# Creating a Package Context Unix Environment

George Morrison
Lear Corporation
5200 Auto Club Drive
Dearborn, MI 48126
(313) 593-9942 Voice

## Abstract

The user (or client software) of a high availability environment perceives the package as though it were a complete and separate system.  This illusion is damaged by a number of problems:

- User accounts must be handled carefully or use an external package (such as NIS) to manage user logins.  In the case of NIS, accounts are more accurately viewed as cluster level accounts as opposed to package accounts.
- Cron processing occurs at the system level, and therefore is not a highly available service.  In addition, cron jobs are a maintenance problem, as crontabs must be kept in sync on all hosts that the package could move to.
- Print services, while as pervasive as in years past, still provide a critical function for many package-eligible applications (e.g., payroll).  As with cron, print spooling is a system service and thus is not a highly available service.

The cluster administrator can also be faced with a significant security problem when a single system hosts multiple packages.  When a system is hosting multiple packages, the system security must not only protect the package applications from misuse or attack from that package's users, but also from the users of other packages that may be present.  While judicious use of standard Unix security tools and techniques may be sufficient, the task of securing one package from the users of all other packages can be a daunting task.

The solution to these problems, detailed in this paper, is the creation of the Package Context Unix Environment (PCUE).  This technique, as presented, can be applied to any hardware that supports MC Service Guard, and provides:

- Per-package user accounts
- Per-package cron processing
- Per-package print services
- High degree of package-to-package file system security
- Virtually no additional per-package patch issues

## Introduction

A new way to look at and use high availability packages as accomplished by the Lear Corporation systems is explained herein.  The paper begins with a "before picture" of the use of packages at Lear.  The theory behind the architecture is presented next, followed by a discussion of some of the key technology used to implement the system.  Finally, the solutions to the problems are presented.

## The Problems

The high availability environment created by MC/Service Guard is a two edged sword.  Of course, it provides an infrastructure that increases application availability to levels appropriate for business critical applications.  But the administration of this environment can be quite complex.  And from the user's perspective, the package is a separate system.

### *User Accounts*

User account management is complicated in a variety of ways as a result of the package moving from system to system.  A typical solution to this problem is the use of NIS, which may not be appropriate or palatable in all cases.  In the HP World '99 paper, High Availability Shell Scripts, an alternate solution was presented that uses PAM to authenticate users against a password file located on a package file system, yielding package-based user accounts.  An additional technique will be presented in this paper that provides the same functionality.  Although both techniques require use of custom code, the technique presented in this paper may not require as much administrative overhead.

## *Package-to-Package Security*

The issue of security is significant in any computer system that performs meaningful business functions. Much has been written on this subject, and many computer professionals have made successful careers in the computer security business. Highly available systems using MC Service Guard pose additional security issues beyond those present with “stand-alone” computer systems. Not only does the host system (node) require securing, but the package must be considered as a separate security issue, with it’s own security requirements separate and distinct from the security requirements of the node.

In part, this problem is eased by having package-based user accounts, as detailed above. However, a significant security problem occurs when more than one package can be hosted on a single node. In this case, the node has security requirements, each package has security requirements, and each package must be made secure from the other packages. For example, the node must protect against unauthorized logins. A package must insure that users or processes that run on that package do not abuse the services provided by the package. And, there must be insurance that users of one package do not abuse the services of another package when both packages are hosted on the same system.

Consider the case where a user has a package-based account on a package. The package on which the account is based is hosted on node that also hosts another package (either by design or as a result of a fail-over event) that has more stringent security requirements than the package where the user has an account. In such a case, the administrative challenge is to be sure the user has adequate security to the resources on the package upon which the account is located, but no access to the resources on the package(s) that the user does not have access to.

## *Cron Processing*

The cron scheduling facility of HP-UX presented another challenge. Lear needed to schedule package context activities, but cron is inherently a system context facility. If a crontab file references a non-existent file, an error would be generated. This is not exactly the end of the world, but a more elegant solution is called for.

The initial Lear solution attacked the problem by requiring each crontab entry to utilize a precursor script. This script (`runcron`) took arguments specifying the package that the batch job is intended to run on, and the command that is to be run. The precursor script verifies that the requested package is present on the node before allowing the job to run. If the package is present, `runcron` sets the environment by switching to the package context and active subsystem. If the requisite package is not on the node, the script makes a log entry noting that fact and exits without error.

The Lear's initial handling of cron also mandated that the exact same crontab must exist on every package. For example, if jobs A, C, and Q are to run on pkg01, jobs B, D, E, M, N, and O are to run on pkg02, and jobs F - L and P run on pkg03, then all jobs, A-Q are on the crontab of pkg01, pkg02, and pkg03. The crontab entries all utilize `runcron`. If all three packages are on different nodes, the pkg01 will get "no such package" log messages (but no cron errors) for jobs B, D - P. Similarly, pkg02 will get "no such package" log messages for jobs A, C, F - L, and P; pkg03 will get "no such package" log messages for jobs A - E, M - O, and Q.

This solution works, but is the precursor script and requirement for the crontab to be the union of all the packages is cumbersome. These requirements also introduce the possibility of errors (not keeping crontabs up to date on all nodes, for example) and add confusion and a learning curve for new users/administrators.

### *Print Services*

Printing also can be an administrative problem. If host-based printing is used, a print queue must be created on each node that the package could be hosted. Again, this problem is compounded as the nodes in a cluster increase. In addition, print jobs in process on a failing node are lost. This is a problem in environments where printouts are business critical (e.g., printing invoices, billing, etc).

## Theory

### *Review of High Availability Contexts*

The concept of High Availability Contexts was introduced in High Availability Shell Scripts; a review of the theory is reproduce here.

A node is a computer in a cluster. Being a member of the cluster adds functionality to the system, but does not alter the fact that the node is a computer system in and of itself. Thus, each node performs certain duties and functions that do not relate to high availability, just as the node would if it were not a member of a cluster.

The package must be, more or less, self-contained. The package depends on a node to do anything useful, of course. However, as a minimum, the package must contain user/application data, and perhaps application code. Alternatively, it is possible to have application code loaded on any node upon which a package may be hosted.

