Lock-free algorithms: Update if you can I'm feeling down

devblogs.microsoft.com/oldnewthing/20110413-00

April 13, 2011

Raymond Chen

A customer was looking for advice on this synchronization problem:

> We have a small amount of data that we need to share among multiple processes. One way to protect the data is to use a spin lock. However, that has potential for deadlock if the process which holds the spinlock doesn't get a chance to release it. For example, it might be suspended in the debugger, or somebody might decide to use `TerminateProcess` to nuke it.
>
> Any suggestions on how we can share this data without exposure to these types of horrible failure modes? I'm thinking of something like a reader takes the lock, fetches the values, and then checks at status at the end of tell if the data is valid. Meanwhile, a writer tries to take the lock with a timeout, and if the timeout fires, then the writer just goes ahead anyway and updates the values, and somehow sets the status on the reader so it knows that the value is no good and it should try again. Basically, I don't want either the reader or writer to get stuck indefinitely if a developer, say, just happens to break into the debugger at the worst possible time.

This can be solved with a publishing pattern. When you want to update the values, you indicate that new values are ready by publishing their new location.

Let's say that the data that needs to be shared is a collection of four integers.

```
struct SHAREDDATA {
 int a, b, c, d;
};
```

Assume that there is a practical limit on how often the value can change; this is usually a safe assumption because you'll have some sort of external rate limiter, like "This value changes in response to a user action." (Even if there is no such limit, most solutions will simply posit one. For example, the `SLIST` functions simply assume that a processor won't get locked out more than 65535 times in a row.) In our case, let's say that the value will not change more than 64 times in rapid succession.

```
#define SHAREDDATA_MAXCONCURRENT 64
SHAREDDATA g_rgsd[SHAREDDATA_MAXCONCURRENT];
UINT g_isd; // current valid value
void GetSharedData(__out SHAREDDATA *psd)
{
 *psd = g_rgsd[g_isd];
}
```

Reading the data simply retrieves the most recently published value. The hard part is publishing the value.

Actually, it's not hard at all.

```
LONG g_isdNext = 1;
void UpdateSharedData(__in const SHAREDDATA *psd)
{
 UINT isd = (UINT)InterlockedIncrementAcquire(&g_isdNext);
 isd %= SHAREDDATA_MAXCONCURRENT;
 g_rgsd[isd] = *psd;
 InterlockedExchange(&g_isdNext, isd);
}
```

Publishing the data is a simple matter of obtaining a slot for the data, using `InterlockedIncrement` to get a unique location to store the data, or at least least unique until `SHAREDDATA_MAXCONCURRENT - 1` intervening publications have occurred. We store our results into the memory we obtained and then publish the new index. The publication needs to be done with release semantics, but since there is no `InterlockedExchangeRelease`, we just do a full barrier exchange.

Note that the update is not atomic with the read. A processor can call `GetSharedData`, revise the values, then publish them, only to find that it overwrite a publication from another processor. If the new values are dependent on the old values (for example, if they are a running total), then you just lost an update.

Note also that if two threads try to update at the same time, it's pretty much random which set of values you get since it's *last writer wins*.

It so happens that in this particular case, the new values had nothing to do with the old values, so there was no problem with lost updates. And in practice, only one process updated the values at a time. (There is a master controller who updates the values, and everybody else just reads them.) Therefore, this simple method meets the requirements.

Exercise: How would you adapt this solution if the new values depended on the old values?

Raymond Chen

Follow