Semaphores don’t have owners

devblogs.microsoft.com/oldnewthing/20051123-14

November 23, 2005

Raymond Chen

Unlike mutexes and critical sections, semaphores don’t have owners. They merely have counts.

The `ReleaseSemaphore` function increases the count associated with a semaphore by the specified amount. (This increase might release waiting threads.) But the thread releasing the semaphore need not be the same one that claimed it originally. This is different from mutexes and critical sections, which require that the claiming thread also be the releasing one.

Some people use semaphores in a mutex-like manner: They create a semaphore with initial count 1 and use it like this:

```
WaitForSingleObject(hSemaphore, INFINITE);
… do stuff ..
ReleaseSemaphore(hSemaphore, 1, NULL);
```

If the thread exits (or crashes) before it manages to release the semaphore, the semaphore counter is not automatically restored. Compare mutexes, where the mutex is released if the owner thread terminates while holding it. For this pattern of usage, a mutex is therefore preferable.

A semaphore is useful if the conceptual ownership of a resource can cross threads.

```
WaitForSingleObject(hSemaphore, INFINITE);
… do some work ..
… continue on a background thread …
HANDLE hThread = CreateThread(NULL, 0, KeepWorking, …);
if (!hThread) {
  … abandon work …
  ReleaseSemaphore(hSemaphore, 1, NULL); // release resources
}


DWORD CALLBACK KeepWorking(void* lpParameter)
{
  … finish working …
  ReleaseSemaphore(hSemaphore, 1, NULL);
  return 0;
}
```

This trick doesn't work with a mutex or critical section because mutexes and critical sections have owners, and only the owner can release the mutex or critical section.

Note that if the `KeepWorking` function exits and forgets to release the semaphore, then the counter is not automatically restored. The operating system doesn't know that the semaphore "belongs to" that work item.

Another common usage pattern for a semaphore is the opposite of the resource-protection pattern: It's the resource-generation pattern. In this model the semaphore count normally is zero, but is incremented when there is work to be done.

```
… produce some work and add it to a work list …
ReleaseSemaphore(hSemaphore, 1, NULL);


// There can be more than one worker thread.
// Each time a work item is signalled, one thread will
// be chosen to process it.
DWORD CALLBACK ProcessWork(void* lpParameter)
{
  for (;;) {
    // wait for work to show up
    WaitForSingleObject(hSemaphore, INFINITE);
    … retrieve a work item from the work list …
    … perform the work …
  }
  // NOTREACHED
}
```

Notice that in this case, there is not even a conceptual "owner" of the semaphore, unless you count the work item itself (sitting on a work list data structure somewhere) as the owner. If the `ProcessWork` thread exits, you do **not** want the semaphore to be released automatically; that would mess up the accounting. A semaphore is an appropriate object in this case.

(A higher performance version of the producer/consumer semaphore is the I/O completion port.)

Armed with this information, see if you can answer this person's question.

[Raymond is currently away; this message was pre-recorded.]

Raymond Chen

Follow