The node, having duties as a member of the cluster and duties that are unrelated to high availability, can be thought of as having one or more personalities. The first personality is the system-centric personality and it is the only independent personality. This personality is the same one the system had before it was “infected” with high availability and the one the system would have if the system was no longer a member of the cluster. Lear refers to this “personality” as the System Context, as in “within the context of the system personality.

The node also has a package-centric personality. The package must be as self-contained as possible, yet it is totally dependant on a node to do anything meaningful. In addition, the package can be passed around from one node to another, resulting in a like transfer of the personality. Lear refers to this conflicted personality as the Package Context.

The context was selected by setting an environment variable, `ctxROOT` to a `/` followed by the package name (for the Package Context) or to just a `/` (for the System Context). This was derived from the file system design that Lear uses: all file systems related to a package are mounted under a single mount point of the same name as the package. See Figure 1.

Note that these contexts, as originally introduced, were simple abstractions used to guide system administrators and scriptwriters. There are no technical or physical barriers preventing system administrators, programmers, or users from operating in all contexts simultaneously. And that is precisely the problem with the original Package Context: it can be totally ignored.

### *A New Beginning*

The Package Context, as originally implemented, was an administrative success. However, the holes in the theory soon became extremely evident to the system administration staff. These "holes" are explained in the Problems section and will not be repeated here. However, the addition of payroll software to the production systems forced the issue: Lear had to have a better, more secure and robust way to handle the Package Context.

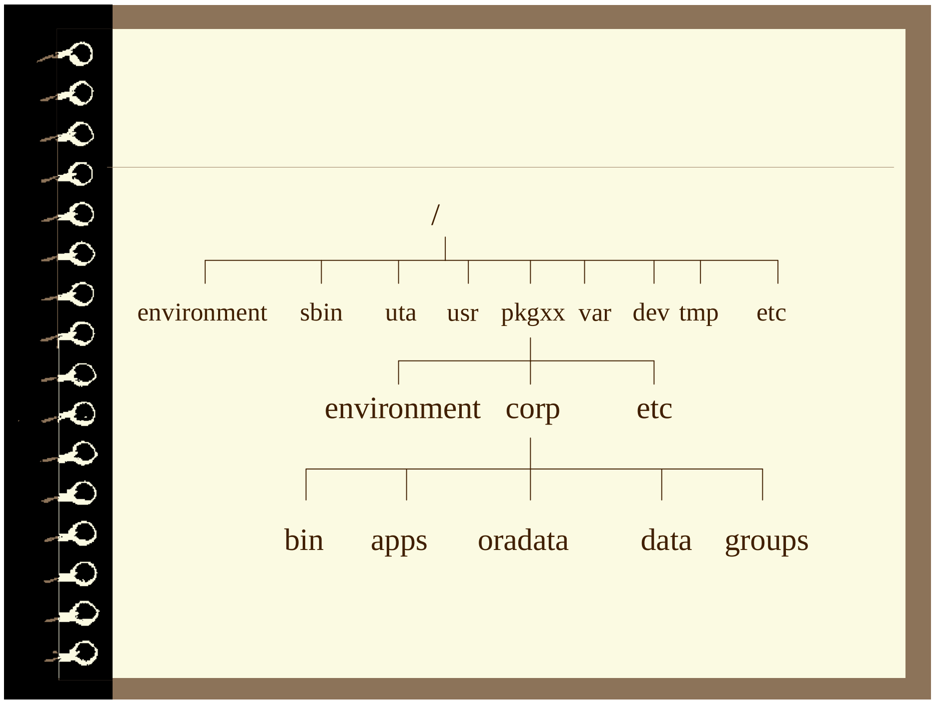


**Figure 1. Example Package Context File System**

The decision to place all package file systems under a single mount point (e.g., `/pkg02`) was somewhat arbitrary and was done solely for the sake of organization. However, on further review, an opportunity presented itself. If, somehow, we could fool the software running under the package into thinking that the package mount point was the root directory (i.e., `/`), then all file access would be confined to the directories that are "owned" by the package. (Dare I say the package file systems would be put into a lock box?) Fortunately, HP-UX (and most other Unix flavors) offers a system call and/or command to do exactly that: `chroot`. The use of `chroot` is actually quite common, although one would guess very few people actually are aware of it. Anonymous FTP uses the `chroot` technique to protect the system from malicious attacks, curious users, etc. In HP-UX, a command (`chroot(1m)`) is provided for administrators to exercise the `chroot` system call. The syntax of the command is:

```
chroot directory command
```

The `chroot` command changes the root directory of the process to *`directory`* and then execs *`command`*. The root directory is a property of processes that is inherited across `fork` and `exec` system calls. The `chroot` command (and by extension, the system call) is standard HP-UX stuff; it is not specific to MC/Service Guard. Anyone with access to the HP-UX `chroot` command[1] can do the following test:

```
# mkdir ~/myroot

# cp /usr/bin/ls ~/myroot

# cp /sbin/sh ~/myroot

# chroot ~/myroot /sh

# ls
```

[1] The chroot system call and chroot command is typically restricted to superuser or equivalent accounts.

```
ls sh

# cd /usr

cd: No such file or directory

# cd /etc

cd: No such file or directory

# pwd

/

# mkdir test

# ls

ls sh test

# ll

ll: Not found

# ls -l

-r-xr-xr-x   1 0  1  Jan 3 2000 ls

-r-xr-xr-x   1 0  1  Jan 3 2000 sh

drwxrwxrwx   2 0  1  Jan 3 2000 test

# cd test

# pwd

/test

# exit

#
```

Note the following after successfully executing the `chroot` command:

1. A new shell was spawned relative to the new root directory.

2. It is not possible to get to directories or commands outside of the new root directory.

3. The `ls` command cannot display the user name and group name of the owner/group of the files and directories. The `ls` command can only display the numeric user ID number and Group ID number. This is because the `ls` command typically uses `/etc/passwd` and `/etc/group` to translate the numbers into the user and group names. Since we did not copy or create these files, `ls` could only display the numbers.

This last observation drives home a point: if file(s) are not put into the file system(s) below the new root, they are not accessible. This is why `/sbin/sh` was used for the test: no shared libraries are required for this copy of the shell. Had the test specified running a program that attempts to load a shared library, the program would have failed. So, to try to run any programs in this New World created with the `chroot` command, we need the full compliment of standard, fully populated HP-UX directories: `/dev`, `/etc`, `/stand`, `/usr`, `/var`, etc.  And like everything else in Unix, there are at least two ways to get access to these directories.

