rename(“open”, “swinging_to_and_fro”);

David Tilbrook (dt@snitor.uucp)

Sietec Open Systems Division

ABSTRACT

The push for open systems is on. The objectives of open systems are to achieve independence of supplier and/or manufacture, portability of software, and environmental differences that are transparent to the user. Part of this push must involve the evolution of a standard software foundation, and in some ways it appears that Unix is evolving as that foundation.

However, the successful achievement of the objectives of the open systems push depends largely on the reliability and consistency of that software foundation.

This paper looks at one subroutine *rename*(2), that has been deemed to be part of the Posix standard. This paper relates the author’s experience with this subroutine across some dozen different platforms and configurations, and draws some conclusions regarding the chances of fulfilling the open systems marketing hype.

1. Introduction

The *rename* function was not part of the early versions of Unix. Its basic functionality was relatively easy to implement, as discussed later, using the *link*(2) and *unlink*(2) system calls. One could live without a kernel implementation of *rename*, and, indeed, on some systems, one still does so. However, it was implemented as part of the 4.2bsd fast file system extensions and has been adopted as part of the Posix standard, so examining this subroutine to attempt to prognosticate the future of the open systems effort is not gratuitous.

The following is an extract of the UPM manual section for *rename*. Other variations exist; however, barring the copious caveats concerning circular directory graphs, and the various error conditions, the basic essence of rename is as described.

NAME

rename – change the name of a file

SYNOPSIS

```
rename(from, to)
char *from, *to;
```

DESCRIPTION

Rename causes the link named *from* to be renamed as *to*. If *to* exists, then it is first removed. Both *from* and *to* must be of the same type (i.e., both directories or both non-directories), and must reside on the same file system.

Rename guarantees that an instance of *to* will always exist, even if the system should crash in the middle of the operation.

RETURN VALUE

A 0 value is returned if the operation succeeds, otherwise *rename* returns −1 and the global variable *errno* indicates the reason for the failure.

Now, how could anyone get that wrong? Actually getting it right is difficult (see *Epilogue #1*). There are all sorts of potential errors, some of which are covered in the parts of the manual section that I did not include. Certain aspects, such as renaming directories, should be approached with great apprehension. In fact I do. *rename* is not provided on a variety of systems on which I have to run my software. On those systems I have to provide an alternative mechanism to support the basic functionality of changing the name of a file. Furthermore, there are many situations where using *rename* arbitrarily can give rise to errors from which one then has to recover (i.e., cross device links or cross directory renames not supported on some file systems). Therefore, I limit my use of *rename* to files within the current directory. An examination of a variety of rename manual pages seems to indicate this subset of *rename*'s supposed capabilities avoids many of the potential errors. Furthermore, by limiting myself to that subset, the *rename* functionality can be almost duplicated using the following code:

```
#include <errno.h>

int
rename(from_f, to_f)
    char * from_f;
    char * to_f;
{
    if (unlink(to_f) == -1 && errno != ENOENT)
        return (-1);
    /* if we crash here, to_f is gone - sigh
     * if from_f cannot be linked we lose as well */
    if (link(from_f, to_f) == -1)
        return (-1);
    return (unlink(from_f));
}
```

The reader will appreciate that this is obviously less efficient than the true *rename* in that it uses three system calls vs. one, and has the added disadvantage that *to_f* is unlinked before we are sure that *from_f* exists and is the same type as *to_f*. However, more importantly, the guarantee that *to_f* always exists, even in the event of a system crash, cannot be maintained, as noted in the comment. But that's a risk we have to take, because on systems that don't offer *rename*, we are forced to do so.

For a long time, even though more and more of the systems on which I was running my code supplied *rename*, I used the above routine (under a different name) to implement the limited semantics. I still avoid using *rename* for any other uses, and I think that that caution is merited, as will be defended in the rest of this document.

Despite this aversion to using *rename* to move files or directories, I do now use it to rename files within a directory. So what's the big deal?

