Writing reentrant and threadsafe code

In single-threaded processes, only one flow of control exists. The code executed by these processes thus need not be reentrant or threadsafe. In multithreaded programs, the same functions and the same resources may be accessed concurrently by several flows of control.

To protect resource integrity, code written for multithreaded programs must be reentrant and threadsafe.

Reentrance and thread safety are both related to the way that functions handle resources. Reentrance and thread safety are separate concepts: a function can be either reentrant, threadsafe, both, or neither.

This section provides information about writing reentrant and threadsafe programs. It does not cover the topic of writing thread-efficient programs. Thread-efficient programs are efficiently parallelized programs. You must consider thread effiency during the design of the program. Existing single-threaded programs can be made thread-efficient, but this requires that they be completely redesigned and rewritten.

Reentrance

A reentrant function does not hold static data over successive calls, nor does it return a pointer to static data. All data is provided by the caller of the function. A reentrant function must not call non-reentrant functions.

A non-reentrant function can often, but not always, be identified by its external interface and its usage. For example, the **strtok** subroutine is not reentrant, because it holds the string to be broken into tokens. The **ctime** subroutine is also not reentrant; it returns a pointer to static data that is overwritten by each call.

Thread safety

A threadsafe function protects shared resources from concurrent access by locks. Thread safety concerns only the implementation of a function and does not affect its external interface.

In C language, local variables are dynamically allocated on the stack. Therefore, any function that does not use static data or other shared resources is trivially threadsafe, as in the following example: `/* threadsafe function */`

```
int diff(int x, int y)
{
       int delta;

       delta = y - x;
       if (delta < 0)
               delta = -delta;

       return delta;
}
```

The use of global data is thread-unsafe. Global data should be maintained per thread or encapsulated, so that its access can be serialized. A thread may read an error code corresponding to an error caused by another thread. In AIX®, each thread has its own **errno** value.

Making a function reentrant

In most cases, non-reentrant functions must be replaced by functions with a modified interface to be reentrant. Non-reentrant functions cannot be used by multiple threads. Furthermore, it may be impossible to make a non-reentrant function threadsafe.

Returning data

Many non-reentrant functions return a pointer to static data. This can be avoided in the following ways:

- Returning dynamically allocated data. In this case, it will be the caller's responsibility to free the storage. The benefit is that the interface does not need to be modified. However, backward compatibility is not ensured; existing single-threaded programs using the modified functions without changes would not free the storage, leading to memory leaks.
- Using caller-provided storage. This method is recommended, although the interface must be modified.

For example, a **strtoupper** function, converting a string to uppercase, could be implemented as in the following code fragment: `/* non-reentrant function */`

```
char *strtoupper(char *string)
{
       static char buffer[MAX_STRING_SIZE];
       int index;

       for (index = 0; string[index]; index++)
               buffer[index] = toupper(string[index]);
       buffer[index] = 0

       return buffer;
}
```

This function is not reentrant (nor threadsafe). To make the function reentrant by returning dynamically allocated data, the function would be similar to the following code fragment: `/* reentrant function (a poor solution) */`

```
char *strtoupper(char *string)
{
       char *buffer;
       int index;

       /* error-checking should be performed! */
       buffer = malloc(MAX_STRING_SIZE);

       for (index = 0; string[index]; index++)
               buffer[index] = toupper(string[index]);
       buffer[index] = 0

       return buffer;
}
```

A better solution consists of modifying the interface. The caller must provide the storage for both input and output strings, as in the following code fragment: `/*`

```
reentrant function (a better solution) */
```

```
char *strtoupper_r(char *in_str, char *out_str)
{
        int index;

        for (index = 0; in_str[index]; index++)
        out_str[index] = toupper(in_str[index]);
        out_str[index] = 0

        return out_str;
}
```

Te non-reentrant standard C library subroutines were made reentrant using caller-provided storage.

Keeping data over successive calls

No data should be kept over successive calls, because different threads may successively call the function. If a function must maintain some data over successive calls, such as a working buffer or a pointer, the caller should provide this data.

Consider the following example. A function returns the successive lowercase characters of a string. The string is provided only on the first call, as with the **strtok** subroutine. The function returns 0 when it reaches the end of the string. The function could be implemented as in the following code fragment: `/* non-reentrant`

```
function */
char lowercase_c(char *string)
{
        static char *buffer;
        static int index;
        char c = 0;

        /* stores the string on first call */
        if (string != NULL) {
                buffer = string;
                index = 0;
        }

        /* searches a lowercase character */
        for (; c = buffer[index]; index++) {
                if (islower(c)) {
                        index++;
                        break;
                }
        }
        return c;
}
```

h

This function is not reentrant. To make it reentrant, the static data, the **index** variable, must be maintained by the caller. The reentrant version of the function could be implemented as in the following code fragment:

```
/* reentrant function */
char reentrant_lowercase_c(char *string, int *p_index)
{
       char c = 0;

       /* no initialization - the caller should have done it */

       /* searches a lowercase character */
       for (; c = string[*p_index]; (*p_index)++) {
               if (islower(c)) {
                       (*p_index)++;
                       break;
                 }
       }
       return c;
}
```

The interface of the function changed and so did its usage. The caller must provide the string on each call and must initialize the index to 0 before the first call, as in the following code fragment:

```
char *my_string;
char my_char;
int my_index;
...
my_index = 0;
while (my_char = reentrant_lowercase_c(my_string, &my_index)) {
       ...
}
```

Making a function threadsafe

In multithreaded programs, all functions called by multiple threads must be threadsafe. However, a workaround exists for using thread-unsafe subroutines in multithreaded programs. Non-reentrant functions usually are thread-unsafe, but making them reentrant often makes them threadsafe, too.

