr/remote`.

If the server bergman has X11 R5 installed in `/export/local/sun4/X11/R5` and `/export/local/sun3/X11/R5`, then for the

Picture 11 Automount indirect NIS map "auto.remote".

```
* \
/           &:/export/local/$ARCH$OSVER \
/kvm        &:/export/local/kvm/$KARCH$OSVER \
/share      &:/export/local/share$OSVER
```

Picture 12 Example of symbolic links that have to be made to enable NFS clients to use software installed on a remote server.

```
/export/local/sun4.sunos.4.1/X11/R5 -> /usr/remote/bergman/X11/R5
/export/local/sun4.sunos.4.1.1/X11/R5 -> /usr/remote/bergman/X11/R5
/export/local/sun4c.sunos.4.1/X11/R5 -> /usr/remote/bergman/X11/R5
/export/local/sun4c.sunos.4.1.1/X11/R5 -> /usr/remote/bergman/X11/R5
/export/local/sun3.sunos.4.1/X11/R5 -> /usr/remote/bergman/X11/R5
/export/local/sun3.sunos.4.1.1/X11/R5 -> /usr/remote/bergman/X11/R5
```

Picture 13 Automount indirect NIS map "auto.package".

```
* \
/     $ALLAPPL:/export/local/packages/$ARCH$OSVER/&
```

server garbo and her clients to be able to access X11 R5 on bergman the symbolic links in Picture 12 on page 21has to be created on the server garbo.

When sun3 and sun4 clients have mounted their `/usr/local` major file set the link will be visible at `/usr/local/X11/R5` for both architectures.

So when `/usr/local/X11/R5` is referenced one is redirected to `/usr/remote/bergman/X11/R5`. The automount map takes care of mounting the right application file set from bergman.

So in the future if you have an application on some other server, or even a server at another department, and you quickly want to make it available on your local server, all you have to do is to make symbolic links in your `/export/local` major file set.

Automount point `/usr/package`

If you create the symbolic links in Picture 12 on page 21 the remote server bergman will be called every time the X11 package is used. And if the server is down then the mount fails.

If the requested package was available on several NFS servers it would be nice if the package could be mounted from any of the NFS servers. As you know the automounter supports this what's called *Replicated Filesystems*. You can list several hostnames separated by commas, in a *multiple location field*, in one single automount entry. The automounter sends multiple mount requests and it mounts the file system from the first NFS server that replies to the mount request.

The `automount(8)` manual page says; *"This request is first made to the local net or subnet. If there is no response, any connected server may respond."*

My experience of the automounter is; If you have a multiple location field with the three hostnames `garbo`, `leander`, and `bergman` where `garbo` is the local hosts, then the other NFS servers will not get any mount requests. The automount will first and only consult the local host. So if the mount path doesn't exist on `garbo` the mount fails, even if there are other NFS servers listed who might have the requested NFS file system. If the local host isn't listed in the multiple location field then the mount will fail only if it fails on all listed hosts.

Therefore automount maps with multiple location fields never work on NFS servers if they doesn't carry the specified NFS file system.

I've seen other automount maps with a multiple location fields and which purpose is to mount installation directory trees each containing only one specific software. But they usually requires that you specify the location of the software and that you have to add one entry for each installed package (software).

My first goal when writing the automount maps is to write maps that never need to be changed. Because of this you don't need to edit the automount maps and push them out on the network and in case of direct maps you don't need to restart the automounter or reboot all your 187 NFS clients.

The indirect automount map for the automount point `/usr/package` is shown in Picture 13.

For the multiple location field I introduce the environment variable **`ALLAPPL`** which holds a comma separated list of hostnames. The variable is set with the others in `/etc/automount.conf`, this is shown in Picture 14. This should be the list of all your application servers.

The servers must support the possible mount paths in the automount map. One path could be `/export/local/packages/sun4`. In the `packages` directory you create symbolic links to the real location of the software's installation directory tree.

Now back to the original problem where you needed to access some software installed on a remote server. Well, you still have to implement the symbolic links but this time, instead of referencing through `/usr/remote` you reference `/usr/package/`*packagename* and you will mount it from the NFS server which answers first, normally the one with the less load.

Picture 14 Setting environment variable **`ALLAPPL`** *in* `/etc/automount.conf`.

```
autodefine ALLAPPL garbo,bergman
```

Picture 15 Automount indirect NIS map "auto.home".

```
*       &:/export/home/&
```

Picture 16 Example of an automount indirect NIS map "auto.home" when the key value has been established.

```
bergman bergman:/export/home/bergman
```

Picture 17 Example of symbolic links to the real location of home to support the /export mount paths.

```
# bergman:
# Correct location of home, no link.
/export/home/bergman
# garbo:
/export/home/garbo -> /home/garbo
# leander:
/export/home/leander -> /export/home
```

Picture 18 Automount indirect NIS map "auto.project" Type A.

```
* \
/       &:/export/project/&
```

Automount point `/home`

Here I'll use an indirect map of a generic format and it's listed in Picture 15. And as you know this is a *catch all entry*. So if someone changes current directory to `/home/bergman` the key changes to whatever is accessed below `/home` and the entry changes to the one listed in Picture 16.