Section 3 describes seven different situations that I have encountered in using *rename*, even when I limit myself to the renaming of a file within the current directory. So if you are thinking that the open systems push is going to alleviate your porting problems, read on. The problems I have discovered with a routine whose semantics can be almost duplicated in five lines of fifth edition code, might cause you to think again. But before launching into these problems, let me discuss installing a file, something one wants to do occasionally.

2. How to Install a file

Inherent in the process of building a system is the problem of copying the new version of the file into its destination directory. If you just say, "Use *install*(1)", you have not considered the problem, nor have you looked at the some 20 different implementations of that command. Furthermore, you are probably dealing with very small software system, because the customary *sh* implementation is excessively slow when dealing with thousands of files. But the efficiency is a minor consideration — the inconsistency of the interface is not, but we will ignore that problem for now.

The major failing of the standard shell implementation of *install* is that it is woefully naive and far too trusting!

When one is installing a production file, one **must** take all the measures one can to ensure that the installation fully succeeds or fails noisily! Furthermore, one **must** ensure that one **never** leaves the system in a state in which neither the new copy nor the old copy is available and usable! For those of you who have never experienced it, let me assure you that trying to cope on a system that failed to correctly install a new copy of */bin/sh*, but not before it removed the old version, is not a pleasant nor relaxing situation.

It is to provide this guarantee that *rename* should be used.

But it is not that simple. Often one has to manipulate the file once it has been installed (e.g., apply *strip* or *ranlib*). Without further ado, I use a program called **instal** which is now briefly explained, as is very good at manifesting and detecting problems with *rename*.

Due to its complexity and importance, *instal* provides a **−X** flag, in addition to the ubiquitous **−x** flag. Its output is presented below and rationalized.

```
% instal -X
instal [flags] targetdir/file newcopy will do the following:
      if targetdir does not exist
            mkdir targetdir [1]
      if targetdir/@nfile exists
            unlink targetdir/@nfile [2]
      copy newcopy targetdir/@nfile [1,3]
      if size(newcopy) != size(targetdir/@nfile)
            abort # check if yet another F.S. failure
      chdir targetdir [2]
      if -r flag ranlib @nfile
      if -s flag strip @nfile
      if -o or -g flags chown/chgrp @nfile [4]
      if -M flag chmod @nfile [4]
      if file exists {
            if @?file exists [5]
                  unlink @?file
            link file @?file [1]
      }
      # NOTE: @nfile exists and file does not or is linked to @?file
      rename @nfile file [1,2]
      if @?file exists and ! -k
            unlink @?file [6]
[1] Check if successful using stat(2) - abort if not
[2] abort if returns -1
[3] this is a built-in copy - cp(1) not used to save time
    and to facilitate checking the result of every write
[4] aborts if -1 unless -I flag specified
[5] To deal with problem on systems that prohibit removing a ETXTBSY
    file and other situations when a file might not be removable,
    instal iterates over '@[0-9]file' until one is found that
    does not exist or can be successfully unlinked.
[6] Failure ignored if due to busy text file,
    otherwise aborts.
```

Note the following:

This happened to us once at the University of Toronto in 1975. It was exciting and yet worrying, and the group that was involved in the recovery involved a Duff, a Reeves, a Tilson, a Tilbrook and others of similar stature.

Note spelling change to ensure that the correct version is used, no matter setting of ${PATH}.

- *instal* copies the new file into the target directory, but not to the ultimate name. The copy is done to a temporary name, thereby protecting the old version until the new copy has been fully installed in the directory and any subsequent processing has been completed.
- The success of the copy is checked by both examining the result of every *write*(2), the *close*(2) and then (since I am a paranoid) by checking that the size (as retrieved using *stat*(2)) has not changed.
- We do not rename the new file to the new location before the old version has been safely linked to a new name. This process is complicated by the possible need to try multiple names before the link can be successfully performed.
- Unlinking the old version is immediately followed by linking in the new version, with absolutely no intervening processing! These operations are combined using *rename* if possible. This is essential since once the unlink of the old file has succeeded, the program does not exist under its usual name — think again about a crashed system on which */bin/sh* does not exist.

