Tip 65: Easy threading with Pthreads

Ben Klemens

9 February 2012

level: Computationally-intensive work
purpose: Use all the processors you've got

If your computer is less than about five years old and is not a telephone, then it has several *cores*, which are independent pipelines for processing. Run `top` right now and hit the 1 key to break out the list of CPUs (though I'm not sure if that's POSIX-standard), and you can see how many you have and how busy they are.

Threading is billed as a complex and arcane topic, but when I first implemented threads in a program, I was delighted by how easy it is to get a serial loop to become a set of threads.

POSIX has had support for threads for forever. C11 adds a C-standard thread library, which I'd like to cover for you, but I'm not yet sure if there's an extant standard C library that implements the threading portion of the new standard.

So the syntax is really not a thing. The hard part is in dealing with the details of how threads interact.

The simplest case, sometimes called a process which is embarrassingly parallel[1], is when you are applying the same task to every element of an array, and each is independent of the other. Any shared data is used read-only. As per the nickname for this kind of thing, this is the easy case, and I'll show you the syntax for making it work here.

The first complication is when there is some resource that is writeable, and is shared across elements. For example, say that we'd like to write a message to stdout as events happen, and we have a badly implemented version of `printf`. If two threads write to stdout at once, then the messages may mangle each other. Instead of

```
Adding another agent.
Removing an agent.
```

we might get mangling like

```
AddingRem anooving ather agent.n age
nt.
```

[I added that *badly implemented* caveat because POSIX dictates that `printf` and family must be thread-safe, meaning that this sort of mangling can't happen. I can't find a straight answer, but I'd be surprised if the Windows standard library's `printf` family isn't also thread-safe.]

[1]`http://en.wikipedia.org/wiki/Embarrassingly_parallel`

Here, we need a *mutex*, which locks a resource (here, stdout) while in use by one thread, and tells the other threads that want to use the resource to hold on until the prior thread is done and releases the lock. Using mutexes will come up next episode.

When you have multiple mutexes that may interact, or you have one thread reading data that was written by another thread, then you're up to rocket science. It's easy for threads to get into states where both are waiting for the other to release a mutex, or for our expectations about the rate at which data gets read or written, and debugging this sort of thing is now a question of the luck of replicating the surprising order of execution when the debugger is running. But I expect that the simple stuff I'll cover here will already be enough for you to safely speed up your code.

The pthreads checklist You have a `for` loop over elements, such as an operation on every element of an array. As above, no iteration has any bearing on any other. If the iterations were run in random order, you wouldn't really care, as long as every array element gets hit once and only once.

We're going to turn that serial `for` loop into parallel threads. It's not trivial, but it's not all that difficult either. We are going to write a single function that will be applied to each element of the array, using `pthread_create`, and then use `pthread_join` to wait for each thread to return. At the end of that disperse/gather procedure, the program can continue as if nothing special had happened.

1. For GCC and Clang, add `-pthreads` to the compiler line. If you're using a makefile, then set something like `CFLAGS=-pthreads -g -Wall`.
2. `#include <pthreads.h>`
3. Write a wrapper function, which will be called by every thread. It has to have a signature of the form `void * your_function (void *)`. That is, it takes one void pointer in, and spits one void pointer out. If you started with a `for` loop, paste the body of the loop into this wrapper, and do the appropriate surgery so that you are acting on the function input instead of element i of the array.
4. Disperse the threads: your `for` loop now applies `pthread_create` to each array element. See the example below.
5. Gather the threads: Write a second `for` loop to call `pthread_join` to gather all of the threads and check their return values.

OK, here's the example. I am sorry to say that it is a word counter, which is such a typical example. However, it's at least a pretty zippy one, which runs about 3x faster than `wc`. [The definitions of a word also differ, so it'll be hard to seriously compare, though.] You should at this point be able to follow the above gather/disperse procedure. I'll point out several little details after the text.

You'll notice that I call in `fstr.h`, over from Tip #52 (Entry #102), so compile with that.

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

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

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

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

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

- The `fstr` line reads the given document into a string; see tip #52 (Entry #102) if you'd like details.

- The POSIX-standard `strtok_r` function cuts a string up at a given set of delimiters. Each call will return one more delimited element. If your delimiter is a single character, this is much faster than a regex and is much more pleasant than inspecting a string element-by-element. The `_r` stands for *reentrant*, because the plain `strtok` function has only one internal scratch space and so can't thread. Over the course of this, the input string on the first call gets written to `scratch` and is mangled.

- You can see that the function takes in a void pointer, so the first line has to be a cast, and the output is a void pointer, so we are forced to use `calloc`. So it goes.

- `argv[0]` is the name of the program, so we skip that, as per tip #58 (Entry #108). The rest of the arguments on the command line are files to be word-counted.

- Now we have the thread creation step. We set up a list of thread info pointers, and then we send to `pthread_create` one of those, the wrapper function, and an item to send in to the wrapper function. Don't worry about the second argument, which controls some threading attributes.

- The throwaway typedef, `wc_struct`, adds immense safety. I still have to be careful to write the inputs and outputs to the pthread system correctly, but the internals of the struct get type-checked, both in `main` and in the wrapper function. Next week, when I change `wc` to a `long int`, the compiler will warn me if I don't do the change correctly. I decided to use one struct for both input and output; if you think it would be more readable if these were different structs, I would not bicker with you.

- The next loop gathers outputs. You can see that `pthread_join` copies the output void pointer to its second argument; then you have to save that output value somewhere. Here we have to cast to a pointer-to-int, and then we can read the data as an integer and save it to `values[i]`.

To do:

Now that you have the form down (or at least, you have a template you can cut and paste), check your code for embarrassingly parallel loops, and thread 'em.

Above, I gave each row of an array one thread; how would you split something into a more sensible number, like two or three threads? Hint: you've got a struct, so you can send in extra info, like start/end points for each thread.