This scheme does not require that all servers mount their home partitions on `/export/home/servername`. All it requires is that the export mount path is supported. This could therefore be a symbolic link to the real location of the home partition. Picture 17 list some examples of symbolic links to the real location of the home FS.

So you don't need to write a complex automount map because the servers store their home partitions differently. Though the server garbo in this example will not be able to use the automounter on `/home`, guess why!

It is still possible to support multiple home partitions and still only having the catch all entry above. What you do is that you let a server with more than one home partition figurate with more than one hostname in the NIS `hosts` map.

So if the server garbo has more than one home partition the server could instead be named `garbo-A` and `garbo-B`, and the two home partitions could be mounted as `/export/home/garbo-A` and `/export/home/garbo-B`. Using the same automount map as above users can access the two home file systems with the paths `/home/garbo-A` or `/home/garbo-B`. One could also view `garbo-A` and `garbo-B` as virtual home-hosts, and the server garbo could then keep its original name `garbo`.

Automount point `/usr/project`

This section explains how the automounter used with the different project areas. I'll use an indirect map for this point too, and it will basically be used in the same way as the indirect NIS map `auto.home`, the map is listed in Picture 18.

Picture 19 Automount indirect NIS map "auto.project" Type B.

```
* \
/           &:/export/project/&/$ARCH$OSVER
/kvm        &:/export/project/&/kvm/$KARCH$OSVER
/share      &:/export/project/&/share$OSVER
```

This version of the map doesn't support the directory hierarchy the way the direct and indirect maps that mount `/usr/local`, `/usr/regional`, and `/usr/remote` does. To support the directory hierarchy the map has to be changed to the one listed in Picture 19 on page 23. Before I explain how the map work I'll explain how the different project areas should be mounted.

On large servers you may have several project areas that are so large that only one fit in on each disk. And therefore you get multiple mount points, one for each project area.

These project areas should all be mounted below `/export/project` on the file server. Each mount point is given the name of the server and a count. The count begins with capital `A` and continues upwards. The first project area on server garbo should be mounted on `/export/project/garbo-A` and the second on `/export/project/garbo-B` and so on.

You see that the name choice is the same as when you supported multiple home partitions above. So the hostname aliases are used both for home partitions and project areas (partitions).

Now, the automount point that is going to be used is not `/usr/project` but rather `/usr/_project`. The users shouldn't know about this. The reason for mounting on `/usr/_project` is that we need to present a *fake view* for the users below `/usr/project`.

But let's examine how the automount map works. For example, the server bergman has two project areas, `bergman-A` and `bergman-B`. To access the area `bergman-B` you could change the current directory to `/usr/_project/bergman-B`. The automounter then exchange the key with the value of what is being accessed below `/usr/_project`. The automount entry after key and variable substitution is listed in Picture 20.

So this final entry selects the correct file sets from the default SunOS version from `bergman-B:/export/project/bergman-B` directory and mounts it on `/usr/_project/bergman-B`.

Now you can add any number of project areas without having to edit the automount maps. All you have to remember is to check that you have enough hostname aliases for the server in the NIS `hosts` map.

Below `/usr/project` we will create the view the users see. First a list where we have a couple of example projects (`tour`, `movie`, `portrait`, and `perfume`) is presented in Picture 21.

The strategy is to hide from the users the real location of their project areas. This is why I chosed to let the automounter mount the project areas on `/usr/_project` in the first place. It is basically the same as when you use the automount special map `-passwd`. When you use this map the home partitions get mounted on `/homes` while the users accesses `/home`. As you know the automounter creates many symbolic links from `/home` to `/homes`.

So you have one directory through which you access the home, but it is on another that the partition actually is going to be mounted. The same thing with `/usr/project` and `/usr/_project`.

Picture 20 Example of an automount NIS map "auto.project" Type B after variable substitution.

```
bergman-B \
/           bergman-B:/export/project/bergman-B/sun4
/kvm        bergman-B:/export/project/bergman-B/kvm/sun4c
/share      bergman-B:/export/project/bergman-B/share
```

Picture 21 Directory listing of NFS server garbo's project areas.

```
/export/project/garbo-A
/export/project/garbo-A/tour/
/export/project/garbo-A/movie/
/export/project/garbo-B/portrait/
/export/project/garbo-B/perfume/
```

The problem here is that the automounter will not create all the symbolic links for us and so we have to create them manually. So for the users to be able to access their projects in `/usr/project` the symbolic links in Picture 22 has to be created.