Okay ... let's get back to *rename* which is supposedly the subject of this paper. The desirability of using *rename* in the above processing should be obvious. Its guarantee of preserving the *to* file is necessary. But before moving on, consider the following two points:

1) *instal* is used to install *instal* itself and other programs that might be ‘‘busy’’. This is why care is taken to link the current version of the target file to the temporary *@?file* so that the ultimate *rename* that unlinks the original and links in the new file does not destroy the last link to a busy text file.
2) Some versions of *make*, including our variant *mimk*, run processes in parallel. Consequently the situation in which multiple *instal*s are manipulating links within the same directory arises often. However, the same file is never involved in parallel renames.

3. Six different permutations of rename, plus the degenerate case

I believe that I have now demonstrated the important of having a *rename* function. The following sub-sections describe experiences I have had with *rename* via the *instal* program. With the exception of the sixth case, each of the cases exist on one or more of the systems attached to our network. The reader should also keep in mind that I am limiting my use of *rename* to regular files within the current directory, thereby eliminating problems that can arise due to busy directories, cross device or directory links, circular paths, etc.

3.1. Behold — a working rename

There are systems that actually supply a *rename* function that seems to work — well, let's say that I have not as yet detected a failure due to a software bug. I mention it to demonstrate, through an existence proof, that it is possible.

3.2. rename undefined

As mentioned, there are relatively new systems that still do not provide *rename*. However, of the seven situations listed in this paper, this is the second easiest with which to cope. If one cannot have a working version, one is almost better off with no version at all. One is faced with the problem of where to put the *unlink/link/unlink* implementation and how to adapt imported software that expects it to be in place, but this is a situation with which I am well familiar.

3.3. rename defined — as a no-op

There is a version of *rename*, distributed by one organization that does nothing — absolutely nothing! It does not rename the files, it does not return −1 — whoops I tell a lie — it does something — it returns 0, which is not exactly nothing, but very close to it.

Yes Viceginia, there are file systems on which copying a file can change its size (even ignoring sparse files such as those created by *dbm*).

Machines on our network currently include targon35s (pyramid) using both att and ucb universes, apollo 3500s (bsd side), mips (bsd side), mx300s (att side), wx200s, m88ks, an old sun3, and various 386s. All constructions are done using source accessed via NFS to one of the targons. With the exception of the sun, all constructions are done on a local disk. All installs are to a local disk.

This situation is actually not the most unpleasant, as the problem is discovered the very first time one tries to use *rename*(2). In this situation one immediately treats the offending system as if *rename* was undefined.

3.4. Almost works — just one minor (detectable) glitch

Remember how I stated how *instal* was to install itself? There is yet another variation on today's theme in that there is a *rename* that works everywhere, except when trying to rename a busy text file (e.g., *instal*). We have not seen any mention of this situation being illegal. In fact on the same system, the *unlink/link/unlink* simulation works as the *link* increments the link count thus the unlink does not reduce it to zero. Some implementations of *unlink* legitimately object if an attempt to remove the last link to a busy file is attempted, but *rename* should not raise this condition when the busy file is the *from* file. So, on such systems *instal* checks for `errno==ETXTBSY` after a failed *rename* call. If the test is positive, the *unlink/link/unlink* simulation is used.

Discovering if the system's *rename* exhibits this errant behaviour is simply a matter of compiling and executing the following program:

```
main()
{
        if (rename("a.out", "new.out") == -1) {
                perror("a.out");
                exit (-1);
        }
        puts("might have worked... please check new.out exists");
}
```

3.5. Almost works — just one minor (undetectable) glitch

Some un-named manufacturer has distributed in the past a version of *rename*() that frequently fails to work when trying to rename a file on a remote file system. Unfortunately, trying to detect that the *rename* failure is due to a file system time out is difficult to do in a consistent and portable way. Furthermore, it goes against my paranoic nature to try to second guess hard errors.

The only course of action is to restart the building process, which almost inevitably completes successfully the second time. This success-on-second-try phenomenon may be due to the fact that the second run is usually done after the constructions on all the other parallel construction streams have completed, thereby with a much reduced network load.

