Tip 66: Protect threaded resources with mutexes

Ben Klemens

11 February 2012

level: You've got down the pthreads form from last time (Entry #115)
purpose: when needed, do one thing at a time

As of last episode (Entry #115), we have multiple threads all looking at the same registers in memory. Say that thread one is writing to an integer `i`, while thread two is reading from `i`. I can use the word *while* because both writing and reading are procedures that a computer requires finite time to execute. A simple single line of C code like `i++` will expand to some number of instructions in assembly code, which will expand to some number of operations in which the processor causes transistors in memory to change state. You do not want to care about this level of detail.

Back at the level of C, we thus have the *mutex*, which provides mutual exclusion. We will associate a mutex with `i`, and any thread may lock the mutex, so that when other threads try to claim the mutex, they are locked out and have to wait. So, for example:

1. The write thread claims the mutex for `i`, begins writing.
2. The read thread tries to claim the mutex, is locked out.
3. The write thread keeps writing.
4. The read thread pings the mutex—¿Can I come in now? It is rejected.
5. The write thread is done and releases the mutex.
6. The read thread pings the mutex, and is allowed to continue. It locks the mutex.
7. The write thread is back to write more data. It pings the mutex, but is locked out.

Et cetera. The read thread is guaranteed that there won't be shenanigans about transistors in memory changing state mid-read, and the write thread is similarly guaranteed that things will be clean.

So any time we have a resource, like stdout or a variable, at least one thread that wants to modify the resource, and two or more threads that will read or write the resource, we will attach a mutex to the resource. At the head of each thread's code to use the resource, we lock the mutex; at the end of that block of code, we release the mutex.

What if one thread never gives up a mutex? Maybe it's caught in an infinite loop. Then all the threads that want to use that mutex are stuck at the step where they are pinging the mutex over and over, so if one thread is stuck, they all are. In fact, *stuck pinging a mutex that is never released* sounds an awful lot like an infinite loop. Here is an easily-possible story regarding two threads and two mutexes:

1. Thread A locks mutex one.
2. Thread B locks mutex two.
3. Thread A find that it has to use the resource protected by mutex two. It pings mutex two, is locked out, and so sleeps.
4. Thread B needs to check on a resource protected by mutex one. It pings mutex one, is locked out, because thread A is sleeping and therefore can't possibly release mutex one.

Now you're in a deadlock, as A waits for B and B waits for A. If you are starting out with threading, I recommend that any given block of code lock one mutex at a time. You can often make this work by just associating a mutex with a larger block of code that you may have thought deserved multiple mutexes.

C11 provides an `_Atomic` keyword, which is also useful to resolve some threading annoyances. It'll be great when it's supported by compilers; meanwhile I can't write about it.

Mutexes, however, are common and easy. Glib provides a setup.

A quick note on `static` **variables** All of the `static` variables in a program, meaning those declared outside of a function plus those inside a function with the `static` keyword, are shared across all threads. Same with anything `malloc`ed (that is, each thread may call `malloc` to produce a pocket of memory for its own use, but any thread that has the address could conceivably use the data there). Automatic variables are specific to each thread.

C11 provides a keyword, `_Thread_local`, that splits off a static variable (either in the file-global scope or in a function via the `static` keyword) so that each thread has its own version, but the variable still behaves like a static variable when determining scope and whether it gets erased at the end of a function. C11's new keyword seems to be an emulation the GCC-specific `__thread` keyword. If this is useful to you, within a function you can use either of:

```
static __thread int i;       //GCC-specific; works today.
   // or
static _Thread_local int i; //C11, when your compiler implements it.
```

Outside of a function, you don't need the word `static`.

The example Let us rewrite the `wc` function from last time to print a message every thousand words. If you throw a lot of threads at the program, you'll be able to see if they all get equal time or run in serial despite our best efforts. There's also a global counter that gets updated every thousand words. The natural (and more accurate—the method here counts only sets of 1,000) way to do such a count is by tallying everything at the end, but this is a simple example so we can focus on wiring up the mutex.

If you comment the mutex lines, you'll be able to watch the threads walking all over each other. To facilitate this, I wrote

```
for (int i=0; i< out->wc; i++) global_wc++;
```

which is entirely equivalent to

```
global_wc += out->wc;
```

but takes up more processor time.

We'll need to add `gthread-2.0` to our makefile flags to get this running. Here's what my (slightly edited) makefile looks like at this point. I set the optimization level to `-O0` so that when we turn of the mutexes, there are more instructions that can trample each other.

```
OBJECTS =fstr.o
CFLAGS =-g -Wall -std=gnu99 -O0 `pkg-config --cflags glib-2.0 gthread-2.0` -pthr
LDLIBS=`pkg-config --libs glib-2.0 gthread-2.0`
CC=gcc

$(P): $(OBJECTS)
```

All of the mutex-oriented changes were inside the function itself. By allocating a mutex to be a static variable, all threads see it. Then, each thread by itself tries the lock before screwing with the global word count, and unlocks when finished with the shared variable.

Here's the code. Please note: as of about a month ago, glib's static mutex has been considered deprecated, because it's so much more useful than the non-static mutex that they basically renamed the static mutex to be the general-purpose one. I'll come back in a few months and rewrite this code accordingly.

```
#include "fstr.h"
#include <pthread.h>
#include <string.h> //strtok_r
#include <glib.h> //mutexes

long int global_wc;

typedef struct{
    int wc;
    char *docname;
} wc_struct;

void *wc(void *voidin){
    wc_struct *in = voidin;
    fstr_t *doc = fstr_new(in->docname);
    static GStaticMutex count_lock = G_STATIC_MUTEX_INIT;

    wc_struct *out = calloc(1, sizeof(wc_struct));

    char *scratch;
    char *delimiters = " `~!@#$%^&*()_-+={[]}|\\;:\",<>./?\n\t"; //rmed apostrop
    char *txt = strtok_r(doc->data, delimiters, &scratch);
    if (!txt) return out; //zero elements.
    while (txt) {
        txt = strtok_r(NULL, delimiters, &scratch);
        out->wc++;

        if (!(out->wc % 1000)){
            g_static_mutex_lock(&count_lock);
            for (int i=0; i< out->wc; i++) global_wc++; //a slow global_wc += ou
            printf("%s has reached a count of %i words here; %li words total.\n"
                    , in->docname, out->wc, global_wc);
            g_static_mutex_unlock(&count_lock);
        }
    }
    return out;
}

int main(int argc, char **argv){
    argc--;
    argv++; //step past the name of the program.

    pthread_t threads[argc];
    wc_struct s[argc];
    for (int i=0; i< argc; i++){
        s[i] = (wc_struct){.docname=argv[i]};
        pthread_create(&threads[i], NULL, wc, &s[i]);
    }

    int values[argc];
    void *rvalue;
    for (int i=0; i< argc; i++){
        pthread_join(threads[i], &rvalue);
        values[i] = ((wc_struct*)rvalue)->wc;
    }

    for (int i=0; i< argc; i++) printf("%s:\t%i\n",argv[i], values[i]);
}
```

To do:

Try this on a few dozen files. I used the complete works of Shakspeare, because I have Gutenberg's Shakespeare[1] broken up by play; I'm sure you've got some files on hand to try out. After you run it as is, comment out the lock/unlock lines, and re-run. Do you get the same counts?

[1]`http://www.gutenberg.org/ebooks/100`