Latches and Barriers

This lesson gives an overview of latches and barriers, predicted to be introduced in C++20.

Latches and barriers are simple thread synchronization mechanisms which enable some threads to wait until a counter becomes zero. At first, don't confuse the new barriers with memory barriers (also known as fences). In C++20, we will presumably get latches and barriers in three variations: `std::latch`, `std::barrier`, and `std::flex_barrier`.

First, there are two questions:

1. What are the differences between these three mechanisms to synchronize threads? You can use an `std::latch` only once, but you can use an `std::barrier` and an `std::flex_barrier` more than once. Additionally, an `std::flex_barrier` enables you to execute a function when the counter becomes zero.
2. What use cases do latches and barriers support that cannot be done in C++11 and C++14 with futures, threads, or condition variables in combination with locks? Latches and barriers address no new use cases, but they are a lot easier to use; they are also more performant because they often use a lock-free mechanism internally.

Now, I will have a closer look at these three coordination mechanisms.

`std::latch`

`std::latch` is a countdown counter; its value is set in the constructor. A thread can decrement the counter by using the method `thread.count_down_and_wait` and wait until the counter becomes zero. In addition, the method `thread.count_down` only decrements the counter by 1 without waiting. `std::latch` also has the method `thread.is_ready` that can be used to test if the counter is zero, and the method `thread.wait` to wait until the counter becomes zero. You cannot increment or reset the counter of a `std::latch`, hence you cannot reuse it.

Here is a short code snippet from the N4204 proposal:

```
void DoWork(threadpool* pool) {
  latch completion_latch(NTASKS);
  for (int i = 0; i < NTASKS; ++i) {
    pool->add_task([&] {
      // perform work
      ...
      completion_latch.count_down();
      }));
  }
  // Block until work is done
  completion_latch.wait();
}
```

I set the `std::latch completion_latch` in its constructor to `NTASKS` (line 2). The thread pool executes `NTASKS` (lines 4 - 8). At the end of each task (line 7), the counter will be decremented. Line 11 is the barrier for the thread running the function `DoWork` and, hence, for the small workflow. This thread has to wait until all tasks have been finished. In this case, an `std::barrier` is quite similar to an `std::latch`.

`std::barrier`

The subtle difference between `std::latch` and `std::barrier` is that you can use `std::barrier` more than once because the counter will be reset to its previous value. Immediately after the counter becomes zero, the so-called completion phase starts. `std::barrier` has an empty completion phase; this changes with `std::flex_barrier`. `std::barrier` has two interesting methods: `std::arrive_and_wait` and `std::arrive_and_drop`. While `std::arrive_and_wait` waits at the synchronization point, `std::arrive_and_drop` removes itself from the synchronization mechanism.

> **The proposal N4204**
>
> The proposal uses a `vector<thread*>` and pushes the dynamically allocated threads onto the vector `workers.push_back(new thread([&]{ ... }`; this is a memory leak. Instead, you should put the threads into a `std::unique_ptr` or create them directly in the vector: `workers.emplace_back[&]{ ...` . This observation holds for the example with `std::barrier` and `std::flex_barrier`. The names in the example with `std::flex_barrier` are a little bit confusing. For example, `std::flex_barrier` is called `notifying_barrier`, so I changed the name to `flex_barrier`. Additionally, the variable `n_threads` (representing the number of threads) was not initialized or was missing; I initialized it to `NTASKS`.

Before I take a closer look at `std::flex_barrier` and the completion phase in particular, I will give a short example demonstrating the usage of `std::barrier`.

```
void DoWork() {
  Tasks& tasks;
  int n_threads{NTASKS};
  vector<thread*> workers;

  barrier task_barrier(n_threads);

  for (int i = 0; i < n_threads; ++i) {
    workers.push_back(new thread([&] {
      bool active = true;
      while(active) {
        Task task = tasks.get();
        // perform task
        ...
        task_barrier.arrive_and_wait();
      }
    });
  }
  // Read each stage of the task until all stages are complete.
  while (!finished()) {
    GetNextStage(tasks);
```

```
      GetNextStage(tasks);
  }
}
```

The `barrier` in line 6 is used to coordinate a number of threads that perform their task multiple times; in this case, there are `n_threads` (line 3). Each thread takes its task at line 12 via `tasks.get()`, performs it and waits - once it is done with its task (line 15) - until all threads are done with their tasks. After that, it takes a new task in line 12 while active returns `true` in line 11. In contrast to `std::barrier`, `std::flex_barrier` has an additional constructor.

`std::flex_barrier`

This additional constructor can be parameterized by a *callable unit* that will be invoked in the completion phase. The callable has to return a number; this number sets the value of the counter in the completion phase. A return of -1 means that the counter keeps the same counter value in the next iteration. Numbers smaller than -1 are not allowed.

The completion phase performs the following steps:

1. All threads are blocked.
2. An arbitrary thread is unblocked and executes the callable unit.
3. If the completion phase is done, all threads will be unblocked.

The code snippet shows the usage of a `std::flex_barrier`.

```
void DoWork() {
  Tasks& tasks;
  int initial_threads;
  int n_threads{NTASKS};
  atomic<int> current_threads(initial_threads);
  vector<thread*> workers;

  // Create a flex_barrier, and set a lambda that will be
  // invoked every time the barrier counts down. If one or more
  // active threads have completed, reduce the number of threads.
  std::function rf = [&] { return current_threads;};
  flex_barrier task_barrier(n_threads, rf);

  for (int i = 0; i < n_threads; ++i) {
    workers.push_back(new thread([&] {
      bool active = true;
      while(active) {
        Task task = tasks.get();
        // perform task
        ...
```

```
        if (finished(task)) {
        current_threads--;
          active = false;

        }
        task_barrier.arrive_and_wait();
      }
    });
  }

  // Read each stage of the task until all stages are complete.
  while (!finished()) {
    GetNextStage(tasks);
  }
}
```

The example follows a similar strategy as the previous example with `std::barrier.` The difference is that this time the `std::flex_barrier` counter is adjusted at runtime; hence, the `std::flex_barrier task_barrier` in line 11 gets a lambda function. This lambda function captures its variable `current_thread` by reference: `[&] { return current_threads;}`. The variable will be decremented in line 21, and `active` will be set to `false` if the thread has completed its task. Therefore, the counter is decremented in the completion phase.

`std::flex_barrier` can increase the counter in contrast with `std::barrier` or `std::latch`. You can read further details of std::latch, std::barrier, and std::flex_barrier at cppreference.com.