It is not definite that this occasional failure is due to a *rename* bug, but because its second time success rate, and the rarity of occurrence on a relatively unimportant system, it has not merited much investigation. It's mildly irritating, but not as much as the next two situations.

3.6. Panic — it's multiple renames

Surprisingly, the version of *rename* that frequently caused a kernel panic is not the most unpleasant, but it's close. This implementation of *rename* seemed to cause the system to get its knickers in a twist when multiple renames were being run in parallel. The result was a kernel panic and a mildly corrupted file system, usually in the directory in which the renames were attempting to function when so rudely interrupted.

The solution in this case was initially to eliminate the parallel runs, and ultimately to change jobs.

3.7. Panicking would be preferable.

Finally the ultimate *rename* disaster. The most recent new port of our software gave rise to a situation on Unix with which I was previously unfamiliar. Way back when — circa 1977 — I saw a directory created in which the normal ordering of **..** and **.** was reversed. It was amusing, yet not unhealthy. However, until I

Our major system builds are done as three parallel job streams, each stream processing three or four of the platforms or configurations. All the constructions use the same remote source file system, via our LAN; consequently the network is usually fairly busy servicing three job streams, each of which is running multiple parallel compiles.

encountered the rename bug described in this section, I have never consciously seen a directory with duplicate entries, that is two entries in the same directory, with the same name, referring to the same inode. Again, this is not as unhealthy as it might seem — provided the inode's link count includes both entries. Unix is fairly adapt at handling multiple references (i.e., links) to the same file. The name, for the most part, is irrelevant, once the directory search has retrieved the inode number. You would find that, if was possible to create two entries in a directory with the same name, the file system primitives and tools (notably *rm*) would all work properly. This is, for the most part, true, even when the duplicated names had different inodes, although specifying or predicting which inode was used could be difficult.

Unfortunately, if the inode link count does not include the duplicate entry, which is the situation produced by this rename bug, the file system is mildly corrupted, but soon to be dramatically corrupted as soon as one of the duplicate entries is unlinked. This will reduce the inode link count to zero, thereby returning the still-in-use inode and disk-blocks to their respective freelists.

The first time that this situation arose, the full system construction had completely, supposedly successfully, at which point the file system integrity checking package aborted on a sequence error. Thus, detecting this problem proved to be yet another defence of one of Dr. Tuna's Software Hygiene Nostrums [Tilbrook 90]:

- Create and frequently use a file system integrity package that checks for spurious files, missing files, obsolete files, and so on.

It was the "and so on" check that revealed the problem. The *comm*(1)-like process that compares the installed file list against the master list aborts on sequence errors and duplicated lines — something which *comm* does not do, hence its replacement. Naturally, I assumed initially that the sequence error was in the manually maintained master list. However, upon rerunning the package the same sequence error was reported indicating a sequence error or duplicate line in the following command's output:

```
find . -print | sort
```

My experience with *sort* is that it works almost without fail, so I ran:

```
find . -print | sort | uniq -d
```

There were dozens of duplicate lines, a situation I had never experienced in my previous 16 years of Unix use. Furthermore, I discovered dozens of files pointing to the wrong contents, probably due to an in-use-inode being reallocated to a new file. Recovery was difficult and time-consuming — I had to remove the entire production tree and start all over again. It did have one positive effect. I was forced to create yet another regression test that actually checked that, not only did the required programs exist in the right directory, with the right modes, but they also contained the right contents.

Further experimentation with the strengthened installation regression testing suite definitively fingered yet another problem with *rename*.

This problem was the most difficult to detect. It was also the most insidious in that it caused the most damage and required the most effort to repair and rectify. The solution was to eliminate parallel *instal*s by reducing the maximum number of parallel processes executed by *mimk* (the D-Tree *make* replacement) to one. Since doing so, the file system has remained relatively healthy.

4. Conclusion

As stated in the abstract, the successful achievement of the objectives of the open systems push depends largely on the reliability and consistency of the software foundation.

My use of an extremely simple and basic, yet essential, function, that has been deemed to be part of that foundation, has yielded a dismayingly wide range of compliance and quality. This leads one immediately to the conclusion that the suppliers' approaches to system validation is less than rigourous.

