Playing with synchroninzation barriers

devblogs.microsoft.com/oldnewthing/20151123-00

November 23, 2015

Raymond Chen

A *synchronization barrier* is a synchronization object that works like this:

- A synchronization barrier knows how many threads it is managing.
- Each thread that calls `EnterSynchronizationBarrier` blocks.
- When the last thread enters the synchronization barrier, all threads are released.
- Once a thread exits the synchronization barrier, it can re-enter it, at which point it blocks again, and the cycle repeats.

The idea here is that you have a multi-step process, and each thread must complete a step before any thread can go on to the next step. For example, you might have a sequence of steps like this:

- Everybody enters the room and sits down.
- Wait for everybody to be seated.
- Start the movie.
- Everybody watches the movie.
- After the movie ends, everybody leaves the room and stands outside.
- Wait for everybody to leave the room.
- One person locks the door.
- Everybody says good-bye and goes home.

The synchronization barrier takes care of the *wait for everybody to…* part.

It is not uncommon that an action needs to be taken after all threads have cleared the barrier, and the action needs to be performed exactly once. In the example above, one such action is "Start the movie." To support this pattern, the `EnterSynchronizationBarrier` returns `TRUE` to exactly one thread; all the other threads get `FALSE`.

Here's a Little Program that demonstrates the synchronization barrier pattern. Each thread is person who wants to watch the movie.

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>

#define WATCHERS 4

SYNCHRONIZATION_BARRIER barrier;
HANDLE movieFinished;

// Watchers gonna watch.
DWORD CALLBACK MovieWatcherThread(void* p)
{
  int id = PtrToInt(p);

  // Build a string that we use to prefix our messages.
  char tag[WATCHERS + 1];
  for (int i = 0; i < WATCHERS; i++) {
    tag[i] = (i == id) ? (char)('1' + i) : ' ';
  }
  tag[WATCHERS] = 0;

  printf("%s Sitting down\n", tag);

  if (EnterSynchronizationBarrier(&barrier, 0)) {
    // We are the one who should start the movie.
    printf("%s Starting the movie\n", tag);

    // For demonstration purposes, the movie is only one second long.
    LARGE_INTEGER dueTime;
    dueTime.QuadPart = -1000000LL;
    SetWaitableTimer(movieFinished, &dueTime, 0,
                     nullptr, nullptr, FALSE);
  }

  // Watch the movie until it ends.
  printf("%s Enjoying the movie\n", tag);
  WaitForSingleObject(movieFinished, INFINITE);

  // Now leave the room.
  printf("%s Leaving the room\n", tag);

  if (EnterSynchronizationBarrier(&barrier, 0)) {
    // We are the one who should lock the door.
    printf("%s Locking the door\n", tag);
  }

  printf("%s Saying good-bye and going home\n", tag);
  return 0;
}

int __cdecl main(int, char**)
{
```

```
  movieFinished = CreateWaitableTimer(nullptr, TRUE, nullptr);
  InitializeSynchronizationBarrier(&barrier, WATCHERS, -1);

  HANDLE threads[WATCHERS];
  for (int i = 0; i < WATCHERS; i++) {
    DWORD threadId;
    threads[i] = CreateThread(nullptr, 0, MovieWatcherThread,
                              IntToPtr(i), 0, &threadId);
  }

  // Wait for the demonstration to complete.
  WaitForMultipleObjects(WATCHERS, threads, TRUE, INFINITE);

  CloseHandle(movieFinished);
  DeleteSynchronizationBarrier(&barrier);
  return 0;
}
```

Each thread represents a person who wants to watch the movie. We sit down, and then wait for everybody else to sit down by entering the synchronization barrier. The call to `EnterSynchronizationBarrier` returns when everybody has sat down.

The watcher whose call to `EnterSynchronizationBarrier` returned `TRUE` takes responsibility for starting the movie.

And then everybody watches the movie, waiting until the movie ends.

Once that's done, each person leaves the room, and then enters another synchronization barrier to wait until everybody is outside.

The synchronization barrier will nominate one person to lock the door.

Everybody says good-bye and goes home.

There is one tricky thing about synchronization barriers: Knowing when it is safe to reenter a synchronization barrier after exiting it. If the synchronization barrier is always used by the same pool of threads (like we did here), then there's no problem. The thread clearly has exited the synchronization barrier if it manages to run code and try to enter it again.

The tricky part is if the collection of threads participating in the barrier changes over time. The rule is that when a thread exits the barrier, then it releases a "unit of synchronization" that another thread can consume by entering the barrier. Usually, the unit is consumed by the same thread that exits the barrier, but that thread could instead hand the unit to another thread and let that other thread take over the job. It is important that there be a causal link between the exiting thread and the future entering thread, so that the future entering thread knows that the previous exiting thread has fully exited.

Suppose there are three steps: Step 1 is handled by threads A and B, and Steps 2 and 3 are handled by threads B and C. You don't want to do this:

Thread A	Thread B	Thread C
Work on step 1	Work on step 1	Idle
EnterSynchronizationBarrier	EnterSynchronizationBarrier	
	Signal thread C to start	
	Work on step 2	Work on step 2
	EnterSynchronizationBarrier	EnterSynchronizationBarrier
	Work on step 3	Work on step 3

The problem here is that thread C may call `EnterSynchronizationBarrier` before thread A has exited it. There is no causal link between thread A exiting the barrier and thread C entering it, which creates the race condition.

In this case, you can solve the problem by having thread A be the one to signal thread C to start working on step 2. That way the "unit of synchronization" is positively handed from thread A to thread C.

Raymond Chen

Follow