Multitasking using setjmp, longjmp

Asked 9 years, 5 months ago Active 3 years, 2 months ago Viewed 7k times

17

11

is there a way to implement multitasking using `setjmp` and `longjmp` functions

longjmp

edited Mar 30 '11 at 1:05

osgx
60.3k 33 261 419

asked Apr 1 '10 at 13:50

GingerJack
2,008 1 13 17

1 Tony Finch's picoro(small co-routines). Co-routines are in Knuth's the art of computing and are co-operative multi-tasking. As well, Simon Tatham has a co-routines web page with nice explanations. – artless noise Feb 24 '14 at 18:10

Also, care should be taken; the `setjmp()` and `longjmp()` are most often/always implemented in assembler and resemble OS context switch code. However, they may not save some state such as *floating point*, *SIMD state*, etc. Whether this is an implementation bug or a standards issue, I don't know. However, this issue will often exist in practice. Knowing what state to save can be a significant boost to context switch speeds. – artless noise Feb 25 '14 at 18:58

See: `setjmp()` and fpmode for more on other CPU state. – artless noise Feb 25 '14 at 22:19

4 Answers

12

You can indeed. There are a couple of ways to accomplish it. The difficult part is initially getting the jmpbufs which point to other stacks. Longjmp is only defined for jmpbuf arguments which were created by setjmp, so there's no way to do this without either using assembly or exploiting undefined behavior. User level threads are inherently not portable, so portability isn't a strong argument for not doing it really.

step 1 You need a place to store the contexts of different threads, so make a queue of jmpbuf stuctures for however many threads you want.

Step 2 You need to malloc a stack for each of these threads.

Step 3 You need to get some jmpbuf contexts which have stack pointers in the memory locations you just allocated. You could inspect the jmpbuf structure on your machine, find out where it stores

disassemble it, and then find instructions it executes when you return from a function, you can find out what the offset ought to be. For example, with system V calling conventions on x86, you'll see that it pops %ebp (the frame pointer) and then calls ret which pops the return address off the stack. So on entry into a function, it pushes the return address and frame pointer. Each push moves the stack pointer down by 4 bytes, so you want the stack pointer to start at the high address of the allocated region, -8 bytes (as if you just called a function to get there). We will fill the 8 bytes next.

The other thing you can do is write some very small (one line) inline assembly to manipulate the stack pointer, and then call setjmp. This is actually more portable, because in many systems the pointers in a jmpbuf are mangled for security, so you can't easily modify them.

I haven't tried it, but you might be able to avoid the asm by just deliberately overflowing the stack by declaring a very large array and thus moving the stack pointer.

Step 4 You need exiting threads to return the system to some safe state. If you don't do this, and one of the threads returns, it will take the address right above your allocated stack as a return address and jump to some garbage location and likely segfault. So first you need a safe place to return to. Get this by calling setjmp in the main thread and storing the jmpbuf in a globally accessible location. Define a function which takes no arguments and just calls longjmp with the saved global jmpbuf. Get the address of that function and copy it to your allocated stacks where you left room for the return address. You can leave the frame pointer empty. Now, when a thread returns, it will go to that function which calls longjmp, and jump right back into the main thread where you called setjmp, every time.

Step 5 Right after the main thread's setjmp, you want to have some code that determines which thread to jump to next, pulling the appropriate jmpbuf off the queue and calling longjmp to go there. When there are no threads left in that queue, the program is done.

Step 6 Write a context switch function which calls setjmp and stores the current state back on the queue, and then longjmp on another jmpbuf from the queue.

Conclusion That's the basics. As long as threads keep calling context switch, the queue keeps getting repopulated, and different threads run. When a thread returns, if there are any left to run, one is chosen by the main thread, and if none are left, the process terminates. With relatively little code you can have a pretty basic cooperative multitasking setup. There are more things you probably want to do, like implement a cleanup function to free the stack of a dead thread, etc. You can also implement preemption using signals, but that is much more difficult because setjmp doesn't save the floating point register state or the flags registers, which are necessary when the program is interrupted asynchronously.

edited Mar 22 '13 at 14:39

answered Mar 21 '13 at 18:03

Sean Ogden

138 1 7

Some specific implementations of setjmp/longjmp may work in such a fashion that one can jinx them to behave as desired, and it's possible that some compilers might even *specify* that their implementations work in a particular fashion that would allow such a thing without having to rely upon undocumented/undefined behavior when targeting such compilers, but I'd suggest instead using a few lines of assembly code to do the stack/register switches. Using setjmp/longjmp is no more portable than assembly code, but might give an illusion of portability. – supercat Mar 21 '13 at 18:10

