

Re: lines of code?

Source: <http://coding.derkeiler.com/Archive/General/comp.programming/2003-12/1028.html>

From: goose (*ruse_at_webmail.co.za*)

Date: 12/12/03

Date: 12 Dec 2003 04:18:17 -0800

Chris Smith <cdsmith@twu.net> wrote in message
news:<MPG.1a42133afc6bc2b498980c@news.pop4.net>...

> *Hi goose,*

>

> *Thanks for your reply. If you don't mind, I'm going to simplify your
> response by eliminating the part about embedded systems. That seems to
> be your area, and the same ideas apply in either case... but I'm
> guessing that for the majority of the readership, high-level libraries
> such as UNIX select() and other means of async I/O are going to be the
> way to go.*

No problem.

>

> *goose wrote:*

> > *Your "alternative" to multiple threads is to "jump back and forth
> > from one task to another [in] the same thread". I'd suggest that
> > if your /solution/ demands multiple threads and/or tasks, then go
> > ahead and use it, but dont try to implement a multiple-thread
> > solution in a single thread. that just wont work nicely.*

>

> *Just to clarify, we are talking about whether multithreading is ever
> justified on a uniprocessor, right? That's what I thought we're talking
> about. It sounds like you're saying that I should use multiple threads
> on a uniprocessor in some cases.*

Nope. If your **solution** is **designed** as lots of seperate tasks,
then perhaps creating threads for each task is best. My contention
is that a lot of **designs** that calls for multiple threads
dont really get anything more than a single-threaded design would.

In other words, very many wizz-bang multi-threaded apps would be
no better in performance and ease of maintenance than single-threaded
apps (very often, there is a performance degradation for the multiple
threads).

> *If you're saying that, then perhaps I*

> *just imagined the difference of opinion in the first place.*

comp.programming: Re: lines of code?

> *Nevertheless, you later said:*
>
> > *in a server environment, lets say a unix machine of some sort*
> > *(or even windows nt or better), you want to read from the*
> > *network *and* do user input ? no problem, use select() to read*
> > *from the network;*
>
> *That's making some assumptions about the problem that I was careful not*
> *to make.*

but thats the example given (to do user input *and* read from a network at the same time). Anyone who can do this *only* via the use of multiple threads has missed something.

> *Specifically, the goals I meant to discuss were intended to be*
> *sufficiently involved as to require some degree of complex state on the*
> *stack. I further intended that there be two fairly "live" goals*
> *involved (and you'll notice that I mean something like "real-time", but*
> *I'm being careful not to say real-time because I don't want to imply the*
> *kind of substantial penalties for delay that are a connotation of that*
> *term). So no, I'm not saying that threads are required every time a*
> *network connection is used in an application.*
>
> *Here's a very specific example:*
>

some clarification required. my comments below.

> *Let's say I've got an application that reads some kind of TCP-based*
> *streaming video protocol from one incoming network connection, performs*
> *a transformation on it, and streams the resulting video out another*
> *network connection.s*

a TCP-based protocol cannot, afaik, be "streaming". this is because of the latency inherent in a connection-based protocol. perhaps some protocol built on IP(UDP), which requires no virtual-circuit setup, and no ack after X windows transmitted?

(sidenote: iirc, most, if not all, video/sound streaming protocols *dont* use TCP, they use UDP).

> *Both streaming protocols are rather involved,*
> *requiring context-sensitive back-and-forth between client and server to*
> *negotiate the data transfer rate and video characteristics required, and*
> *the implementation of such a server ends up involving multiple levels of*
> *recursion.*

I do not understand. the implementation of such a *server* ends up involving multiple levels of recursion ? "recursion" is a design decision. Are you certain you do not mean to say "protocol" instead of "server" ?

Re: lines of code?

comp.programming: Re: lines of code?

- > *The transformation itself may be simple, but the application*
- > *then has to implement the same very involved streaming video protocol*
- > *(from the other side) to retransmit the transformed data.*

and here you say protocol. which do you mean, server or protocol.

- >
- > *The problem with using select is this: every time you call select, it*
- > *might give you any amount of data in either direction.*

yes.

