APE — The ANSI/POSIX Environment

Howard Trickey
howard@plan9.att.com

Introduction

When a large or frequently-updated program must be ported to or from Plan 9, the ANSI/POSIX environment known as APE can be useful. APE combines the set of headers and object code libraries specified by the ANSI C standard (ANSI X3.159-1989) with the POSIX operating system interface standard (IEEE 1003.1-1990, ISO 9945-1), the part of POSIX defining the basic operating system functions. Using APE will cause slower compilation and marginally slower execution speeds, so if the importing or exporting happens only infrequently, due consideration should be given to using the usual Plan 9 compilation environment instead. Another factor to consider is that the Plan 9 header organization is much simpler to remember and use.

There are some aspects of required POSIX behavior that are impossible or very hard to simulate in Plan 9. They are described below. Experience has shown, however, that the simulation is adequate for the vast majority of programs. A much more common problem is that many programs use functions or headers not defined by POSIX. APE has some extensions to POSIX to help in this regard. Extensions must be explicitly enabled with an appropriate `#define`, in order that the APE environment be a good aid for testing ANSI/POSIX compliance of programs.

Pcc

The `pcc` command acts as a front end to the Plan 9 C compilers and loaders. It runs an ANSI C preprocessor over source files, using the APE headers to satisfy `#include` <*file*> directives; then it runs a Plan 9 C compiler; finally, it may load with APE libraries to produce an executable program. The document *How to Use the Plan 9 C Compiler* explains how environment variables are used by convention to handle compilation for differing architectures. The environment variable `$objtype` controls which Plan 9 compiler and loader are used by `pcc`, as well as the location of header and library files. For example, if `$objtype` is `mips`, then `pcc` has `cpp` look for headers in `/mips/include/ape` followed by `/sys/include/ape`; then `pcc` uses `vc` to create `.v` object files; finally, `vl` is used to create an executable using libraries in `/mips/lib/ape`.

Psh and Cc

The `pcc` command is intended for uses where the source code is ANSI/POSIX, but the programs are built in the usual Plan 9 manner — with `mk` and producing object files with names ending in `.v`, etc. Sometimes it is best to use the standard POSIX `make` and `cc` (which produces object files with names ending in `.o`, and automatically calls the loader unless `-c` is specified). Under these circumstances, execute the command:

```
ape/psh
```

This starts a POSIX shell, with an environment that includes the POSIX commands `ar89`, `c89`, `cc`, `basename`, `dirname`, `expr`, `false`, `grep`, `kill`, `make`, `rmdir`, `sed`, sh, stty, true, uname, and yacc. There are also a few placeholders for commands that cannot be implemented in Plan 9: chown, ln, and umask.

The cc command accepts the options mandated for the POSIX command c89, as specified in the C-Language Development Utilities Option annex of the POSIX Shell and Utilities standard. It also accepts the following nonstandard options: -v for echoing the commands for each pass to stdout; -A to turn on ANSI prototype warnings; -S to leave assembly language in *file*.s; -Wp, *args* to pass *args* to the cpp; -W0, *args* to pass *args* to 2c, etc.; -Wl, *args* to pass *args* to 2l, etc.

The sh command is ksh93, a POSIX-compliant Korn Shell. The Plan 9 implementation does not include the emacs and vi editing modes. Ksh93 is a separate product, so its source code is not included in the Plan 9 distribution.

The stty command only has effect if the ape/ptyfs command has been started to interpose a pseudo-tty interface between /dev/cons and the running command. None of the distributed commands do this automatically.

Symbols

The C and POSIX standards require that certain symbols be defined in headers. They also require that certain other classes of symbols not be defined in the headers, and specify certain other symbols that may be defined in headers at the discretion of the implementation. POSIX defines *feature test macros*, which are preprocessor symbols beginning with an underscore and then a capital letter; if the program #defines a feature test macro before the inclusion of any headers, then it is requesting that certain symbols be visible in the headers. The most important feature test macro is _POSIX_SOURCE: when it is defined, exactly the symbols required by POSIX are visible in the appropriate headers. Consider <signal.h> for example: ANSI defines some names that must be defined in <signal.h>, but POSIX defines others, such as sigset_t, which are not allowed according to ANSI. The solution is to make the additional symbols visible only when _POSIX_SOURCE is defined.

To export a program, it helps to know whether it fits in one of the following categories:

1. Strictly conforming ANSI C program. It only uses features of the language, libraries, and headers explicitly required by the C standard. It does not depend on unspecified, undefined, or implementation-dependent behavior, and does not exceed any minimum implementation limit.
2. Strictly conforming POSIX program. Similar, but for the POSIX standard as well.
3. Some superset of POSIX, with extensions. Each extension is selected by a feature test macro, so it is clear which extensions are being used.

With APE, if headers are always included to declare any library functions used, then the set of feature test macros defined by a program will show which of the above categories the program is in. To accomplish this, no symbol is defined in a header if it is not required by the C or POSIX standard, and those required by the POSIX standard are protected by #ifdef _POSIX_SOURCE. For example, <errno.h> defines EDOM, ERANGE, and errno, as required by the C standard. The C standard allows more names beginning with E, but our header defines only those unless _POSIX_SOURCE is defined, in which case the symbols required by POSIX are also defined. This means that a program that uses ENAMETOOLONG cannot masquerade as a strictly conforming ANSI C program.