Locking shared resources

Functions that use static data or any other shared resources, such as files or terminals, must serialize the access to these resources by locks in order to be threadsafe. For example, the following function is thread-unsafe:

```
/* thread-unsafe
function */
int increment_counter()
{
       static int counter = 0;

       counter++;
       return counter;
}
```

To be threadsafe, the static variable **counter** must be protected by a static lock, as in the following example:

```
/* pseudo-code threadsafe function */
```

```
int increment_counter();
{
        static int counter = 0;
        static lock_type counter_lock = LOCK_INITIALIZER;

        pthread_mutex_lock(counter_lock);
        counter++;
        pthread_mutex_unlock(counter_lock);
        return counter;
}
```

In a multithreaded application program using the threads library, mutexes should be used for serializing shared resources. Independent libraries may need to work outside the context of threads and, thus, use other kinds of locks.

Workarounds for thread-unsafe functions

It is possible to use a workaround to use thread-unsafe functions called by multiple threads. This can be useful, especially when using a thread-unsafe library in a multithreaded program, for testing or while waiting for a threadsafe version of the library to be available. The workaround leads to some overhead, because it consists of serializing the entire function or even a group of functions. The following are possible workarounds:

- Use a global lock for the library, and lock it each time you use the library (calling a library routine or using a library global variable). This solution can create performance bottlenecks because only one thread can access any part of the library at any given time. The solution in the following pseudocode is acceptable only if the library is seldom accessed, or as an initial, quickly implemented workaround. `/* this is pseudo code! */`

  ```
  lock(library_lock);
  library_call();
  unlock(library_lock);

  lock(library_lock);
  x = library_var;
  unlock(library_lock);
  ```
- Use a lock for each library component (routine or global variable) or group of components. This solution is somewhat more complicated to implement than the previous example, but it can improve performance. Because this workaround should only be used in application programs and not in libraries, mutexes can be used for locking the library. `/* this is pseudo-code! */`

  ```
  lock(library_moduleA_lock);
  library_moduleA_call();
  unlock(library_moduleA_lock);

  lock(library_moduleB_lock);
  x = library_moduleB_var;
  unlock(library_moduleB_lock);
  ```

Reentrant and threadsafe libraries

Reentrant and threadsafe libraries are useful in a wide range of parallel (and asynchronous) programming environments, not just within threads. It is a good programming practice to always use and write reentrant and threadsafe functions.

Using libraries

Several libraries shipped with the AIX Base Operating System are threadsafe. In the current version of AIX, the following libraries are threadsafe:

- Standard C library (**libc.a**)
- Berkeley compatibility library (**libbsd.a**)

Some of the standard C subroutines are non-reentrant, such as the **ctime** and **strtok** subroutines. The reentrant version of the subroutines have the name of the original subroutine with a suffix **_r** (underscore followed by the letter *r*).
When writing multithreaded programs, use the reentrant versions of subroutines instead of the original version. For example, the following code fragment:

```
token[0] =
strtok(string, separators);
i = 0;
do {
       i++;
       token[i] = strtok(NULL, separators);
} while (token[i] != NULL);
```

should be replaced in a multithreaded program by the following code fragment:

```
char
*pointer;
...
token[0] = strtok_r(string, separators, &pointer);
i = 0;
do {
       i++;
       token[i] = strtok_r(NULL, separators, &pointer);
} while (token[i] != NULL);
```

Thread-unsafe libraries may be used by only one thread in a program. Ensure the uniqueness of the thread using the library; otherwise, the program will have unexpected behavior, or may even stop.

Converting libraries

Consider the following when converting an existing library to a reentrant and threadsafe library. This information applies only to C language libraries.

- Identify exported global variables. Those variables are usually defined in a header file with the **export** keyword. Exported global variables should be encapsulated. The variable should be made private (defined with the **static** keyword in the library source code), and access (read and write) subroutines should be created.
- Identify static variables and other shared resources. Static variables are usually defined with the **static** keyword. Locks should be associated with any shared resource. The granularity of the locking, thus choosing the number of locks, impacts the performance of the library. To initialize the locks, the one-time initialization facility may be used.

- Identify non-reentrant functions and make them reentrant. For more information, see Making a Function Reentrant.
- Identify thread-unsafe functions and make them threadsafe. For more information, see Making a Function threadsafe.

Parent topic: Multithreaded programming

Parent topic: General programming concepts

Related concepts:
One-time initializations

Related information:
admin
cdc
delta
get
prs
sccsdiff
sccsfile