### The Copy Technique

The brute force method to solving this problem would be to just copy all of the existing files to the file system where the new root directory would be (assuming root access and sufficient disk space):

```
# mkdir ~/newroot

# find /etc /stand /usr /var | cpio -pdumv ~/newroot

# find /opt /lib /dev | cpio -pxdumv ~/newroot
```

The problems with copying the standard file systems to the new root directory are file system space, updates to the kernel, and HP-UX patches.  File system space is a fairly easy question to address (and is the least of the problems); either you have the space and can use it for a copy of the HP-UX stuff, or you don't.

A copy of the kernel (vmunix) needs to be in `/stand` under the new root.  This is because many programs use this file to help them extract information from HP-UX.  Bad things happen if the kernel that is running is different from the kernel file that programs use for information.  In addition to being used by straight HP-UX programs, some third party applications rely on `/stand/vmunix` as well.  For example, some versions of Oracle will not start if `/stand/vmunix` is different from the in-core kernel.  So, in our brave new world, we need to be sure the package copy of the kernel stays in sync with the copy kernel used at boot time.  This limitation can be overcome by adding a command similar to the following to the package startup script:

```
cp /stand/vmunix /pkgwhatever/stand
```

Finally, the issue of patching must be addressed.  If a copy of the standard directories is made, then whenever the system is patched, the package copy of the files must be updated as well.  It may be possible to run a package copy of the HP-UX commands and libraries that are at a different patch level than the same files in the system context, but this would seem to invite problems that would be very difficult to positively identify.  Note that the `swinstall` command supports loading software to alternate root directories.  This would seem to allow the package copy of HP-UX files to be patched separately from the system context.  (Lear has not tested or researched this option).  However, this is another area that would seem to invite problems.  Additionally, the `swinstall` man page notes that swinstall does not configure the software loaded to the alternate root directory, so this would have to be done separately.  Good operational procedures may be able to handle situation such as these, but there is at least one other approach that is arguably more robust.

### The Mount Technique

HP-UX offers a method to "mount" a directory in more than one place.  No, this is not soft links. This is what would seem to be a little used technique called local file systems.  The use of local file systems is a Unix technique whereby a directory can be mounted as though it were a separate file system. Lear became aware of this technique totally by accident: a junior administrator mistyped a mount command!

Local file systems offer an elegant solution to the problem at hand.  By simply mounting the system context directories onto the package context directories, a full compliment of system directories can be made available to the package context with no increase in disk space use.  In addition, the issue of patch now can become moot: path the system context as normal, and the patched files appear in the package context by virtue of local file systems. Thus, the equivalent commands to the copy technique are:

```
# mkdir /newroot

# mkdir /newroot/etc

# mkdir /newroot/stand

# mkdir /newroot/usr

# mkdir /newroot/var

# mkdir /newroot/opt

# mkdir /newroot/lib

# mkdir /newroot/dev

# mount /etc /newroot/etc

# mount /stand /newroot/stand

# mount /usr /newroot/usr

# mount /var /newroot/var

# mount /opt /newroot/opt

# mount /lib /newroot/lib

# mount /dev /newroot/dev
```

Figure 2, below, shows a diagram of a HA package to which these techniques have been applied.

## ***Trouble in Paradise***

Once this new world order was established via the mount technique, a few disadvantages began to be evident. For example, it could be advantageous to have a var directory that was at least partially independent of the system's `var` directory. This was a clear advantage of the copy technique - the package had its own copy of *everything*. However, a package copy of some directories, e.g., `/var/sam`, would seem to hold little advantage.

There a couple of options to get the best of both worlds (some files the same as the system, some unique to the package) without resorting to the copy technique. The first would be to use more mounts. For example, take the case where /var/spool, /var/tmp and /var/adm is chosen to be unique to the package,

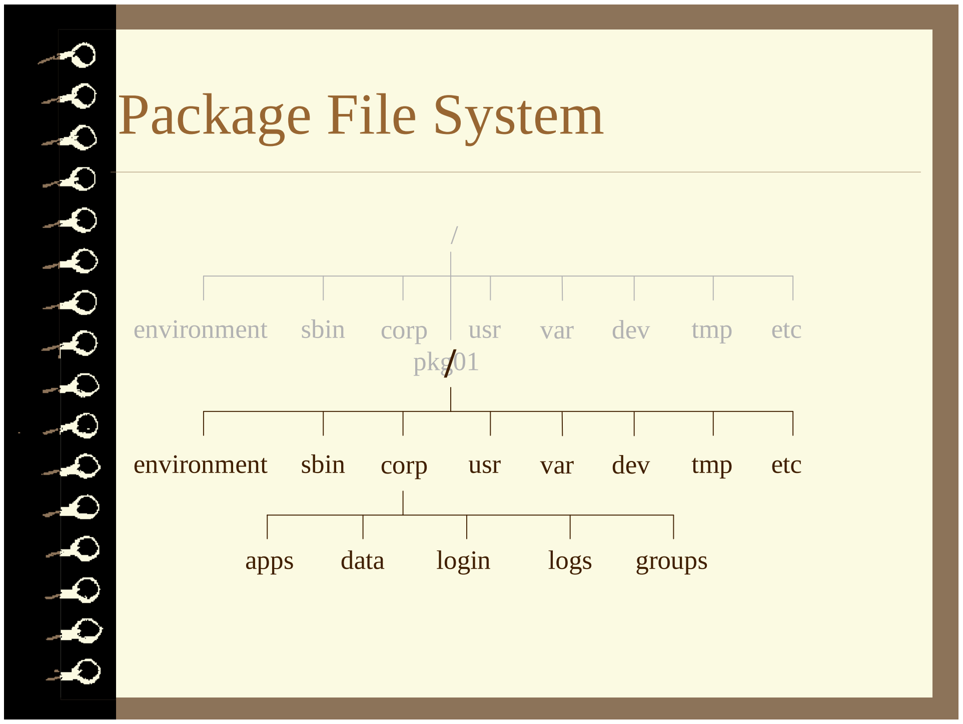

## Package File System

**Figure 2. Package File System after chroot call**

while /var/opt and /var/sam are elected to be the system directories. The mount technique would then be:

```
# mkdir /pkgxx/var

# mkdir /pkgxx/var/spool

# mkdir /pkgxx/var/tmp

# mkdir /pkgxx/var/adm

# mkdir /pkgxx/var/opt

# mkdir /pkgxx/var/sam

# mount /var/opt /pkgxx/var/opt

# mount /var/sam /pkgxx/sam
```

A second alternative to modify the mount technique, and the one that was ultimately chosen by Lear, was to mount the directories that are only partially used to a separate mount point in the package. Then soft links were created to the files and directories that are to be made visible in the package.

