NICHOLAS M. STOUGHTON

an update on standards

Nick is the USENIX Standards Liaison and represesnts the Association in the POSIX, ISO, C, and LSB working groups. He is the ISO organizational representative to the Austin group, a member of INCITS committees J11 and CT22, and the Specification Authority subgroup leader for the LSB.

nick@usenix.org

As you know if you've been following this column, the POSIX standard is undergoing a revision. This is the third official full revision since it first became a standard in 1988. In this article, we'll take a more detailed look at some of the new interfaces that are planned for inclusion in the revised standard. There are four separate sets of new interfaces, each of which is currently an official Open Group specification.

SET 1: GENERAL INTERFACES

There are several extremely useful interfaces in the GNU C library, glibc, many of which are also found in other vendors' libraries. These interfaces can be broadly grouped into the following categories:

- Directory handling: alphasort(), dirfd(), and scandir().
- Signal handling: psignal() and psiginfo().
- Standard I/O extensions: dprintf(), fmemopen(), getdelim(), getline(), open_memstream(), and open_wmemstream().
- Temporary files: mkdtemp().
- String handling: stpcpy(), stpncpy(), strndup(), strnlen(), strsignal(), mbsnrtowcs(), wcpcpy(), wcpncpy(), wcscasecmp(), wcsdup(), wcsnlen(), and wcsnrtombs().

I don't plan to describe each and every one of these interfaces in detail, but there are some interesting points to note. First and foremost is the relationship between this project and the Technical Report the ISO C committee is preparing on "bounds checking interfaces." Although the ISO C document contains newly invented functions to supplement the standard I/O and string handling functions of the ISO C standard, it will only be a Technical Report. This is not the same as a Standard; it is a way of testing the water, providing a trial-use period to see whether industry is interested in going that way. At this point, a few companies have indicated an interest in that approach, including both Microsoft and Cisco.

However, the interfaces listed above will be going into the POSIX standard and will have the full weight of an International Standard to them. They are not invention, and they have been implemented (quite probably on the system you are using). Many of them solve the same problem, buffer overflow, that the ISO C Technical Report tries to, but in a very different way. There is a second part to the ISO C technical report planned, which will reference many of these new POSIX interfaces as better alternatives if you are designing new programs. On the other hand, if you are retrofitting large, established code bases to fix potential buffer overflows, then the ISO C inventions may be useful.

Two interfaces fmemopen() and open_memstream(), are particularly interesting, in that they provide a way of performing standard I/O to dynamically allocated memory buffers. Consider the following:

```
char *
itos (int i)
{
    FILE *f;
    size_t len;
    char *buf;

    if((f = open_memstream(&buf,
    &len)) == NULL)
            return NULL;
    fprintf(f, “%d”, i);
    fclose(f);
    return buf;
}
```

Although this is a rather trivial use of the new functionality, it serves to illustrate the point. The function converts an integer to a string, allocating space for the string as required. A more conventional program might have chosen to use a static buffer and assumed that the size of an integer was *n* bits, and therefore the maximum length that the string could ever be was *m* bytes. And the program would have overflowed its buffer when ported to a system with a larger size of integer.

SET 2: PATHNAMES RELATIVE TO OPEN DIRECTORIES

Solaris 10 introduced a handful of file system interfaces to work on files with extended attributes. These interfaces were all named with an ...at() suffix, and they took a file descriptor of an open directory as the first argument. Relative pathnames are relative to the open directory, and not (necessarily) relative to the current working directory. For example, openat() behaves as ordinary open(), except that it takes an additional argument, the file descriptor for relative pathnames. In the Solaris case, openat() accepts an additional value, O_XATTR, for the file mode.