- > *So you read*
- > *enough data to get seven deep in function calls in the streaming client*
- > *code, and then run out of available data. However, you *do* have a*
- > *frame available now to transmit to your *own* client, and you were also*
- > *waiting for available space in the buffer to do so. That space is now*
- > *available. Implementing the protocol to transmit that frame to the*
- > *client now is going to leave you five function calls deep in the*
- > *streaming server code. At that point, you want to return to receiving*
- > *data from your own source. How do you do that?*
- >
- > *The only reasonable answer is that you rewrite*

not "rewrite". if you had designed recursion in, then yes, you would have to rewrite. if you had not, then no, theres no need to rewrite.

you took a design decision to *use* recursion, along with the pros and cons of recursion in this particular example. as a result, your design /needs/ multiple threads.

taking a design decision to use multiple threads, then showing how difficult it would be to write the *same* design in a single thread isn't very conducive to your argument :-).

- > *those trasmit and receive*
- > *libraries so that they don't make function calls that will take time.*
- > *That means that you move their use of recursion to use of an explicitly*
- > *managed external stack, in conjunction with some kind of a deterministic*
- > *state machine to manage the protocol.*

if you are writing a protocol stack, a deterministic state machine is unavoidable.

- > *Now, that's not a desirable*
- > *outcome: given a choice, most people would far rather read and*
- > *understand a top-down implementation of a context-ful network protocol*

I do not understand what you mean by this, in particular "context-ful". given a choice when reading code for a protocol stack, I'll take a state-machine over any other type of design any day.

Re: lines of code?

comp.programming: Re: lines of code?

> *than a (state machine + external data) implementation. You are being
> forced into the latter case by your choice to avoid multiple threads.*

nope. if you really are writing a protocol stack (no matter how complex), then you really **should** be doing it with a state machine anyway. what good is recursion if the other party hangs in a manner that sends you recursing forever?

the only times I have seen protocol stacks implemented in a manner /other/ than state machines is when the constraints required something else (language being used did not allow pointers to functions, protocol was too simple to require one, etc).

>
> > *however my rant was about coming up with a solution that involves
> > multiple threads merely because the environment offers it.*
>
> *Yet you continued to disagree when Michael and I both pointed out that
> there are legitimate reasons for multithreading on a uniprocessor.*

Micheal, iirc, never pointed out anything: heres the relevant posting (snipped irrelevant text):

ruse@webmail.co.za (goose) wrote in message
news:<ff82ae1b.0312090727.67b761e9@posting.google.com>...
> *Michael Borgwardt <brazil@brazils-animeland.de> wrote in message
news:<br40r5\$28g5p5\$1@ID-161931.news.uni-berlin.de>...*
> > *goose wrote:*
> > > > *unless your are programming for an architecture with more than
> > > > one CPU, multi-threaded code does not actually buy you anything.*
> > > >
> > > *Although I agree with the rest of your posting, this is nonsense.*
> > >
> > >
> > > *before you dismiss it totally, consider that:*
> > > *a) I never mentioned any particular language.*
> > > *b) Java threads can be user-land threads, and not always native
> > > threads.*
> > > *c) I was talking in the context of native threads.*
> > > *d) multi-threaded programming with native threads which is
> > > aimed at a single processor machine really *is* useless.*
> >
> > *All of these points are completely irrelevant to the argument.*
>
> *I beg to differ. My original post is "complete nonsense" only if
> I was designing a solution in java.*
>
<snipped>

comp.programming: Re: lines of code?

a few points to consider from the above post:

- a) I should have qualified my statement about multiple threads not actually buying you anything ("multiple threads dont **always** get you anything more than single-threaded", for example).
- b) Micheal never pointed out that there were legitimate reasons for using threads. his subsequent posts made it very clear that his use of multiple threads was in the context of java gui; I'm no java expert, but I am under the impression that java is kinda useless if you cannot have more than one thread; select() functionality has only recently (2 years?) been added to java, no ?
- c) In order for you (or Micheal) to justify the "this is nonsense" response to my "does not buy you anything", you would have to show that all multi-threaded design is superior to single-threaded, i.e. you claimed it is nonsense, now prove it. I've claimed that it does not buy you anything, and I've given examples where it did not buy you anything. You've[1] claimed that my claim is nonsensense, so now prove it.
- d) I did not "continue to disagree", I only continued to disagree that my post was "nonsense".