Note that, in the event of a system crash during a rename function, the link count for the *from* file might actually be one higher than it should be. This is because the *from* link count is incremented before the *to* entry's is set to that inode. But, this is a situation that may be harmlessly rectified by *fsck* when the system is rebooted.

Mind you, its performance is pretty abysmal on some of our 386 systems.

The test was to run all the programs with the **–x** flag, which generates the program's synopsis and description, and compare the collected outputs against a master list.

rename is a function that can be tested relatively easily. Checking that a call to it works is easily done using any number of standard tools (e.g., *ls*) or mechanisms (e.g., *stat*). Actually, *rename* is so fundamental that it is rather surprising that its failure is not detected fairly quickly by the supplier through normal system use — assuming suppliers actually use their product before they ship it. I detected the problems in the five buggy versions through normal use.

If suppliers have problems creating and testing a function as simple as *rename*, am I wrong in thinking that they might have similar problems when it comes to more complicated routines with much broader domains, such as semaphores, sockets, signals, process control, programming languages, or window managers?

Am I unjustified in being publicly critical of some of the suppliers who exhibit what can only be interpreted as gross negligence in the interest of marketing or politics?

I do not have a lot of confidence in the current batch of suppliers. The cited problems with *rename* form a very small, but highly representative, subset of the problems I have discovered in trying to maintain our products on 9 different platforms simultaneously.

What can one do to improve this gloomy forecast?

Hit them where it hurts — in their commission belts. When some salesman starts promoting his/her product as a wonderful, leading edge open systems approach solution to your problems, say:

> ‘‘Oh yeah? Tell me how you test your *rename*(2).
>
> Then we'll get on to things that aren't trivial!’’

Bibliography

Tilbrook 90b David Tilbrook and John McMullen, *Washing Behind Your Ears − or − The Principles of Software Hygiene.*, Keynote address, EurOpen Fall Conference, Nice, 1990.

5. Epilogue #1

To verify the origin of the first *rename* I called Kirk McKusick. In our conversation he told me that when he presented the design of the new directory format and the kernel implementations of the *mkdir*, *rmdir*, and *rename* functions to the 4.2bsd review committee, Dennis Ritchie questioned the wisdom of allowing one to rename directories. Dennis thought that it would be very difficult to get right. Kirk then went on to state that Dennis's insight was correct: implementing *rename* was one of the most difficult coding tasks he has ever done. Furthermore he related some problems that I have not discussed, but occur even when we limit the analysis to renaming files within the current directory. For example, consider the problems of executing any two of the following calls in parallel:

```
rename("alpha", "beta");
rename("alpha", "gamma");
rename("gamma", "alpha");
rename("beta", "alpha");
```

while ensuring the guarantee that the ‘‘to’’ file existed, even in the event of a system crash.

Granted ... getting it right is difficult, but not impossible.

6. Epilogue 2

Initially this paper was written quickly, without any intended destination or audience. It was just a case of: ‘‘this mess should be documented’’. In the absence of a formal outlet, I mailed a copy to another renowned curmudgeon, Barry Shein, of Software Tool & Die.

The next day, I received a reply, which contained in part:

‘‘That rename() grump is a good one, a good sermon. It's worse than you state, at another level. The world is now going nuts.

Examining the variations in the interpretation of the C language's semantics would be another, much longer, paper. Discussing coping with different releases of a window system is a thesis.

Under BSD and all versions of Sun/OS previous to 4.1 that I know of, the SYNOPSIS for the manual section for rename (and many similar ‘‘from−>to’’ kind of calls) looked like:

```
rename(from,to)
char *from, *to;
```

as you mention. Now, in some sort of rabid attempt to be SVR4/POSIX/1003 or something compatible, someone decided to change all those SYNOPSIS to:

```
rename(path1,path2)
char *path1, *path2;
```

Now, isn’t that precious? What might have motivated a company to call forth their legions of tech writers to change these manual pages in this way?’’

Given that new synopsis, how long do you think it will be before I can add an eighth case (the one in which the arguments are reversed) to my list?