Changes to the Programming Environment in the Fourth Release of Plan 9

Rob Pike

rob@plan9.bell-labs.com

Introduction

The fourth release of Plan 9 includes changes at many levels of the system, with repercussions in the libraries and program interfaces. This document summarizes the changes and describes how existing programs must be modified to run in the new release. It is not exhaustive, of course; for further detail about any of the topics refer to the manual pages, as always.

Programmers new to Plan 9 may find valuable tidbits here, but the real audience for this paper is those with a need to update applications and servers written in C for earlier releases of the Plan 9 operating system.

9P, NAMELEN, and strings

The underlying file service protocol for Plan 9, 9P, retains its basic form but has had a number of adjustments to deal with longer file names and error strings, new authentication mechanisms, and to make it more efficient at evaluating file names. The change to file names affects a number of system interfaces; because file name elements are no longer of fixed size, they can no longer be stored as arrays.

9P used to be a fixed-format protocol with NAMELEN-sized byte arrays representing file name elements. Now, it is a variable-format protocol, as described in *intro*(5), in which strings are represented by a count followed by that many bytes. Thus, the string `ken` would previously have occupied 28 (NAMELEN) bytes in the message; now it occupies 5: a two-byte count followed by the three bytes of `ken` and no terminal zero. (And of course, a name could now be much longer.) A similar format change has been made to `stat` buffers: they are no longer `DIRLEN` bytes long but instead have variable size prefixed by a two-byte count. And in fact the entire 9P message syntax has changed: every message now begins with a message length field that makes it trivial to break the string into messages without parsing them, so `aux/fcall` is gone. A new library entry point, `read9pmsg`, makes it easy for user-level servers to break the client data stream into 9P messages. All servers should switch from using `read` (or the now gone `getS`) to using `read9pmsg`.

This change to 9P affects the way strings are handled by the kernel and throughout the system. The consequences are primarily that fixed-size arrays have been replaced by pointers and counts in a variety of system interfaces. Most programs will need at least some adjustment to the new style. In summary: NAMELEN is gone, except as a vestige in the authentication libraries, where it has been rechristened ANAMELEN. `DIRLEN` and `ERRLEN` are also gone. All programs that mention these constants will need to be fixed.

The simplest place to see this change is in the `errstr` system call, which no longer assumes a buffer of length ERRLEN but now requires a byte-count argument:

```
char buf[...];

errstr(buf, sizeof buf);
```

The buffer can be any size you like. For convenience, the kernel stores error strings internally as 256-byte arrays, so if you like — but it's not required — you can use the defined constant ERRMAX=256 as a good buffer size. Unlike the old ERRLEN (which had value 64), ERRMAX is advisory, not mandatory, and is not part of the 9P specification.

With names, stat buffers, and directories, there isn't even an echo of a fixed-size array any more.

Directories and wait messages

With strings now variable-length, a number of system calls needed to change: `errstr`, `stat`, `fstat`, `wstat`, `fwstat`, and `wait` are all affected, as is `read` when applied to directories.

As far as directories are concerned, most programs don't use the system calls directly anyway, since they operate on the machine-independent form, but instead call the machine-dependent `Dir` routines `dirstat`, `dirread`, etc. These used to fill user-provided fixed-size buffers; now they return objects allocated by `malloc` (which must therefore be freed after use). To 'stat' a file:

```
Dir *d;

d = dirstat(filename);
if(d == nil){
        fprint(2, "can't stat %s: %r\n", filename);
        exits("stat");
}
use(d);
free(d);
```

A common new bug is to forget to free a `Dir` returned by `dirstat`.

`Dirfstat` and `Dirfwstat` work pretty much as before, but changes to 9P make it possible to exercise finer-grained control on what fields of the `Dir` are to be changed; see *stat*(2) and *stat*(5) for details.

Reading a directory works in a similar way to `dirstat`, with `dirread` allocating and filling in an array of `Dir` structures. The return value is the number of elements of the array. The arguments to `dirread` now include a pointer to a `Dir*` to be filled in with the address of the allocated array:

```
Dir *d;
int i, n;

while((n = dirread(fd, &d)) > 0){
        for(i=0; i<n; i++)
                use(&d[i]);
        free(d);
}
```

A new library function, `dirreadall`, has the same form as `dirread` but returns the entire directory in one call:

```
n = dirreadall(fd, &d)
for(i=0; i<n; i++)
        use(&d[i]);
free(d);
```

If your program insists on using the underlying `stat` system call or its relatives, or wants to operate directly on the machine-independent format returned by `stat` or `read`, it will need to be modified. Such programs are rare enough that we'll not discuss them here beyond referring to the man page *stat*(2) for details. Be aware, though, that it used to be possible to regard the buffer returned by `stat` as a byte array that began with the zero-terminated name of the file; this is no longer true. With very rare exceptions, programs that call `stat` would be better recast to use the `dir` routines or, if their goal is just to test the existence of a file, `access`.

