How can SIGINT be safely delivered on the main thread?

devblogs.microsoft.com/oldnewthing/20080728-00

July 28, 2008

Raymond Chen

Commenter AnotherMatt wonders why Win32 console programs deliver console notifications on a different thread. Why doesn't it deliver them on the main thread?

Actually, my question is the reverse. Why does unix deliver it on the main thread? It makes it nearly impossible to do anything of consequence inside the signal handler. The main thread might be inside the heap manager (holding the heap critical section) when the signal is raised. If the signal handler tried to access the heap, it would deadlock with itself if you're lucky, or just corrupt the heap if you aren't.

For example, consider this signal handler:

```
void catch_int(int sig_num)
{
    /* re-set the signal handler again to catch_int, for next time */
    signal(SIGINT, catch_int);
    /* and print the message */
    printf("Don't do that");
    fflush(stdout);
}
```

What happens if the signal is raised while the main program is executing its own `fflush`, say after it had already flushed half the buffer? If two threads called `fflush`, the second caller would wait for the first to complete. But here, it's all coming from within the same thread; the second caller can't wait for the first caller to return, since the first caller can't run until the second caller returns!

(Note also that this signal handler potentially modifies `errno`, which can lead to "impossible" bugs in the main program.)

Win32 doesn't believe in interrupt user-mode code with other user-mode code asynchronously because it makes it impossible to reason about the state of the process. Delivering the console notification on a second thread means that if the second thread tries to access the heap while the first thread is inside the heap manager, the second thread will dutifully wait for the heap to stabilize before it goes ahead and starts mucking with it.

Raymond Chen

Follow