## *The Brave New World*

Having arrived at this point, one might ask, so what? Let’s reflect on what has now transpired. Assuming that everything has been setup with either the hybrid mount/soft link technique, the package context now is a separate little Unix world. The separation is not complete; the host is depended on for a kernel and for the "standard" Unix file systems (`/usr`, `/opt`, etc.). But the parts of the file system that are unique to the package can now host standard, file system-based Unix facilities. File system-based facilities, for the purposes of this paper, are those services whose core operation does not depend on any kernel-based IPC such as shared memory, semaphores, or message queues. The successful operation of file system-base facilities is defined solely by the file system in which the facility runs. Three such facilities are the system scheduler (`cron`), the printing spooler (`lpsched`), and user accounts.

The system scheduler operates by checking a configuration file and running programs whenever indicated by the configuration file. The scheduler does not make use of any kernel-based IPC that would conflict with another instance of the scheduler *so long as the two instances operated in separate file systems*. Once again, one does not need MC/Service Guard to accomplish this. On any HP-UX system, setup a separate file system with either the copy technique or the mount/soft-link hybrid technique (taking care to be sure `/var/spool/cron` and `/var/adm/cron` are distinct and separate from the system context of these directory hierarchies). You can then start a copy of the cron daemon in your pseudo package. You can now submit jobs in the package that are independent of any schedule set in the system context; `crontab –l` will report the different schedules; `/var/adm/cron/log` in each context will show the run-log for only the given context.

The print spooler also does not depend on anything other than separate file system to operate. Thus by insuring that the package has a separate `/var/spool/lp`, `/etc/lp`, and `/var/adm/lp`, it is possible to stop and start separate copies of lpsched. The lpstat command will report only on jobs and printers configured in the given context; changes to the print environment in one context is completely isolated from changes in any other context.

This separation of the printing scheduler into the package context has an important administrative effect. Printers must be now be created and maintained in each *package*; without the PCUE, printers must be created and maintained on each *node*. That is, in "classic" MC/Service guard, if a network printer is required for use with applications on pkg02 and pkg02 can be hosted on host1 and host2, then the printer must be created on both host1 and host2. With the PCUE, the printer need only be created and maintained in pkg02.

The information about user accounts is contained in the `/etc/passwd` file, and, to lesser degree, the `/etc/group` file. When a user logs onto a Unix system, the login program uses these files (or their trusted systems equivalent) to authenticate the user, the change the UID of the process to whatever is in the `passwd` file. Note that now there are <u>separate</u> `passwd` and `group` files for each package setup with a PCUE. This has a couple of implications. First, if someway can be found to run `login(1m)` from inside the PCUE, the system `passwd` file and the system `group` file does not need to contain the account information about package users. (A technique to run `login(1m)` from within the PCUE will be shown in the *Tools and Technologies* section titled "pkgnet - inetd Wrapper program"). However, it may be handy to continue to maintain a composite `passwd` file of all package users so that operations from the system context (e.g., `ps(1)`, and `ll(1)`) will be able to translate UID to a more meaningful user account name. In the case where no package users are in the system context passwd file, the security of the system context is enhanced significantly. On the other hand, if the user accounts are duplicated in the system context `passwd` file, there is no reason to have valid passwords for these accounts. Thus, the security for the system context is still enhanced, if not to the level of having no package user account information in the system `passwd` file.

The group file can be much easier to deal with. If it is very seldom that the information in `/etc/group` used outside of the PCUE, then there is no reason to include package group information in the system context `group` file. The use of the system `group` file for package groups would be limited to translating the GID to a group name, providing that `login` is run within the PCUE. Thus, if an installation can live with seeing the GID in the output of directory listings when doing file system work from the system context, then package group information can be excluded from the system `group` file.

Another potential implication of separate `passwd` and `group` files in each package is that, technically, the UID and GID need not be unique between packages. In practice, this is more likely to be useful at with the GID than the UID. If UIDs are non-unique between packages, then, a user in one package could interfere with a different user's process when a) both packages are on the same host and b) both users have the same UID. Remember, although the package file systems are different, the kernel is still common to all packages on a given node. The kernel's protection model in the area of, for example, the kill command is based totally on UID. Thus, in this example, a user could kill processes belonging to another user on a different package. In this same vein, it is not possible to give someone root access to the PCUE and believe that, since their movements are confined to the package, the system is protected from this person. A "package root" user (i.e, an account with UID of 0) can use the kill command on any process, even those on a different package or in the system context.

Note that as a matter of consistency, it would seem prudent to set a standard and stick to it. If you elect to put package user account information in the system context `passwd` file, then also include package group information in the system `group` file.

One final point about using the `passwd` and `group` file from within the PCUE. These files can be maintained (from within the PCUE) using standard HP-UX commands such as `sam(1m)` and `vipw(1m)`. This is made possible, in part, by the fact that the locking mechanism used by these commands is file system-based.

The use of these file system-based facilities gives rise to the question of starting and stopping these facilities in an orderly manner for package startup and shutdown. In the absence of MC/Service Guard, standard HP-UX typically manages the life cycle of these facilities via the `rc` script, starting and stopping them based on the system run-level. The run-level (and `rc`) is controlled by the system `init` process. Of course, there can only be one `init` process on a system. Or can there?

The `init` process essentially processes its configuration file, `/etc/inittab`, and maintains the system state, or run-level, based on the entries in the file. In this perspective, `init` is very much a file system-based facility. However, the `init` process also is responsible for halting the system, rebooting the system, and is guaranteed to have a process ID of 1. These qualities are what guarantee that there is only one `init` process running on a system. The `init` process, as delivered from HP, cannot be run in the package context. Fortunately, HP is not the only source of `init`.