Similar changes have affected the `wait` system call. In fact, `wait` is no longer a system call but a library routine that calls the new `await` system call and returns a newly allocated machine-dependent `Waitmsg` structure:

```
Waitmsg *w;

w = wait();
if(w == nil)
        error("wait: %r");
print("pid is %d; exit string %s\n", w->pid, w->msg);
free(w);
```

The exit string `w->msg` may be empty but it will never be a nil pointer. Again, don't forget to free the structure returned by `wait`. If all you need is the pid, you can call `waitpid`, which reports just the pid and doesn't return an allocated structure:

```
int pid;

pid = waitpid();
if(pid < 0)
        error("wait: %r");
print("pid is %d\n", pid);
```

Quoted strings and tokenize

`Wait` gives us a good opportunity to describe how the system copes with all this free-format data. Consider the text returned by the `await` system call, which includes a set of integers (pids and times) and a string (the exit status). This information is formatted free-form; here is the statement in the kernel that generates the message:

```
n = snprint(a, n, "%d %lud %lud %lud %q",
        wq->w.pid,
        wq->w.time[TUser], wq->w.time[TSys], wq->w.time[TReal],
        wq->w.msg);
```

Note the use of `%q` to produce a quoted-string representation of the exit status. The `%q` format is like %s but will wrap `rc`-style single quotes around the string if it contains white space or is otherwise ambiguous. The library routine `tokenize` can be used to parse data formatted this way: it splits white-space-separated fields but understands the `%q` quoting conventions. Here is how the `wait` library routine builds its `Waitmsg` from the data returned by `await`:

```
Waitmsg*
wait(void)
{
        int n, l;
        char buf[512], *fld[5];
        Waitmsg *w;

        n = await(buf, sizeof buf-1);
        if(n < 0)
                return nil;
        buf[n] = ' ';
        if(tokenize(buf, fld, nelem(fld)) != nelem(fld)){
                werrstr("couldn't parse wait message");
                return nil;
        }
        l = strlen(fld[4])+1;
        w = malloc(sizeof(Waitmsg)+l);
        if(w == nil)
                return nil;
        w->pid = atoi(fld[0]);
        w->time[0] = atoi(fld[1]);
        w->time[1] = atoi(fld[2]);
        w->time[2] = atoi(fld[3]);
        w->msg = (char*)&w[1];
        memmove(w->msg, fld[4], l);
        return w;
}
```

This style of quoted-string and `tokenize` is used all through the system now. In particular, devices now `tokenize` the messages written to their `ctl` files, which means that you can send messages that contain white space, by quoting them, and that you no longer need to worry about whether or not the device accepts a newline. In other words, you can say

```
echo message > /dev/xx/ctl
```

instead of `echo -n` because `tokenize` treats the newline character as white space and discards it.

While we're on the subject of quotes and strings, note that the implementation of `await` used `snprint` rather than `sprint`. We now deprecate `sprint` because it has no protection against buffer overflow. We prefer `snprint` or `seprint`, to constrain the output. The %q format is cleverer than most in this regard: if the string is too long to be represented in full, %q is smart enough to produce a truncated but correctly quoted string within the available space.

Mount

Although strings in 9P are now variable-length and not zero-terminated, this has little direct effect in most of the system interfaces. File and user names are still zero-terminated strings as always; the kernel does the work of translating them as necessary for transport. And of course, they are now free to be as long as you might want; the only hard limit is that their length must be represented in 16 bits.

One example where this matters is that the file system specification in the `mount` system call can now be much longer. Programs like `rio` that used the specification string in creative ways were limited by the NAMELEN restriction; now they can use the string more freely. `Rio` now accepts a simple but less cryptic specification language for the window to be created by the `mount` call, e.g.:

```
    % mount $wsys /mnt/wsys 'new -dx 250 -dy 250 -pid 1234'
```

In the old system, this sort of control was impossible through the `mount` interface.