Pcc and cc do not predefine any preprocessor symbols except those required by the ANSI C standard: __STDC__, __LINE__, __FILE__, __DATE__, and __TIME__. Any others must be defined in the program itself or by using -D on the command line.

Extensions

The discipline enforced by putting only required names in the headers is useful for exporting programs, but it gets in the way when importing programs. The compromise is to allow additional symbols in headers, additional headers, and additional library functions, but only under control of extension feature test macros. The following extensions are provided; unless otherwise specified, the additional library functions are in the default APE library.

_LIBG_EXTENSION. This allows the use of the Plan 9 graphics library. The functions are as described in the Plan 9 manual (see *graphics* (2)) except that div had to be renamed ptdiv. Include the <libg.h> header to declare the needed types and functions.

_LIMITS_EXTENSION. POSIX does not require that names such as PATH_MAX and OPEN_MAX be defined in <limits.h>, but many programs assume they are defined there. If _LIMITS_EXTENSION is defined, those names will all be defined when <limits.h> is included.

_BSD_EXTENSION. This extension includes not only Berkeley Unix routines, but also a grab bag of other miscellaneous routines often found in Unix implementations. The extension allows the inclusion of any of: <bsd.h> for bcopy(), bcmp(), and similar Berkeley functions; <netdb.h> for gethostbyname(), etc., and associated structures; <select.h> for the Berkeley select function and associated types and macros for dealing with multiple input sources; <sys/ioctl.h> for the ioctl function (minimally implemented); <sys/param.h> for NOFILES_MAX; <sys/pty.h> for pseudo-tty support via the ptsname(int) and ptmname(int) functions; <sys/resource.h>; <sys/socket.h> for socket structures, constants, and functions; <sys/time.h> for definitions of the timeval and timezone structures; and <sys/uio.h> for the iovec structure and the writev and readv functions used for scatter/gather I/O. Defining _BSD_EXTENSION also enables various extra definitions in <ctype.h>, <signal.h>, <stdio.h>, <unistd.h>, <sys/stat.h>, and <sys/times.h>.

_NET_EXTENSION. This extension allows inclusion of <libnet.h>, which defines the networking functions described in the Plan 9 manual page *dial* (2).

_REGEXP_EXTENSION. This extension allows inclusion of <regexp.h>, which defines the regular expression matching functions described in the Plan 9 manual page *regexp* (2).

_RESEARCH_SOURCE. This extension enables a small library of functions from the Tenth Edition Unix Research System (V10). These functions and the types needed to use them are all defined in the <libv.h> header. The provided functions are: srand, rand, nrand, lrand, and frand (better random number generators); getpass, tty_echoon, tty_echooff (for dealing with the common needs for mucking with terminal characteristics); min and max; nap; and setfields, getfields, and getmfields (for parsing a line into fields). See the Research Unix System Programmer s Manual, Tenth Edition, for a description of these functions.

Common Problems

Some large systems, including X11, have been ported successfully to Plan 9 using APE (the X11 port is not included in the distribution, however, because supporting it properly is too big a job). The problems encountered fall into three categories: (1) non-ANSI C/POSIX features used; (2) inadequate simulation of POSIX functions; and (3) compiler/loader bugs. By far the majority of problems are in the first category.

POSIX is just starting to be a target for programmers. Most existing code is written to work with one or both of a BSD or a System V Unix. System V is fairly close to POSIX, but there are some differences. Also, many System V systems have imported some BSD features that are not part of POSIX. A good strategy for porting external programs is to first try using CFLAGS=-D_POSIX_SOURCE; if that doesn't work, try adding _D_BSD_EXTENSION and perhaps include <bsd.h> in source files. Here are some solutions to problems that might remain:

- Third (environment) argument to main. Use the environ global instead.
- OPEN_MAX, PATH_MAX, etc., assumed in <limits.h>. Rewrite to call sysconf or define _LIMITS_EXTENSION.
- <varargs.h>. Rewrite to use <stdarg.h>.

The second class of problems has to do with inadequacies in the Plan 9 simulation of POSIX functions. These shortcomings have rarely gotten in the way (except perhaps, for the link problem).

- Functions for setting the userid, groupid, effective userid and effective groupid do not do anything useful. The concept is impossible to simulate in Plan 9. Chown also does nothing.
- execlp and the related functions do not look at the PATH environment variable. They just try the current directory and /bin if the pathname is not absolute.
- Advisory locking via fcntl is not implemented.
- isatty is hard to do correctly. The approximation used is only sometimes correct.
- link always fails.
- With open, the O_NOCTTY option has no effect. The concept of a controlling tty is foreign to Plan 9.
- setsid forks the name space and note group, which is only approximately the right behavior.
- The functions dealing with stacking signals, sigpending, sigprocmask and sigsuspend, do not work.
- umask has no effect, as there is no such concept in Plan 9.
- code that does getenv("HOME") should be changed to getenv("home") on Plan 9.