There are various open-source organizations, including the Free Software Foundation and the various Linux vendors. Linux, of course, has an `init` process, and thus the source is readily available. It is fairly straightforward to locate the portions of code that makes `init` undesirable for the package context and modify or remove them. Once modified, Lear renamed the binary to `pkginit`, to make it readily distinguished from the system `init` process. To use `pkginit`, the package must have a context copy of `/etc/inittab`, `/etc/utmp`, and `/var/adm/wtmp`. The later two are used to record the current run-level, as well as run-level transitions. Although not strictly necessary, it is also handy to have context copies of rc and the rc start/stop scripts for cron and lp.

Once `pkginit` is properly installed and configured, the package control script can simply start `pkginit` during package startup, and shutdown pkginit during package shutdown. The `pkginit` process, in turn, will process the package's copy of `/etc/inittab`, calling the appropriate rc start/stop scripts, just as `init` would to start or stop HP-UX. Note that since there are context-separate `inittab` and `pkginit` processes, recognize that each package has a run-level of its own. This run-level is separate from the system run-level and is defined solely by the package administrator. It is not necessary to define the run states as is done by the system context, although doing so is likely to be less confusing and less error prone.

Now the package has its own init process, separate run-levels from the system context, its own scheduler, and its own print spooler. As a side effect of having separate `/etc/utmp` and `/var/adm/wtmp` files, the information reported by `who(1)` will be recorded separately. In fact, the package has its own little Unix world that comes to life as the package starts, and terminates when the package shuts down. The package can be thought of as a symbiont of the system. It is this symbiotic relationship that Lear calls the Package Context Unix Environment.

## The Solution

### *Tools and Technologies*

In this section, some the tools and techniques that are used in the PCUE are presented. These include standard HP-UX commands as well as custom-built programs. For the custom-built programs, documentation of the program's use is provided.

- The `chroot` command/system call. According to the man page for the `chroot` system call, a program that calls chroot will, upon the call's successful return, the directory passed as an argument to the call will "become the root directory". The man page further defines the root directory as "the starting point for path searches for path names beginning with `/`."

  The man page also reminds the reader that the entry `..` in any root directory is interpreted as meaning the root directory itself. Thus, `/..` cannot be used to "escape" the PCUE into file systems that are not to be accessed by non-administrative personnel.

  The only way to "undo" a successful call to `chroot` is to terminate the process that called `chroot`.

  - pkglogin – IP Address to package name translation. The translation is accomplished by running `cmviewcl(1m)` and extracting the package names. Each of the package names is then used to search `/etc/cmcluster`, using the package name as the name of a subdirectory, and looking for a file called `pkg.ctl` contained therein. If located, the string IP_ADDRESS is greped out and then used to compare against the IP address passed to pkglogin. If a match is found, the package name is displayed on the standard output.

  - getcpkg – Get Current Package. This command is the basis for most of the functionality of the PCUE; without it, the PCUE could not operate. The command has a very straightforward goal: display the name of the package that the calling process is "in". That is, if a user telnets to an IP address that belongs to a package named pkg02, then `getcpkg` will produce the string "pkg02" on the standard output when run from that user's shell. How it goes about doing that is much less straightforward.

    There are two distinct scenarios that `getcpkg` is designed to handle. The first is being called from outside the PCUE and the second is being called from inside the PCUE.

    When a process calls getcpkg from outside the PCUE, the concept of the package is meaningless unless the process has been started by way of a reference to an IP address that is owned by the package. The most typical case is when a user telnets to the package[2]. In this case, the `getcpkg` command extracts the user's IP address (where he/she is connecting from) and the user's

[2] See the section below "pkgnet - inetd Wrapper program" for other applications of getcpkg outside of the PCUE.

destination IP address (where he/she is connecting to)[3]. The destination IP address is then compared against the IP addresses of each package. If a match is found, the name of the package is displayed. If not found, `getcpkg` exits silently.

When `getcpkg` is called from within the PCUE, the technique for determining the package is somewhat easier. In this case, `getcpkg` examines the ID number of the root directory. If this is not 0, the process is not under the "real" root. It then looks at all of the directories in the mount table, comparing the ID number of the package's mount directory with the ID number of the root directory that the process is running in. When a match is found, the name of the package is displayed on the standard output.

- execsh – Execute shell. This commands was the initial answer to the question "how do users get in to the PCUE?" When a system receives a `telnet`, `rlogin`, or `remsh` request, the system Internet daemon (`inetd`) is the first point of control. The `inetd` daemon monitors the ports in `/etc/inetd.conf` (numbers defined by `/etc/services`) and starts the associated application when a request is received on the specific port. There is only one `inetd` process per system (even with the PCUE), and thus all `login` and `ftp` requests begin life in the System Context. So an early challenge was to get the user's shell running in the PCUE.

  The initial login shell executes the `execsh` program by design. The Lear environment uses a single home directory in the system context for all users; the profile file in this directory, among other things, determines the package the user has logged into (via `getcpkg`, above) and uses execsh to switch to the appropriate PCUE.

  The execsh program takes one mandatory argument and an optional second argument[4]. The mandatory argument is the name of the package to be switched to; the second (optional) argument is the name of a program to run once inside of the PCUE. The default program to run is a copy of the user's shell, as defined by the environment variable, `SHELL`.

  The `execsh` program does the following:

  1. Changes the current directory to the (real) root directory.

  2. Executes the chroot system call, using the name of the package as the argument. This is one of the things that makes having a directory under root with the same name as the package; if this is not acceptable, `execsh` would need modification.

     Note that the use of the chroot function is restricted to the root user, which implies that `execsh` must be run setuid to root.

  3. Writes an entry in the PCUEs `/etc/utmp` file. This is how running who in the PCUE shows information on users in the package. The PCUEs `/etc/wtmp` file is also updated, allowing package-based audit reporting.

  4. The user's password record is retrieved. Since this is done after the chroot call, this is the user's record from the PCUE `/etc/passwd` file. The current directory is changed to the one specified in the user's password file entry. This allows users to share a single, common home directory in the system context, while having their own, private home directory in the PCUE.

[3] See "Creating Shell Scripts in a High Availability, Multi-site environment" for more details on how this can be accomplished.
[4] The program is currently coded to accept no arguments, and as such will use getcpkg to determine the package to switch to. This use of execsh is not used in practice.

5. The (pseudo) terminal device file being used by the session is changed ownership to the user and access permissions are set to rw--w---- (620 octal). This is necessary, in particular, if the `write(1)` or `wall(1m)` commands will be used.

