Use reentrant functions for safer signal handling

How and when to employ reentrancy to keep your code bug free

Dipak Jha (dipakjha@in.ibm.com)
Software Engineer
IBM

20 January 2005

If you deal with concurrent access of functions, either by threads or processes, you can face problems caused by non-reentrancy of the functions. In this article, learn through code samples how anomalies can result if reentrancy is not ensured, especially with regard to signals. Five recommended programming practices are included, along with a discussion of a proposed compiler model in which the compiler front end deals with reentrancy.

In the early days of programming, non-reentrancy was not a threat to programmers; functions did not have concurrent access and there were no interrupts. In many older implementations of the C language, functions were expected to work in an environment of single-threaded processes.

Now, however, concurrent programming is common practice, and you need to be aware of the pitfalls. This article describes some potential problems due to non-reentrancy of the function in parallel and concurrent programming. Signal generation and handling in particular add extra complexity. Due to the asynchronous nature of signals, it is difficult to point out the bug caused when a signal-handling function triggers a non-reentrant function.

This article:

- Defines reentrancy and includes a POSIX listing of a reentrant function
- Provides examples to show problems caused by non-reentrancy
- Suggests ways to ensure reentrancy of the underlying function
- Discusses dealing with reentrancy at the compiler level

What is reentrancy?

A *reentrant* function is one that can be used by more than one task concurrently without fear of data corruption. Conversely, a *non-reentrant* function is one that cannot be shared by more than one task unless mutual exclusion to the function is ensured either by using a semaphore or by disabling interrupts during critical sections of code. A reentrant function can be interrupted at any time and resumed at a later time without loss of data. Reentrant functions either use local variables or protect their data when global variables are used.

A reentrant function:

- Does not hold static data over successive calls
- Does not return a pointer to static data; all data is provided by the caller of the function
- Uses local data or ensures protection of global data by making a local copy of it
- Must not call any non-reentrant functions

Don't confuse reentrance with thread-safety. From the programmer perspective, these two are separate concepts: a function can be reentrant, thread-safe, both, or neither. Non-reentrant functions cannot be used by multiple threads. Moreover, it may be impossible to make a non-reentrant function thread-safe.

IEEE Std 1003.1 lists 118 reentrant UNIX® functions, which aren't duplicated here. See Resources for a link to the list at unix.org.

The rest of the functions are non-reentrant because of any of the following:

- They call `malloc` or `free`
- They are known to use static data structures
- They are part of the standard I/O library

Signals and non-reentrant functions

A *signal* is a software interrupt. It empowers a programmer to handle an asynchronous event. To send a signal to a process, the kernel sets a bit in the signal field of the process table entry, corresponding to the type of signal received. The ANSI C prototype of a signal function is:

```
void (*signal (int sigNum, void (*sigHandler)(int))) (int);
```

Or, in another representation:

```
typedef void sigHandler(int);
SigHandler *signal(int, sigHandler *);
```

When a signal that is being caught is handled by a process, the normal sequence of instructions being executed by the process is temporarily interrupted by the signal handler. The process then continues executing, but the instructions in the signal handler are now executed. If the signal handler returns, the process continues executing the normal sequence of instructions it was executing when the signal was caught.

Now, in the signal handler you can't tell what the process was executing when the signal was caught. What if the process was in the middle of allocating additional memory on its heap using `malloc`, and you call `malloc` from the signal handler? Or, you call some function that was in the middle of the manipulation of the global data structure and you call the same function from the signal handler. In the case of `malloc`, havoc can result for the process, because `malloc` usually maintains a linked list of all its allocated area and it may have been in the middle of changing this list.

An interrupt can even be delivered between the beginning and end of a C operator that requires multiple instructions. At the programmer level, the instruction may appear atomic (that is, cannot be divided into smaller operations), but it might actually take more than one processor instruction to complete the operation. For example, take this piece of C code:

```
temp += 1;
```

On an x86 processor, that statement might compile to:

```
mov ax,[temp]
inc ax
mov [temp],ax
```

This is clearly not an atomic operation.