That having been said, I think there's a lot to be said for cooperative multi-tasking. Many compilers expressly document what registers (if any) need to be preserved by external assembly-language modules. A pre-emptive multitasker would have to preserve all registers that a compiler might be using, which could

...things like C++ exceptions, depending upon how they're implemented, may or may not play nice with even cooperative multitasking. One would have to research how exceptions are implemented to know what is required of the stacks maintained by the running threads. – supercat Mar 21 '13 at 18:17

1 If you use the stack overflow method to adjust the stack pointer, there's no asm, and the approach works for all machines in which the stack grows down and the stack frames begin with RA and FP (need to use sizeof(int*) to get the appropriate size for the offsets). That covers pretty much all x86/AMD64 machines that use Windows, OSX or Linux. – Sean Ogden Mar 22 '13 at 15:09

1 A lot of C++ implementations use one or more static variables to control exception handling; trying to switch between threads that use exceptions will require that the task-switcher know about such variables and swap them. Swapping them isn't hard--making sure there aren't any static variables that one doesn't know about is the hard part. – supercat Mar 23 '13 at 21:40

8

It may be bending the rules a little, but GNU pth does this. It's possible, but you probably shouldn't try it yourself except as an academic proof-of-concept exercise, use the pth implementation if you want to do it seriously and in a remotely portable fashion -- you'll understand why when you read the pth thread creation code.

(Essentially it uses a signal handler to trick the OS into creating a fresh stack, then longjmp's out of there and keeps the stack around. It works, evidently, but it's sketchy as hell.)

In production code, if your OS supports makecontext/swapcontext, use those instead. If it supports CreateFiber/SwitchToFiber, use those instead. And be aware of the disappointing truth that one of the most compelling use of coroutines -- that is, inverting control by yielding out of event handlers called by foreign code -- is unsafe because the calling module has to be reentrant, and you generally can't prove that. This is why fibers still aren't supported in .NET...

answered Aug 9 '10 at 5:41

user414736
81 1 1

2 The Netscape Portable Runtime (NSPR) also appears to define macros for doing this using a simpler but hairier method: they just call setjmp and then change the machine stack pointer and instruction pointer in the buffer. Google "_MD_INIT_CONTEXT" for an entertaining read. – user414736 Aug 9 '10 at 7:10

3

This is a form of what is known as userspace context switching.

It's possible but error-prone, especially if you use the default implementation of setjmp and longjmp. One problem with these functions is that in many operating systems they'll only save a subset of 64-bit registers, rather than the entire context. This is often not enough, e.g. when dealing with system libraries (my experience here is with a custom implementation for amd64/windows, which worked pretty stable all things considered).

That said, if you're not trying to work with complex external codebases or event handlers, and you know what you're doing, and (especially) if you write your own version in assembler that saves more of the current context (if you're using 32-bit windows or linux this might not be necessary, if you use some versions of BSD I imagine it almost definitely is), and you debug it paying careful attention to the disassembly output, then you may be able to achieve what you want.

answered Nov 5 '11 at 17:08

1

As was already mentioned by Sean Ogden, longjmp() is not good for multitasking, as it can only move the stack upward and can't jump between different stacks. No go with that.

As mentioned by user414736, you can use getcontext/makecontext/swapcontext functions, but the problem with those is that they are not fully in user-space. They actually call the sigprocmask() syscall because they switch the signal mask as part of the context switching. This makes swapcontext() much slower than longjmp(), and you likely don't want the slow co-routines.

To my knowledge there is no POSIX-standard solution to this problem, so I compiled my own from different available sources. You can find the context-manipulating functions extracted from libtask here:
https://github.com/stsp/dosemu2/tree/devel/src/arch/linux/mcontext
The functions are: getmcontext(), setmcontext(), makemcontext() and swapmcontext(). They have the similar semantic to the standard functions with similar names, but they also mimic the setjmp() semantic in that getmcontext() returns 1 (instead of 0) when jumped to by setmcontext().

On top of that you can use a port of libpcl, the coroutine library:
https://github.com/stsp/dosemu2/tree/devel/src/base/misc/libpcl
With this, it is possible to implement the fast cooperative user-space threading. It works on linux, on i386 and x86_64 arches.

edited Jul 12 '16 at 12:23

answered Jan 24 '16 at 0:34

stsp
83 6