6. The program `forks` and `execs` a new shell. In the case where a second argument is supplied to `execsh`, this program is run in lieu of the shell.

7. When the shell terminates, `execsh` cleans up the terminal device file, and writes the appropriate `utmp` and `wtmp` records.

- pkginit – package initialization file. This program, taken from the source code of a Linux distribution, was watered down to its simplest form: a `/etc/inittab` file processor. The original program did things like respond to the CTL-ALT-DEL key sequence, responded to power failure notifications, processed the files `/etc/initrunlvl` and `/var/log/initrunlvl`, assumed responsibilities for some console I/O, and initiated single user and "emergency shells". These activities are not appropriate for package-level init processing[5].

  Since all of the code that makes initd "special" has been removed, despite its name, this command (`pkginit`) can be used in non-HA systems as well[6]. Also be aware that since the concept of a single-user state makes no sense outside of the context of a machine boot, the run-levels s and S have been removed. Similarly, no run-level holds any meaning with regard to single or multi user, other than run-level 0, which causes `pkginit` to terminate normally.

- pkgps – Package Process Status. Although, as indicated earlier, the PCUE shares all kernel services with any other package and the non-package-based (system context) software, it can be convenient to filter kernel information on a per-package basis. This is true with the `ps(1)` command. Running the `ps(1)` command produces a list of all process running on the node; the illusion of the-package-is-the-system is much better maintained if the output of the `ps(1)` command contains only the processes that are currently in the same PCUE that the `ps(1)` command is.

  The pkgps command uses the `pstat(2)` system call to obtain information about all running processes. As processes are discovered that has the same root directory as the `ps` command itself, the information is formatted[7] and produced on the standard output.

- pkgbdf – Package Berkley Disk Free. This command is another case of filtering information maintained by the kernel on a per-package basis. In the case of `bdf(1m)`, the default output can be than just confusing. Since, in the PCUE, what users (and application software) think is the root directory is not the what the kernel thinks is the root directory, the information reported by `bdf(1m)` will actually be wrong. The pkgbdf command rectifies this issue by:

  1. Reporting only file systems used by the PCUE in which the `pkgbdf` command is running.

  2. Removing the leading /<package name> from the mount directory information.

[5] At first blush, processing power failure notifications would seem to be a very good thing. Leaving the merits of that point aside, it was decided that having an initd process respond to that type signal was not appropriate. If such processing were to be done, it would be done by a process other than pkginit.
[6] It is not recommended to use pkginit to replace or supplement the HP supplied init program. The pkginit program can be used in non-HA environments if a separate inittab file is used. No other user of pkginit has been tested.
[7] At this time, pkgps takes no options; it produces output in a format similar to the –f option of ps.

- pkgnet – inetd Wrapper programs. The Internet Daemon (`inetd(1m)`) program listens on the ports specified by the `/etc/inetd.conf`, as mentioned earlier. Technically, `inetd` creates a socket. Once the socket is created and communications is established, the socket can be treated (programmatically) as though it was any other open file. Specifically, inetd:

  1. Creates a socket for each port it will wait for connections on. The socket is represented as an open file identified by a file descriptor.

  2. Bind each socket to a specific port.

  3. Listens for a connection on the socket from a (usually) remote system.

  4. inetd then forks. Open file descriptors are preserved across forks, so the socket is valid in both parent and child processes. The parent (original) process will typically close the socket, and if specified by `inetd.conf`, wait for the child to terminate. Because the socket is still open in the child process, closing the socket in the parent does not terminate the connection to the remote system.

  5. The child of inetd will then close all files except the socket. It will then duplicate the socket descriptor to file descriptors 0, 1, and 2 (stdin, stdout, and stderr).

  6. The child process then calls `exec(2)` to the program specified by inetd.conf. Again, open files are preserved across exec calls[8]. So, the new program has stdin, stdout, and stderr "connected to" the socket. Thus any writes to stdout or stderr are ferried to the remote system. Likewise reading from stdin receives data from the remote system.

  Instead of inetd.conf specifying a program like `telnetd(1m)` or `ftpd(1m),` it is perfectly legal to specify another program or shell script, which could then exec the "real" program. This intermediate program or script is referred to as a wrapper program. Obviously, a wrapper that simply execs the "real" program would be of little value. However, the wrapper program could do almost anything (providing it did not disturb stdin, stdout and stderr) prior to, concurrently with or after running the "real" program. In the case of the PCUE, the action taken by pkgnet is to run getcpkg and determine if the remote system is connecting to a package IP address. Once a package is determined, pkgnet simply execs a `chroot(1m)` command using the package name returned by getcpkg for the new root directory and the "real" program as the target. For example, to run telnetd, the pseudo code for pkgnet is shown in Figure 3. In reality pkgnet is a little more generic. Instead of hard coding the program that is to be launched and the arguments to be passed, pkgnet takes this information from the command line. Thus to use pkgnet, an inetd.conf entry for telnetd that was originally

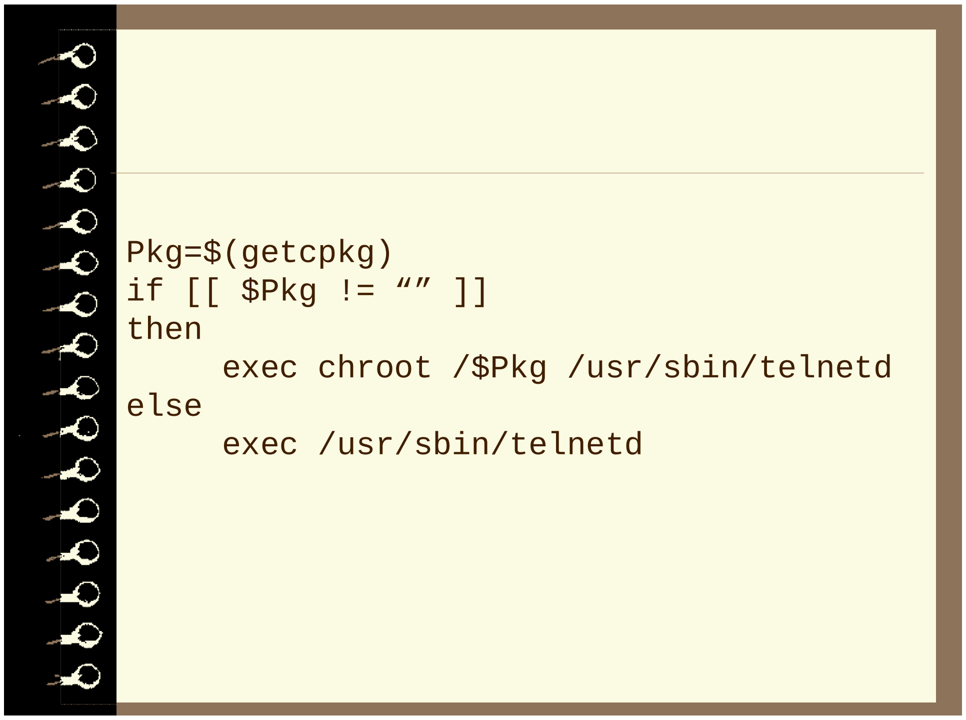
[8] This is the default behavior for open files. It is possible to direct the system to close open files when running `exec(2)`.

```
ftp    stream tcp nowait root /usr/lbin/ftpd   ftpd -l
```

