Introduction

The following laboratory report describes the implementation of a computation manager using the producer-consumer pattern and the Hoare monitor synchronization construct following a boss-worker paradigm.

The requests may be of different types (held in an `enum`). In this case the types are A, B and C. A shared buffer is used to store calculation requests issued by the GUI. Groups of workers per type will then fetch these requests in the order of arrival and execute them. The results are then returned to the GUI, which displays them in the order of arrival.

Because this project uses a Hoare monitor, the `monitorIn()` and `monitorOut()` methods are always called at the beginning and the end of methods that work with shared data.

Data structures

This project uses an `EnumIndexedArray` (custom wrapper around `std::array`) to store the computation requests and the condition variables by computation type. This structure is indexed by the `ComputationType` enumeration, allowing for type-safe access to queues of computation requests. Each queue is related to a specific computation type (e.g., A, B, C), ensuring that requests are organized and retrievable based on their nature. This design facilitates efficient fetching of tasks by workers specialized in particular computation types and abstracts away the implementation details, namely the casts, necessary to use an `enum` as an index in an `std::array`.

The results are managed in an `std::deque`, a structure that supports efficient insertion and removal from both ends. Contrary to an `std::queue`, it supports random access to elements, which is necessary to remove items from the middle of the container. This structure was chosen for its ability to maintain the order of results (results are returned to the client in the same sequence as the requests were submitted) and for being efficient in terms of insertion and removal respectively at the beginning and end of the container.

Step 1 — `requestComputation` and `getWork`

Conception

The `requestComputation(Computation c)` method is designed to handle incoming computation requests from client threads. Each request has a unique ID assigned, which is incremented for each new request to maintain the order. The method uses `wait(Condition& cond)` on a condition variable when the buffer reaches its capacity, thus implementing a blocking behavior as required. When there is enough space in the buffer, the request is added to the queue and the method signals that the queue is not empty, potentially unblocking worker threads waiting for new tasks.

Worker threads use `getWork()` to fetch computation tasks. The method ensures exclusive access to the buffer and checks if there are available tasks of the requested type. If the queue is empty, it blocks the thread using `wait()` on a condition variable until new tasks are available. Once a task is fetched, the method signals that the queue is not full, potentially unblocking client threads waiting to submit new computations. This ensures a continuous flow of tasks to workers.

Tests

The following manual tests have been made and passed:

- The provided automated tests succeed.
- When an A work request is issued, the first type A worker handles it.
- When a B work request is issued, the type B worker handles it.
- When a C work request is issued, the type C worker handles it.
- When A, B and C work requests are issued, they are all handled by the corresponding worker simultaneously.
- When 5 A work request are issued, the 2 type A workers manages the first 2 requests, while the 3 others are waiting. Once the first worker finished the computation, it gets the next A request until none are waiting anymore.
- When any request is issued while its corresponding worker is busy, the requests waits for an available worker before being executed.

- When multiple work requests for any worker are issued, the requests id are set in order in the console.

Step 2 — `getNextResult` and `provideResult`

Conception

The `getNextResult` method is responsible for retrieving the next available result in the correct order. Upon entering the method, the monitor is locked to ensure exclusive access to the resultsQueue. It immediately checks if the system has been stopped, and if so, exits and throws a stop exception. A while loop is used to wait for an available result. This loop is crucial because it accounts for scenarios where results are not ready in order, necessitating a re-check whenever a thread is woken up. If the queue is empty or the next result in order is not yet available, the thread waits on the `resultAvailable` condition variable. Once a result is available and it's the next in order, it is retrieved from the back of the queue, and the queue is updated by removing this element. If there are more results available after retrieving one, the method signals the next waiting thread, ensuring a continuous flow of result processing.

`provideResult` allows computation threads to return completed results. The method searches the `resultsQueue` for the corresponding computation ID using `std::find_if`. Upon finding the matching entry, it updates the `std::optional<Result>` with the actual result. After updating the result, the method signals that a new result is available, potentially waking up a client thread waiting in `getNextResult`.

The implementation implicitly ensures result ordering by the manner in which results are stored and retrieved. Each result is associated with its computation ID, and `getNextResult` retrieves results based on this ordering. The use of `std::optional<Result>` allows for results to be stored in the queue before they are actually available, ensuring that the retrieval order matches the request order, regardless of the computation completion order. The same behavior may be achieved by using a list to store the results and then sorting it upon a request for result, although this method would be less efficient.