This example shows what can happen if a signal handler runs in the middle of modifying a variable:

Listing 1. Running a signal handler while modifying a variable

```
#include <signal.h>
#include <stdio.h>

struct two_int { int a, b; } data;

void signal_handler(int signum){
   printf ("%d, %d\n", data.a, data.b);
   alarm (1);
}

int main (void){
 static struct two_int zeros = { 0, 0 }, ones = { 1, 1 };

 signal (SIGALRM, signal_handler);

 data = zeros;

 alarm (1);

while (1)
  {data = zeros; data = ones;}
}
```

This program fills `data` with zeros, ones, zeros, ones, and so on, alternating forever. Meanwhile, once per second, the alarm signal handler prints the current contents. (Calling `printf` in the handler is safe in this program, because it is certainly not being called outside the handler when the signal happens.) What output do you expect from this program? It should print either 0, 0 or 1, 1. But the actual output is as follows:

```
0, 0
1, 1

(Skipping some output...)

0, 1
1, 1
1, 0
1, 0
...
```

On most machines, it takes several instructions to store a new value in `data`, and the value is stored one word at a time. If the signal is delivered between these instructions, the handler might find that `data.a` is 0 and `data.b` is 1, or vice versa. On the other hand, if we compile and run this code on a machine where it is possible to store an object's value in one instruction that cannot be interrupted, then the handler will always print 0, 0 or 1, 1.

Another complication with signals is that, just by running test cases you can't be sure that your code is signal-bug free. This complication is due to the asynchronous nature of signal generation.

Non-reentrant functions and static variables

Suppose that the signal handler uses `gethostbyname`, which is non-reentrant. This function returns its value in a static object:

```
static struct hostent host; /* result stored here*/
```

And it reuses the same object each time. In the following example, if the signal happens to arrive during a call to `gethostbyname` in `main`, or even after a call while the program is still using the value, it will clobber the value that the program asked for.

Listing 2. Risky use of gethostbyname

```
main(){
  struct hostent *hostPtr;
  ...
  signal(SIGALRM, sig_handler);
  ...
  hostPtr = gethostbyname(hostNameOne);
  ...
}

void sig_handler(){
  struct hostent *hostPtr;
  ...
  /* call to gethostbyname may clobber the value stored during the call
  inside the main() */
  hostPtr = gethostbyname(hostNameTwo);
  ...
}
```

However, if the program does not use `gethostbyname` or any other function that returns information in the same object, or if it always blocks signals around each use, you're safe.

Many library functions return values in a fixed object, always reusing the same object, and they can all cause the same problem. If a function uses and modifies an object that you supply, it is potentially non-reentrant; two calls can interfere if they use the same object.

A similar case arises when you do I/O using streams. Suppose the signal handler prints a message with `fprintf` and the program was in the middle of an `fprintf` call using the same stream when the signal was delivered. Both the signal handler's message and the program's data could be corrupted, because both calls operate on the same data structure: the stream itself.

Things become even more complicated when you're using a third-party library, because you never know which parts of the library are reentrant and which are not. As with the standard library, there can be many library functions that return values in fixed objects, always reusing the same objects, which causes the functions to be non-reentrant.

The good news is, these days many vendors have taken the initiative to provide reentrant versions of the standard C library. You'll need to go through the documentation provided with any given library to know if there is any change in the prototypes and therefore in the usage of the standard library functions.

Practices to ensure reentrancy

Sticking to these five best practices will help you maintain reentrancy in your programs.