becomes

```
ftp    stream tcp nowait root /corp/bin/pkgnet pkgnet  /usr/lbin/ftpd –l
```

assuming, of course that `pkgnet` is in `/corp/bin`. Note that the string ftpd is not duplicated when used with pkgnet. This is a function of how `inetd(1m)` uses the information in `/etc/inetd.conf` to exec programs.



```
Pkg=$(getcpkg)
if [[ $Pkg != “” ]]
then
     exec chroot /$Pkg /usr/sbin/telnetd
else
     exec /usr/sbin/telnetd
```

**Figure 3. pkgnet pseudo code**

Using this style of coding, pkgnet can be used with virtually any inetd.conf entry[9]. By applying pkgnet (or equivalent) to all connection entries, any method a user employs to connect to the package will result in a switch to the PCUE. Aside from a very brief time while inetd runs pkgnet and pkgnet subsequently runs the indicated program, *no processing occurs outside of the PCUE on the behalf of a user’s connection request or the resulting session*. The implications of this are profound.

- The switch to the PCUE takes place prior to any attempts to authenticate the user, rendering it unnecessary to have any mention of the user in the system password and/or group files whatsoever. However, see the precautions mentioned in “Theory” section, under the subsection “The Brave New World”.

- As a corollary to the above point, it is possible (if not likely) to have user accounts in the System Context (e.g., administrators) that are not in the Package Context. By not making the information for these accounts visible to the Package Context, hacking becomes more difficult.

[9] Although the technique is technically sound, of the entire “standard” HP-UX inetd.conf entries, only `telnetd(1m)`, `ftpd(1m)` and `remshd(1m)` have been tested.

- Everything (in terms of System Context file system objects) that is not explicitly placed in the PCUE is not accessible to users in the PCUE.

### *Prerequisites*

1. All package file systems must mount under a single mount point. At Lear, this mount point is under the root directory and the mount point name is the same as the package name. However, neither of these are requirements.

2. Applications must either be loaded on the package file systems or loaded on a file system that can be mounted to the package file systems. Once the `chroot` call has been made, everything above the new root instantly becomes unavailable.

3. There must be a separate directory under `/etc/cmcluster` for each package, each bearing the name of the package. In this directory, there must be a file named pkg.ctl that has the IP_ADDRESS array defining the IP addresses owned by the package. This can be the same as the package control script, but that is not strictly necessary. This prerequisite allows `pkglogin` to perform IP address to package name translation; if it is not reasonable to comply with this requirement, then pkglogin must be modified.

## Conclusions

The Package Context Unix Environment provides a secure way to separate packages from each other and from the system. In addition, file system based services such as the print spooler and scheduler can be run in the package context, providing the basis for highly available printing and scheduling. By implementing an inetd(1m) wrapper program, network facilities such as ftp(1), telnet(1), and rlogin(1) can be used to access the PCUE with virtually no system context footprint. Administration of cluster that utilizes the PCUE is greatly simplified, as each package can be treated as a separate machine. The impact of patching systems with a PCUE can be minimized with judicious use of local file systems and symbolic links.

## Manual Pages

## NAME

getcpkg – get current package

## SYNOPSIS

```
getcpkg [-i] [-d] [-f]
```

## DESCRIPTION

Display the current package on standard output or nothing if no current package.

### Options.

**getcpkg** recognizes the following options:

**-i** Display internet addresses (only). No attempt is made to resolve a package name from the source IP address.

**-d** Turn on debugging. No diagnostic messages are produced unless debugging is on.

**-f** Assume parent is daemon spawned by inetd(1m)

## EXTERNAL INFLUENCES

None

## RETURN VALUE

Upon successful completion, **getcpkg** returns 0. Otherwise, the error codes currently returned are:

1 – unable to open pipe to sub-process

4 – inetd process not found.

5 – inetd process not found.

6 – unable to open `/etc/mnttab`

7 – unable to find the root directory in `/etc/mnttab`

## DIAGNOSTICS

Diagnostic messages are sent to syslogd(1m) with the LOG_AUTH facility using a *ident* string of **getcpkg**.

"call to PAMGetDaemon failed"
Produced if **getcpkg** is unable to identify a process that is a child of inetd(1m) and a parent of **getcpkg**.

"setmntent failed"
Occurs when PAMGetDaemon has detected the possibility that **getcpkg** is running in the PCUE but is unable to successfully call setmntent(3), indicating a failure to access the PCUE file `/etc/mnttab`.

"statvfsdev failed for <file system>, <error message>"
Occurs when **getcpkg** cannot get information about a file system found in the PCUE file `/etc/mnttab`. This is a non-fatal error; the file system is skipped.

"Checking <file system>"
Produced if **getcpkg** is unable to identify a process that is a child of inetd(1m) and a parent of **getcpkg**.

"no mntent found"
Produced if **getcpkg** is unable to successfully call getmntent(3)**.**

"running <command>"
Produced if **getcpkg** when runs lsof(8). <command> is the string passed to popen(3).

"call to PAMGetDaemon had an unknown failure <d>"
Produced if PAMGetDaemon returns an error that **getcpkg** is unable handle. The exact integer returned is <d>.

