

Re: A new Critical Section for high contention situations

Source:

<http://www.tech-archive.net/Archive/Development/microsoft.public.win32.programmer.kernel/2008-05/msg00207.htm>

- *From:* "Kürsat" <xx@xxxxxx>
 - *Date:* Thu, 15 May 2008 19:56:15 +0300
-

Keith,

Thank you for your comments. I am not agree with you about your claims below :

You said "there is no synchroniztion", then, what does synchronization mean? AFAIK, synchronization means "accepting only one thread to access certain resource or functionality at a time". My object achive this, so, I think, there is synchronization.

You said "there is no contention", then what does contention mean? AFAIK, contention (in the current context) means "more than one ththread to attemp to access certain resource or functionality at the same time". While using my object, this condition is highly possible, so "there is contention".

You mentioned "race conditions" but, race conditions occurs on resources not on synchronization objects.

Please correct me if I am in the wrong. I don't claim that this short implementation is correct but your arguments are contradictory with my knowledge.

"Keith Moore" <keithmo@xxxxxxxxxxxx> wrote in message <news:eTMyk0ptIHA.4260@xxxxxxxxxxxxxxxxxxxxxxxxxxxx>

Kürpat wrote:

Hi,

"CRITICAL_SECTION"s are ideal inter-thread synchronization solutions for low-contention situations but as the contention increases the performance falls down accordingly because of kernel transition. So developed a simple synchronization object which always remains in user mode (at least I think so). I tested both the CRITICAL_SECTION and my object with a high contention schenario and I see my object gives ~%45 better results (with a dual core CPU). Wow, cool! What do you think about my implementation?

Re: A new Critical Section for high contention situations

Is there any obstacle to use it in a commercial software?

(I am not sure about originality of the implementation, it may be implemented many times by many developers.)

Thanks in advance.

////////////////////////////////////

```
#define OWNED 1
#define NOT_OWNED 0

class CRITICAL
{
private:
int m_nLock;

public:
CRITICAL () : m_nLock (NOT_OWNED) {}

void enter ()
{
if (InterlockedCompareExchange ((LONG *) &m_nLock, OWNED,
NOT_OWNED) == OWNED)
{
Sleep (0);
}
}

void leave ()
{
InterlockedExchange ((LONG *) &m_nLock, NOT_OWNED);
}
};
```

There's no contention with this lock because... uhh... there's no synchronization. If enter() is called on an already-locked object, the calling thread will call Sleep(0) *once* then return. Race conditions galore will ensue.

KM