Practice 1

Returning a pointer to static data may cause a function to be non-reentrant. For example, a `strToUpper` function, converting a string to uppercase, could be implemented as follows:

Listing 3. Non-reentrant version of strToUpper

```
char *strToUpper(char *str)
{
        /*Returning pointer to static data makes it non-reentrant */
       static char buffer[STRING_SIZE_LIMIT];
       int index;

       for (index = 0; str[index]; index++)
                buffer[index] = toupper(str[index]);
       buffer[index] = '\0';
       return buffer;
}
```

You can implement the reentrant version of this function by changing the prototype of the function. This listing provides storage for the output string:

Listing 4. Reentrant version of strToUpper

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

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

        return out_str;
}
```

Providing output storage by the calling function ensures the reentrancy of the function. Note that this follows a standard convention for the naming of reentrant function by suffixing the function name with "_r".

Practice 2

Remembering the state of the data makes the function non-reentrant. Different threads can successively call the function and modify the data without informing the other threads that are using the data. If a function needs to maintain the state of some data over successive calls, such as a working buffer or a pointer, the caller should provide this data.

In 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 follows:

Listing 5. Non-reentrant version of getLowercaseChar

```
char getLowercaseChar(char *str)
{
        static char *buffer;
        static int index;
        char c = '\0';
        /* stores the working string on first call only */
        if (string != NULL) {
                buffer = str;
                index = 0;
        }

        /* searches a lowercase character */
        while(c=buff[index]){
         if(islower(c))
         {
             index++;
             break;
         }
        index++;
       }

      return c;
}
```

This function is not reentrant, because it stores the state of the variables. To make it reentrant, the static data, the `index` variable, needs to be maintained by the caller. The reentrant version of the function could be implemented like this:

Listing 6. Reentrant version of getLowercaseChar

```
char getLowercaseChar_r(char *str, int *pIndex)
{

        char c = '\0';

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

        /* searches a lowercase character */

       while(c=buff[*pIndex]){
          if(islower(c))
          {
             (*pIndex)++; break;
          }
       (*pIndex)++;
       }
         return c;
}
```

Practice 3

On most systems, `malloc` and `free` are not reentrant, because they use a static data structure that records which memory blocks are free. As a result, no library functions that allocate or free memory are reentrant. This includes functions that allocate space to store a result.

The best way to avoid the need to allocate memory in a handler is to allocate, in advance, space for signal handlers to use. The best way to avoid freeing memory in a handler is to flag or record the objects to be freed and have the program check from time to time whether anything is waiting to be freed. But this must be done with care, because placing an object on a chain is not atomic, and if it is interrupted by another signal handler that does the same thing, you could "lose" one of the objects. However, if you know that the program cannot possibly use the stream that the handler uses at a time when signals can arrive, you are safe. There is no problem if the program uses some other stream.

Practice 4

To write bug-free code, practice care in handling process-wide global variables like `errno` and `h_errno`. Consider the following code:

Listing 7. Risky use of errno

```
if (close(fd) < 0) {
  fprintf(stderr, "Error in close, errno: %d", errno);
  exit(1);
}
```

Suppose a signal is generated during the very small time gap between setting the `errno` variable by the `close` system call and its return. The generated signal can change the value of `errno`, and the program behaves unexpectedly.

Saving and restoring the value of `errno` in the signal handler, as follows, can resolve the problem:

Listing 8. Saving and restoring the value of errno

```
void signalHandler(int signo){
  int errno_saved;

  /* Save the error no. */
  errno_saved = errno;

  /* Let the signal handler complete its job */
  ...
  ...