"popen for lsof failed (<error message>)
Produced if **getcpkg** is unable to run the lsof(8) command.

"get <string> from lsof"
The <string> is the output of the lsof(8) command.

"popen for echo/cut/cut failed (<error message>"
Produced if **getcpkg** is unable to run the indicated pipeline. <error message> is the errno string.

"IP address of remote system: <IP Address>"
Produced when **getcpkg** has deduced the (source) IP address of the remote system. <IP Address> is the result of the deduction.

"IP address of package: <IP Address>"
Produced when **getcpkg** has deduced the (target) IP address of the package. <IP Address> is the result of the deduction.

"popen for pkglogin failed (<error message>)
Produced if **getcpkg** is unable to run the pkglogin(8) command. <error message> is the errno string.

**EXAMPLES**

**WARNINGS**

The current version of **getcpkg** depends on the third party utility lsof(8). This command must be installed and fully operational (i.e., return all information about sockets in use) for **getcpkg** to operate correctly.

**AUTHOR**

George Morrison developed **getcpkg**.

**SEE ALSO**

lsof(8), pkglogin(8), inetd(1m)

## NAME

pkginit, telinit – package initialization

## SYNOPSIS

```
pkginit
telinit [ 0|1|2|3|4|5|6 ]
```

## DESCRIPTION

**pkginit** This command is intended to be the father of all PCUE process. Its primary role is to create processes from a descriptions in the file `/etc/inittab` (see inittab(4)). Only one run-level has any special meaning to **pkginit**. Entering run-level 0 causes **pkginit** to terminate. Any other uses of run-levels are defined by entries in the inittab(4) file. After it has spawned all of the processes specified, **pkginit** waits for one of its children to die or until it is signaled by telinit to change the package's run-level. The **pkginit** process listens for commands from **telinit** on the FIFO `/tmp/initctl`.

**telinit** The **telinit** command is used to send information to **pkginit**, namely requests to change run-levels. It does this by sending the requested run-level to pkginit via the FIFO `/tmp/initctl`. **telinit** is linked to **pkginit**.

## AUTHOR

George Morrison modified **pkginit**. The code originates from the sysvinit-2.74 package.

## FILES

PCUE:

```
   /etc/inittab
   /tmp/initctl
```

## SEE ALSO

init(1m)

## NAME

pkgps – package process status

## SYNOPSIS

`pkgps`

## DESCRIPTION

Display the processes running in the current package. Processes that are running in the system context or in other package contexts are ignored.

## EXTERNAL INFLUENCES

None

## RETURN VALUE

Upon successful completion, **pkgps** returns 0. Otherwise, the error codes currently returned are the same as those returned by the ps(1) command.

## WARNINGS

The current version of **pkgps** depends on getcpkg(8).

pkgps only displays process status in the same format as ps(1) with option –f. No other display formats are implemented.

pkgps selects processes in a fashion similar to ps(1) with the –e option. No other process selection is implemented.

## AUTHOR

George Morrison developed **pkgps**.

## SEE ALSO

ps(1), getcpkg(8)

## STANDARDS CONFORMANCE

No standards have been set

## NAME

pkglogin – IP to package name translation

## SYNOPSIS

```
pkglogin xxx.xxx.xxx.xxx
```

## DESCRIPTION

Display the current package on standard output or nothing if no current package.

## EXTERNAL INFLUENCES

None

## RETURN VALUE

Upon successful completion, **pkglogin** returns 0.

## WARNINGS

The current version of **pkglogin** depends on the ability to run cmviewcl(1m). It will check to see if this command is executable from `/usr/sbin` and exit if this is not the case.

**pkglogin** assumes the package control script is named `pkg.ctl`.

## AUTHOR

George Morrison developed **pkglogin**.

## FILES

```
/etc/cmcluster/<pkgname>/pkg.ctl
```

**NAME**

pkgbdf – package disk free

**SYNOPSIS**

```
pkgbdf [–p pkg] [-b] [-i] [-l] [-t fstype | [filesystem|file]]
```

**DESCRIPTION**

Display the disks mounted under the current package on standard output. File systems that are mounted only to the system context or in another package context are ignored.

**Options.**

**pkgbdf** recognizes the following options:

**-p pkg** Display disk free information for the indicated package. If this option is not present, pkgbdf degenerates to the standard bdf utility.

**-b** Turn on debugging. No diagnostic messages are produced unless debugging is on.

**-i** Sam as –i for bdf command

**-l** Process local file systems only

**-t fstype** Process file systems of *fstype* only.

filesystem|file
Display information for the indicated filesystem or the file system on which the indicated file(s) reside only.

**EXTERNAL INFLUENCES**

None

**RETURN VALUE**

Upon successful completion, pkgbdf return 0. Otherwise, the error codes currently returned are the same as returned by bdf(1m).

**AUTHOR**

George Morrison developed **pkgbdf**.

**FILES**

PCUE:

```
/etc/mnttab
```

**SEE ALSO**

bdf(1m)

**NAME**

pkgnet – package inetd wrapper

**SYNOPSIS**

```
pkgnet pkgnet path [args]
```

**DESCRIPTION**

**pkgnet** is used as a wrapper for daemons spawned by inetd. It is designed to be used in the system `/etc/inetd.conf` to start the daemons in the appropriate package context, or, failing to determine a package context, in the system context. The overall operation is as follows. When inetd detects a request to start a daemon on one of the ports that it is watching, it spawns **pkgnet**, passing the name of the intended daemon as an argument.

**pkgnet**, using getcpkg(8), determines if the client system has sent the request to an IP addressed of a package on the node. If not, pkgnet executes the daemon program.

If **pkgnet** determines that the request was sent to on IP address of a package on the system, then pkgnet does a chroot(1m) call to the package's root directory. If the chroot call is successful, the daemon is executed. Thus all files and directories referenced by the daemon are in the package context.

**EXTERNAL INFLUENCES**

None

**RETURN VALUE**

Upon successful completion, **pkgnet** returns whatever the daemon (passed to **pkgnet** as an argument) returns.

**EXAMPLES**

```
telnet stream tcp nowait root /corp/bin/pkgnet pkgnet /usr/lbin/telnetd
```

**AUTHOR**

George Morrison developed **pkgnet**.

**FILES**

System:

```
/etc/inetd.conf
```

**SEE ALSO**

getcpkg(8)