Is there a way to make a function atomic in C?

Is there a way to make a function atomic in C.

I am not looking for a portable solution.(platforms looking for - Win,Linux)

c atomic operations

edited Apr 1 '09 at 8:12

asked Apr 1 '09 at 7:45

FL4SOF
790 2 10 21

If you're not looking for a portable solution, you'll have to specify your platform(s) – snemarch Apr 1 '09 at 8:10

7 Answers

Maybe.

It depends entirely on your definition of "atomic".

- In a single core, deeply embedded environment without an operating system involved you can usually disable and enable interrupts. This can be used to allow a function to be atomic against interrupt handler code. But if you have a multi-master bus, a DMA engine, or some other hardware device that can write memory independently, then even masking interrupts might not provide a strong enough guarantee in some circumstances.
- In an RTOS (real time operating system) environment, the OS kernel usually provides low level synchronization primitives such as critical sections. A critical section is a block of code that behaves "essentially" atomically, at least with respect to all other critical sections. It is usually fundamental to the OS's implementation of other synchronization primitives.
- In a multi-core environment, a low level primitive called a spinlock is often available. It is used to guard against entry to a block of code that must be atomic with respect to other users of the same spinlock object, and operates by blocking the waiting CPU core in a tight loop until the lock is released (hence the name).
- In many threading environments, more complex primitives such as events, semaphores, mutexes, and queues are provided by the threading framework. These cooperate with the thread scheduler such that threads waiting for something to happen don't run at all until the condition is met. These can be used to make a function's actions atomic with respect to other threads sharing the same synchronization object.

A general rule would be to use the highest level capabilities available in your environment that are suited to the task. In the best case, an existing thread safe object such as a message queue can be used to avoid needing to do anything special in your code at all.

answered Apr 1 '09 at 8:09

RBerteig
24.6k 45 85

If you want to make sure your function won't be interrupted by signal, use `sigprocmask()` to mask and unmask signals, although some signals cannot be blocked (like `SIGKILL`) and behaviour for blocking some signals (like `SIGSEGV`) is undefined.

See `man sigprocmask` for details.

edited Apr 1 '09 at 7:59

answered Apr 1 '09 at 7:49

qrdl
18.7k 7 28 64

Not portably, at least. For some systems, you can probably approach it, by doing things like turning of machine interrupts, to prevent the kernel from pre-empting your function. But it will be very hard, especially for non-embedded systems.

answered Apr 1 '09 at 7:47

unwind
200k 30 270 387

You'll need platform-specific support to do that - either through use of special compiler intrinsics for hardware instructions, or by using operating system support. Neither C nor C++ has standardized synchronization stuff.

answered Apr 1 '09 at 7:49

snemarch
3,444 12 25

Define what you mean by 'atomic.' Do you mean 'atomic' in the sense that no other process or thread will be selected for scheduling when you run your function? Or do you mean that any shared objects referenced in your function will not be modified by any other threads while your function runs?

For the former, you can't really control that from userspace. If you're on a single-CPU machine you can **possibly** guarantee atomicity by raising your process priority to the highest priority possible (from userspace). But even then it's not guaranteed because your scheduling algorithm may still allow another process to run. The only reliable way of doing this is from the operating system. For a single-CPU machine you'd disable interrupts. For a multi-core machine you'd need to lock the bus and wait for all processes running on other CPUs to be taken off.

The question here is: **why do you want to guarantee atomicity?** In general, the requirement that only your process may be running and not others, ought not to exist in userspace. If you want to make sure certain data structures are only accessed by one thread at a time, then you should use a portable thread library (like `pthread` for example), and fence off your function as a critical section.

answered Apr 1 '09 at 13:58

FreeMemory
5,069 3 21 34

If by atomic you mean 'only one thread at a time', then you could just protect the function with a Critical Section block (in Windows). In Linux, I use a mutex lock/unlock to more or less emulate a critical section.

answered Apr 1 '09 at 15:48

Marc Bernier
1,863 13 26

You might want to look into POSIX semaphores, mutexes or the like, which might work on both Windows and Linux.

Using e.g. cygwin or minGW it's even possible to write portable code between Linux and Windows.

Even better you can create windows libs on Linux:
http://cdtdoug.blogspot.com/2009/05/mingw-cross-for-linux.html

answered May 18 '09 at 17:11

robert.berger
3,102 1 7 5