Tests

The following manual tests have been made and passed:

- The provided automated tests succeed.
- When an A work request is issued, the first available type A worker handles it and the result is returned.
- When a B work request is issued, the first available type B worker handles it and the result is returned.
- When a C work request is issued, the first available type C worker handles it and the result is returned.
- When A, B and C work requests are issued, they are all handled by the corresponding worker simultaneously and all results are returned in first-in first-out order once the last worker has finished.
- When 5 A work request are issued, the 2 type A workers manage the first 2 requests, while the 3 others are waiting. Once the first worker finished the computation, it gets the next A request until none are waiting anymore. The result are returned in the same order than the launched request id.
- When any requests are issued while its corresponding worker is busy, the requests wait. Each result is returned in order.
- When the sequence A-A-B-C is issued the first result returned is A, followed by the results A, B and C.
- When the sequence A-B-C-A is issued, first the results A, B and C are returned, Then the second A result.

Step 3 — `abortComputation` and `continueWork`

Conception

The `abortComputation` method is triggered by the user to cancel a specific computation. Upon reception of the request, the manager must find the computation to abort and remove if from either the requests or results queue, depending on whether the workload was already picked up by a worker thread or not. If the computation was in the requests queue, the manager must notify other functions waiting on a `notFull` condition that it may proceed and remove the result from the results queue. If the computation was already being worked on, it must only be removed from the results queue.

The `continueWork` method is periodically called by the workers to check whether the assigned workload has been aborted or not or if the program is stopped. If the program is stopped, the method must simply return false. Otherwise, a boolean indicating whether the computation has been aborted or not is returned. This is done by checking if the computation is still in the requests queue.

Tests

The following cases were tested and verified to work as expected:

- The provided automated tests succeed.
- When an ongoing computation is aborted, it stops on the next `continueWork()` call and the result is not displayed.
- When an unfinished computation is aborted, the result is not displayed.
- When the buffer for a given type is full, and an unfinished computation of the same type is aborted, another computation request of the same type may be added.
- When the first computation of a given type is aborted, the result of the second computation of the same type is displayed first.
- When the first computation of a given type is aborted and the second computation of the same type is already finished, the result of the second computation of the said type is immediately displayed.
- When a result for a computation is available but is waiting on a computation of another type to finish before being displayed, cancellation of the computation will not display the result.
- A cancellation request for an already cancelled workload is logged, but ignored.
- A cancellation request for an already finished and displayed workload is logged, but ignored.

Step 4 — `stop` and `throwStopException`

Conception

The `stop` method is called by the user to stop the program. It sets the `stopped` boolean to true and notifies all conditions (`notFull`, `notEmpty` and `resultAvailable`) that the program has been stopped. The program must be modified in the following ways:

1. On method entrance, the `stopped` flag must be checked and the method must return when it is set to true.
2. Any method which has a wait condition must check if the program has been stopped upon waking up, such as in listing 1.

If the program has been stopped, the method must signal eventual other waits of the same condition that the program has been stopped. This pattern is known as *cascading*. Once the last wait has been cascaded to, it releases the monitor and exits by throwing a `StopException` through the `throwStopException` method. The control flow then cascades back up the signaling chain, releasing monitors and throwing exceptions, until the `stop` method is reached and exited.

```
if (stopped) {
    signal(resultAvailable);
    monitorOut();
    throwStopException();
}
```

Listing 1: Excrept of the stop condition checking in the `ComputationManager::getNextResult()` method.

Tests

The following cases where tested and verified to work as expected:

- The provided automated tests succeed.
- When no request are present, the program stops immediately and the threads are joined.
- When there is work being done and a stop is requested, the program throws `StopException` and the threads are joined.
- When the program is stopped, the workloads end on the next `continueWork()` call.
- When the program is stopped, no more results are displayed.
- When the program is stopped and a request for a new computation is issued, the request is ignored.
- When the program is stopped and a cancellation request is issued, the cancellation is ignored.
- When there are workloads waiting in the buffer and the program is stopped, the workloads are not executed.

Conclusion

In this project, the main challenge was to understand the correct flow of the work requests and their dispatched results, as well as implementing it correctly with the producer-consumer pattern in a Hoare monitor. The implementation of the Hoare monitor was particularly complex due to the necessity of orchestrating the waiting points for the different conditions. Special attention had to be paid so that the conditions verify the state of the program once control is returned to them. These points had to be synchronized precisely with the signal mechanisms to ensure deadlock-free operation.