  /* Restore the errno*/
  errno = errno_saved;
}
```

Practice 5

If the underlying function is in the middle of a critical section and a signal is generated and handled, this can cause the function to be non-reentrant. By using signal sets and a signal mask, the critical region of code can be protected from a specific set of signals, as follows:

1. Save the current set of signals.
2. Mask the signal set with the unwanted signals.
3. Let the critical section of code complete its job.
4. Finally, reset the signal set.

Here is an outline of this practice:

Listing 9. Using signal sets and signal masks

```
sigset_t newmask, oldmask, zeromask;
...
/* Register the signal handler */
signal(SIGALRM, sig_handler);

/* Initialize the signal sets */
sigemtyset(&newmask); sigemtyset(&zeromask);

/* Add the signal to the set */
sigaddset(&newmask, SIGALRM);

/* Block SIGALRM and save current signal mask in set variable 'oldmask'
*/
sigprocmask(SIG_BLOCK, &newmask, &oldmask);

/* The protected code goes here
...
...
*/

/* Now allow all signals and pause */
sigsuspend(&zeromask);

/* Resume to the original signal mask */
sigprocmask(SIG_SETMASK, &oldmask, NULL);

/* Continue with other parts of the code */
```

Skipping `sigsuspend(&zeromask);` can cause a problem. There has to be some gap of clock cycles between the unblocking of signals and the next instruction carried by the process, and any occurrence of a signal in this window of time is lost. The function call `sigsuspend` resolves this problem by resetting the signal mask and putting the process to sleep in a single atomic operation. If you are sure that signal generation in this window of time won't have any adverse effects, you can skip `sigsuspend` and go directly to resetting the signal.

Dealing with reentrancy at the compiler level

I would like to propose a model for dealing with reentrant functions at the compiler level. A new keyword, `reentrant`, can be introduced for the high-level language, and functions can be given a `reentrant` specifier that will ensure that the functions are reentrant, like so:

```
reentrant int foo();
```

This directive instructs the compiler to give special treatment to that particular function. The compiler can store this directive in its symbol table and use it during the intermediate code generation phase. To accomplish this, some design changes are required in the compiler's front end. This reentrant specifier follows these guidelines:

1. Does not hold static data over successive calls
2. Protects global data by making a local copy of it
3. Must not call non-reentrant functions
4. Does not return a reference to static data, and all data is provided by the caller of the function

Guideline 1 can be ensured by type checking and throwing an error message if there is any static storage declaration in the function. This can be done during the semantic analysis phase of the compilation.

Guideline 2, protection of global data, can be ensured in two ways. The primitive way is by throwing an error message if the function modifies global data. A more sophisticated technique is to generate intermediate code in such a way that the global data doesn't get mangled. An approach similar to Practice 4, above, can be implemented at the compiler level. On entering the function, the compiler can store the to-be-manipulated global data using a compiler-generated temporary name, then restore the data upon exiting the function. Storing data using a compiler-generated temporary name is normal practice for the compiler.

Ensuring guideline 3 requires the compiler to have prior knowledge of all the reentrant functions, including the libraries used by the application. This additional information about the function can be stored in the symbol table.

Finally, guideline 4 is already guaranteed by guideline 2. There is no question of returning a reference to static data if the function doesn't have one.

This proposed model would make the programmer's job easier in following the guidelines for reentrant functions, and by using this model, code would be protected against the unintentional reentrancy bug.

Resources

- You can read or download IEEE Std 1003.1 from unix.org, a Web site of The Open Group (registration is required to view or download the document).
- Starting with Synchronization is not the enemy (developerWorks, July 2001), this series of three articles covers issues of threading and concurrency when programming in the Java™ language.
- PowerPC developers will appreciate the insights presented in Save your code from meltdown using PowerPC atomic instructions (developerWorks, November 2004); it describes techniques for safe concurrent programming in PowerPC assembly language.
- Good background for UNIX programmers includes UNIX Network Programming by W. Richard Stevens and Design of the Unix Operating System by Maurice J. Bach.
- Find more resources for Linux developers in the developerWorks Linux zone.
- Get involved in the developerWorks community by participating in developerWorks blogs.
- Browse for books on these and other technical topics.

About the author

Dipak Jha

Dipak provides Level 3 support for Distributed File System (DFS). His work involves kernel- and user-level debugging of dumps and crashes, as well as fixing the reported bugs on the AIX and Solaris platforms. Contact Dipak at dipakjha@in.ibm.com.