While we're on the subject of `mount`, note that with the new security architecture (see *factotum*(4)), 9P has moved its authentication outside the protocol proper. (For a full description of this change to 9P, see *fauth*(2), *attach*(5), and the paper *Security in Plan 9*.) The most explicit effect of this change is that `mount` now takes another argument, `afd`, a file descriptor for the authentication file through which the authentication will be made. For most user-level file servers, which do not require authentication, it is sufficient to provide `-1` as the value of `afd`:

```
    if(mount(fd, -1, "/mnt/wsys", MREPL,
       "new -dx 250 -dy 250 -pid 1234") < 0)
           error("mount failed: %r");
```

To connect to servers that require authentication, use the new `fauth` system call or the reimplemented `amount` (authenticated mount) library call. In fact, since `amount` handles both authenticating and non-authenticating servers, it is often easiest just to replace calls to `mount` by calls to `amount`; see *auth*(2) for details.

Print

The C library has been heavily reworked in places. Besides the changes mentioned above, it now has a much more complete set of routines for handling `Rune` strings (that is, zero-terminated arrays of 16-bit character values). The most sweeping changes, however, are in the way formatted I/O is performed.

The `print` routine and all its relatives have been reimplemented to offer a number of improvements:

(1) Better buffer management, including the provision of an internal flush routine, makes it unnecessary to provide large buffers. For example, `print` uses a much smaller buffer now (reducing stack load) while simultaneously removing the need to truncate the output string if it doesn't fit in the buffer.

(2) Global variables have been eliminated so no locking is necessary.

(3) The combination of (1) and (2) means that the standard implementation of `print` now works fine in threaded programs, and `threadprint` is gone.

(4) The new routine `smprint` prints into, and returns, storage allocated on demand by `malloc`.

(5) It is now possible to print into a `Rune` string; for instance, `runesmprint` is the `Rune` analog of `smprint`.

(6) There is improved support for custom print verbs and custom output routines such as error handlers. The routine `doprint` is gone, but `vseprint` can always be used instead. However, the new routines `fmtfdinit`, `fmtstrinit`, `fmtprint`, and friends are often a better replacement. The details are too long for exposition here; *fmtinstall*(2) explains the new interface and provides examples.

(7) Two new format flags, space and comma, close somewhat the gap between Plan 9 and ANSI C.

Despite these changes, most programs will be unaffected; `print` is still `print`. Don't forget, though, that you should eliminate calls to `sprint` and use the `%q` format when appropriate.

Binary compatibility

The discussion so far has been about changes at the source level. Existing binaries will probably run without change in the new environment, since the kernel provides backward-compatible system calls for `errstr`, `stat`, `wait`, etc. The only exceptions are programs that do either a `mount` system call, because of the security changes and because the file descriptor in `mount` must point to a new 9P connection; or a `read` system call on a directory, since the returned data will be in the new format. A moment's reflection will discover that this means old user-level file servers will need to be fixed to run on the new system.

File servers

A full description of what user-level servers must do to provide service with the new 9P is beyond the scope of this paper. Your best source of information is section 5 of the manual, combined with study of a few examples. `/sys/src/cmd/ramfs.c` is a simple example; it has a counterpart `/sys/src/lib9p/ramfs.c` that implements the same service using the new *9p*(2) library.

That said, it's worth summarizing what to watch for when converting a file server. The `session` message is gone, and there is a now a `version` message that is exchanged at the start of a connection to establish the version of the protocol to use (there's only one at the moment, identified by the string `9P2000`) and what the maximum message size will be. This negotiation makes it easier to handle 9P encapsulation, such as with `exportfs`, and also permits larger message sizes when appropriate.

If your server wants to authenticate, it will need to implement an authentication file and implement the `auth` message; otherwise it should return a helpful error string to the `Tauth` request to signal that authentication is not required.

The handling of `stat` and directory reads will require some changes but they should not be fundamental. Be aware that seeking on directories is forbidden, so it is fine if you disregard the file offset when implementing directory reads; this makes it a little easier to handle the variable-length entries. You should still never return a partial directory entry; if the I/O count is too small to return even one entry, you should return two bytes containing the byte count required to represent the next entry in the directory. User code can use this value to formulate a retry if it desires. See the DIAGNOSTICS section of *stat*(2) for a description of this process.

The trickiest part of updating a file server is that the `clone` and `walk` messages have been merged into a single message, a sort of 'clone-multiwalk'. The new message, still called `walk`, proposes a sequence of file name elements to be evaluated using a possibly cloned fid. The return message contains the qids of the files reached by walking to the sequential elements. If all the elements can be walked, the fid will be cloned if requested. If a non-zero number of elements are requested, but none can be walked, an error should be returned. If only some can be walked, the fid is not cloned, the original fid is left where it was, and the returned `Rwalk` message should contain the partial list of successfully reached qids. See *walk*(5) for a full description.