> *Do*

> *you agree with these legitimate reasons, or not?*

there are so few legitimate reasons for designing in multiple threads, that even the example you gave could possibly be redesigned to /not/ use multiple threads.

there **are** legitimate reasons, possibly one out of every 10 "solutions" designed with multiple-threads of execution should really be multiple-threaded (or 1 out of every 100, or 1 out of every 1000, etc ad nauseum). I've come across at **least** 10 multi-threaded apps that /should not/ have been multi-threaded, and only 1 that was legitimately so.

>

> *Of course, in embedded systems, interrupts solve **some** of this problem,*

> *exactly because they provide the ability to do some of what threads do:*

no, they provide entirely different functionality, for example, you can be assured that an interrupt routine will run to completion (you can't be sure of anything like that within a thread), will **never** lose data as long as their is buffer space, will not be starved if thread priorities change, etc.

a thread is a seperate but concurrent (to the programmer) path of execution, an interrupt routine is not, even to the programmer.

> *interrupt an involved task, saving its entire state, and go do something*

Re: lines of code?

comp.programming: Re: lines of code?

- > *else. Even there, though, if the task to be done on the basis of the*
- > *interrupt is also involved and stateful, you run into problems because*
- > *the stack for the interrupt context is generally not preserved;*

you had better clarify your usage of the word "stack". I don't understand what you mean by that last sentence. the /stack/ on most machines is not available anyway, all you have is scope for your variables.

what do you mean?

- > *so you*
- > *can deal with one stateful task + one interactive (non-batch) task, but*
- > *two tasks that are *both* stateful and interactive is still beyond you*
- > *without resorting to jmping through hoops to move the entire state off*
- > *of the program stack.*

once again, I don't know where you've gotten that idea. consider a small program on an embedded chip (any embedded chip) with an embedded OS (say ... uCOS). you want two "tasks" that interact with the user? no can do. if you do have two threads that both call display routines, you are heading for a possible race condition (at best, at worst your product will crash at crucial moments).

what you *should* do is either

- a) have a separate thread/task for display only. it sits in a loop and displays stuff from a circular buffer (with control codes if necessary). your other tasks will have to call a display routine which will enter a critical section, add the data to the buffer, exit the critical section and then return.

or

- b) have all your various modules as state-machines. have a function pointer to each one (lcd_state_machine, keypad_state_machine, sio_state_machine, etc); at startup init all of them to start states, then sit in a loop and run each state machine in turn. things like sio will be interrupt driven anyway, lcd won't be.

I know which one can be *guaranteed* not to get deadlocked, or cause a race: hint, it's not a :-).

this of course, is also applicable to large server development as well. your protocol stack written with recursion will fall over miserably the minute someone figures out the right bit of data or header to repeat when talking to it, a state-machine otoh, can easily tell if junk is coming along the way (although one designed to loop states (pumping lemma) will hang not crash, as it uses no extra resources like the stack does in recursion).

>

- > > *the idea is not to squeeze every last bit of performance out of the*

Re: lines of code?

comp.programming: Re: lines of code?

- > > *system, but to make it easier and simpler to understand. a single*
- > > *thread of execution is *always* easier to read and then draw a diagram*
- > > *off, especially if the code is all new to you.*
- >
- > *Reading the above, do you still agree?*

I find the example above a rather nasty one for your argument. it helps my argument more than it does yours :-).

also note that I now clarify: there are very **few** legitimate uses of threads. its up to you now to either say that my claim is "nonsense", or to show so many legitimate uses of multiple-threads that i cow and shiver in abject terror ;-).

(at least give 10 legitimate uses of threads, because right now I can think of ten pieces of application software all of which use threads, but find that only 1 of them has what I would call a legitimate use of threads).

seriously though, I **have** given an example of a use of threads, in another post in this same thread (:-), so you could not seriously have been thinking that I decry **all** uses of threads. otoh, I got the impression from you and Micheal (more from Micheal, to be honest), that pointing at the fact that threads are needed so little is a bit like insulting a mans wife. Micheal has not acknowledged (and nor have you, for that matter) that a lot of thread-happy designs need not be that way.

- > *Is it really easier to*
- > *understand a single-threaded program that implements complex network*
- > *protocols as deterministic state machines?*

yes. deterministic state machines have something going for them. if you get the state-diagram right, you're 90% done. the complex network protocol that is going to decode/encode the protocol with recursive routines wont last too long; I'd be surprised if someone did not figure out that he could slow the machine to a crawl if he repeated the right data.

- > *I would find that hard to*
- > *believe.*

its not about belief;
goose,
the lawnmower man.