In the glibc case, the extended attributes part of these interfaces was dropped, but the concept of handling pathnames relative to an open directory proved a powerful mechanism for addressing a number of security and other related issues, and so the concept was extended to all system interfaces that took a pathname. One other interface in this set is fexecve(), which is similar to execve() except that it executes the file on an open file descriptor. This allows, for example, a program to open the file it is about to execute, lock it, checksum it, and only execute it if it matches the expected checksum. Without fexecve(), an application that attempted to do this would suffer a vulnerability that the file could be replaced between successfully checksumming it and executing it.

One other useful feature of this set is the ability to avoid (or at least postpone) buffer overflow with pathnames that exceed PATH_MAX bytes.

The complete list of interfaces in this set is as follows: faccessat(), fchmodat(), fchownat(), fdopendir(), fexecve(), fstatat(), futimesat(), linkat(), mkdirat(), mkfifoat(), mknodat(), openat(), readlinkat(), renameat(), symlinkat(), and unlinkat().

One other noteworthy point must be made here: futimesat() may yet change its name and functionality slightly. There is an intention in this revision of POSIX to include file timestamps with nanosecond granularity. Until now POSIX has specified only one-second granularity on files. However, almost all OS vendors now have support for a finer-grain resolution, typically at the nanosecond level. As processors get faster and faster, the ability for tools to be able portably to distinguish between a source file and a file generated from that source becomes more and more important. Since futimesat() is a new function, both in glibc (as far as I am aware, it has not been implemented anywhere else) and POSIX, this may be the best place to add support for setting file time stamps at this fine a granularity. This aspect is still under discussion in the committee.

SET 3: ROBUST MUTEXES

Developers of multi-threaded applications are probably well aware of the problems that can arise when a process terminates while one of its threads holds a mutex lock. While it is sometimes possible for another thread to unlock the mutex and recover its state, this is at best an unreliable and unportable mechanism.

Robust mutexes are introduced in this set of new interfaces. A robust mutex is simply a mutex with a special “robust” bit set in its attributes. Whenever a thread that owns a robust mutex terminates, current or future waiters on that mutex will be notified that the owner is dead. Another thread then has the opportunity to take over and clean up the state that was protected by the mutex and to make the mutex once again consistent.

One important feature of this proposal is that it is only intended to deal with abnormal termination of the process owning the mutex (e.g., if the process was subject to a signal). It is not intended to be a way to encourage bad programming and have applications simply not bother to clean up properly at exit, and so on; therefore, if a thread is terminated by cancellation or if it calls pthread_exit(), it is expected that that thread will handle its own cleanup properly (e.g., by registering appropriate cleanup handlers).

This set of interfaces includes pthread_mutex_consistent(), pthread_mutexattr_getrobust(), and pthread_mutexattr_setrobust(). It also alters the behavior of several other existing mutex APIs, essentially by adding the EOWNERDEAD error return.

SET 4: THREAD-AWARE LOCALES

The concept of locales to allow processes to have different natural-language interfaces has always been a part of POSIX. Until now, the process has been the object that is associated with a locale. This set of new APIs permits individual threads to be in different locales.

The major new concept in this set of interfaces is the locale_t object. Applications can create as many locale objects as they require, each one associated with a different locale. Each thread can then choose to use one of these locales, and in doing so does not affect the behavior of any other thread. Compare this with the old concept of the process as a whole being in a given locale; if one thread changed the locale, then all the threads in that process would be changed.

The fundamental interfaces in this set are newlocale(), duplocale(), freelocale(), and uselocale().

In addition to these, all of the ctype.h character categorization functions gain a new locale object counterpart. For example, as well as isalnum(int c), there is an isalnum_l(int c, locale_t l) interface. The former returns true if the character represented by c is alphanumeric. The new interface returns true if the character is alphanumeric in the locale represented by l.

THE TIMETABLE

The revision project has been working up to full steam over the past couple of years, but it is now in full-scale development mode. The first committee drafts appeared in July (as I write this article). The second draft, which will probably be the first one to be publicly balloted, is scheduled for November 2006. The document will probably take until April 2008 before it is completely approved. As always, the Austin Group welcomes any interested party to join the process. For details, see www.opengroup.org/austin.