As you see the users doesn't need to know where their project areas are stored. Creating the links is a piece of cake. You don't have to create them one by one by hand. This is what you do; you change directory to `/usr/_project` and make an access to each project area you want to be visible to the users and then you create all links in one step in a `foreach` loop. This map is a generic map so there may be hundreds of project areas together on the network. Your department's and other department's project areas.

How to create the links is shown below in Picture 23.

Mixing type A and type B maps

If you need to support both types of the `auto.project` automount map you should create the maps `auto.projectA` and `auto.projectB`. These indirect maps are then used with the automount-points `/usr/_projectA` and `/usr/_projectB`. Symbolic links are still made from `/usr/project`.

Adding new automount maps to NIS

Now there isn't much to explain here. One should always make it simple. There already exists an automount map in NIS, the `auto.master` NIS map. So you could just use those entries as templates and duplicate everything that has to do with `auto.master`. Then change `auto.master` to `auto.project`, `auto.remote` or whatever.

Picture 22 Symbolic links to direct the users to the correct project areas shown in picture 19.

```
/usr/project/tour -> /usr/_project/garbo-A/tour
/usr/project/movie -> /usr/_project/garbo-A/movie
/usr/project/portrait -> /usr/project/garbo-B/portrait
/usr/project/perfume -> /usr/project/garbo-B/perfume
```

Picture 23 Commands to make the symbolic links from /usr/project to the desired project areas.

```
cd /usr/_project
# The following activates the desired mounts
ls garbo-A/ garbo-B/ bergman-A/ bergman-B/ bergman-C/ secret-X/
# Next we make the links automagically
cd /usr/project
foreach link ( /usr/_project/*/* )
      ln -s $link
end
```

APPENDIX

Ericsson automount fix in `/etc/rc.local`

```
############ ERICSSON AUTOMOUNT FIX ############
#
SERVER=$server
export SERVER
if [ -f /usr/share/adm/automount_boot ]; then
      /usr/share/adm/automount_boot
fi

#
########## END ERICSSON AUTOMOUNT FIX #########
```

`automount_boot` Bourne shell script

```
#! /bin/sh

ECHO=/usr/5bin/echo

# Routine to show that something is happening when waiting for
# the automounter to finish initialization.

autowaitsign () {
      case "$autowsign" in
      "" | "|" )
            autowsign="/";;
      "/" )
            autowsign="-";;
      "-" )
            autowsign="\\\\";;
      "\\\\" )
            autowsign="|";;
      esac
      $ECHO "$autowsign\b\c"
}

# convenience routine to be used by the automount.conf file
autodefine () {
      eval $1=\"$2\"
      export $1
# Comment out the follwing line if you don't want the
# little spinning sign.
      autowaitsign
}
```

```
$ECHO ' auto\c'

# Define DURATION to a default value so that it is set
# if not set in automount configuration file.
autodefine DURATION 300
# Define the rest of the "standard" environment.
autodefine ARCH `arch`
autodefine KARCH `arch -k`
if [ "$SERVER" = "" ]; then
      autodefine SERVER `hostname`
fi
# Read the configuration file and start the automounter
if [ -f /etc/automount.conf ]; then
      . /etc/automount.conf
      if [ -f /usr/etc/automount ]; then
            /bin/rm -rf /tmp_mnt/auto*
            if [ -f /etc/auto.master.server ]; then
                  automount -m -tl $DURATION -f /etc/auto.master.server
            else
                  automount -tl $DURATION &
            fi
            $ECHO 'mount\c'
      else
            $ECHO 'mount(failed)'
      fi
else
      $ECHO 'mount(failed)'
fi
```

`automount_halt` Bourne shell script

```
#! /bin/sh
ECHO=/usr/5bin/echo
RUNNING=`ps -xc | grep automount | grep -v grep | awk '{ printf("%s",$1)}'`
$ECHO " "
if [ ! "$RUNNING" = "" ]; then
      $ECHO "Automounter is running ..\c"
      kill $RUNNING
      for sec in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
      do
            RUNNING=`ps -xc | grep automount | grep -v grep \
                  | awk '{ printf("%s",$1)}'`
            $ECHO ".\c"
            if [ "$RUNNING" = "" ]; then
                  $ECHO " killed."
                  $ECHO "Successfully killed the automounter"
                  $ECHO " "
                  exit 0
            fi
      done
      $ECHO " still running"
      $ECHO "Automount is hung, can not kill it"
else
      $ECHO "AUTOMOUNTERISNOTRUNNING"
fi
$ECHO " "
```