Coroutine

Arnav Gupta

October 26, 2024

Contents

1 Coroutine

A routine that can also be suspended at some point and resumed from that point when control returns.

State of a coroutine consists of:

- **execution location** starting at the beginning of the coroutine and remembered at each suspend
- **execution state** holding the data created by the code the coroutine is executing (coroutine stack)
- **execution status** (active/inactive/terminated) changes as control resumes and suspends in a coroutine

Coroutine is activated at the point of last suspension.

Couroutines allow for retaining state between calls (like a finite state machine) and execute synchronously with other coroutines.

2 Semi-Coroutine

Acts asymmetrically by implicitly reactivating the coroutine that previously activated it.

`_Coroutine` wraps coroutine and provides all class properties:

- distinguished member `main` can be suspended and resumed (usually private or protected)
- no parameters or return value (supplied by public members and communication variables)

All coroutines inherit from the base type `uBaseCoroutine`.

```
class uBaseCoroutine {
    protected:
        void resume(); // context switch to this
        void suspend(); // context switch to last resumer
        virtual void main() = 0; // starting routine for coroutine
    private:
        uBaseCoroutine();
        uBaseCoroutine( unsigned int stackSize ); // set stack size
        void verify(); // check stack
```

```
        const char * setName( const char * name ); // printed in error messages
        const char * getName() const;
        uBaseCoroutine & starter() const; // coroutine performing first resume
        uBaseCoroutine & resumer() const; // coroutine performing last resume
}
```

Resume/suspend causes a context switch between coroutine stacks:

- first `resume` starts main on new stack, subsequent resumes reactivate the last `suspend`
- `suspend` reactivates the last resumer
- routine frame at the top of the stack knows where to activate execution
- `suspend` and `resume` are protected members to prevent external calls to give more control over how these are called to the coroutine developer

When deleted, a coroutine's stack is always unwound and any destructors are executed.

Cannot use `catch( ... )` in a coroutine if it may be deleted before terminating, since a cleanup exception is raised to force stack unwinding.

`resume` in constructor allows coroutine main to get to first suspend.

2.0.1 Correct Coroutine Usage

Eliminate computation or flag variables retaining information about execution state. Instead, let the coroutine do that work.

To convert a direct program to a coroutine:

1. put processing code into coroutine main
2. convert reads if program is consuming or writes if program is producing to suspend
3. use interface members and communication variables to transfer data in and out of coroutine

3 μC++ EHM

Supports throw and resume for raising, and termination and resumption for handling.

Supports propagation of nonlocal and concurrent exceptions.

All exception types are grouped into a hierarchy.

4 Exception Type

μC++ restricts exception types to those defined by `_Exception`. All `_Exception` types must have public default and copy constructor.

5 Inherited Members

Each exception type inherits the following from `uBaseEvent`:

```
class uBaseEvent {
    uBaseEvent( const char *const msg = "" );
    const char *const message() const; // returns string message
    const uBaseCoroutine &source() const; // returns coroutine/task that raised excepti
    const char *sourceName() const; // returns name of coroutine/task that raised excep
    virtual void defaultTerminate(); // called if an exception is thrown but not handle
    virtual void defaultResume(); // called if an exception is resumed but not handled
}
```

`msg` is printed if the exception is not caught (string copied, so safe to use even if context of the raise is deleted).

Coroutine/task may be deleted when the exception is caught so `source` can return an undefined reference, but `sourceName` copies the name, so that will be defined.

`defaultTerminate` usually forwards an `UnhandledException` to the resumer/joiner. `defaultResume` usually throws the exception.

6 Raising

Can throw and resume with `_Throw` and `_Resume` respectively. If no exception type is passed, they are rethrows and reresumes respectively.

`_Resume` has an optional `_At` clause that allows the specified exception or currently propagating exception to be raised at another coroutine/task.

Only resumption allows nonlocal raise since raising execution state is often unaware of the handler's execution state. Resumption allows the faulting

execution choose if it can process the exception as resumption or rethrow the exception for termination.

In μC++, handlers can catch the specific derived exception type (rather than base).

7 Handler

7.1 Resumption

Using `_CatchResume` after a `try` body handle resumption, however these must precede `catch` handlers.

A resumption handler can access types and variables visible in its local scope.

Lexical link gives access to declaration block.

Resumption handler cannot perform a `break`, `continue`, `goto`, or `return`.

If correction is impossible from resumption, the handler should throw an exception rather than breaking to cause the stack to unwind correctly.

7.2 Termination/Resumption

Exception types for termination and resumption can overlap. That is, `catch` and `_CatchResume` can have the same exception type. Then, the termination handler is called if the resumption handler throws.

If no resumption handler exists, `defaultResume` is called which throws anyways.

7.3 Object Binding

`_Resume` and `_Throw` implicitly store the `this` associated with the member raising an exception.

For a static member or free routine, there is no binding. For a non-local raise, the binding is the coroutine/task executing the raise.

7.4 Bound Handlers

When a raising object is specified in a handler, an exception is caught when the bound and handler objects are equal and the raised exception equals the handler exception or its base type.

8 Nonlocal Exceptions

Exceptions raised by a source execution at a faulting execution, possible because each coroutine has its own stack.

Nonlocal resumption becomes a local resumption in the faulting coroutine. The source delivers a nonlocal exception immediately, but propagation only occurs when faulting becomes active, so that must be resumed.

The faulting coroutine checks for nonlocal exceptions around `_Enable`, `suspend`, and `resume`. The handler then returns to the implicit local raise.

Multiple nonlocal exceptions are queued and delivered in FIFO order depending on the current enabled exceptions.

Nonlocal delivery is initially disabled for a coroutine so this must be explicitly enabled with `_Enable`.

An unhandled exception in a coroutine raises a nonlocal exception of type `uBaseCoroutine::UnhandledException` at the coroutine's last resumer and then terminates the coroutine.

Exception forwarding can occur accross any number of coroutines until a task main forwards and then the program terminates by calling main's `set_terminate`.

The original exception is in the `UnhandledException` exception and can be thrown by `uh.triggerCause()`. If the original exception has a `defaultTerminate` routine, that will override `UnhandledException`.

While the coroutine terminates, control returns to its last resumer rather than its starter.

`UnhandledException` is always enabled.

9 Memory Management

Coroutine stacks expand to the next stack, rather than the heap.

Default μC++ coroutine stack size 256K and does not grow, but it can be adjusted through the coroutine constructor.

Sometimes better to allocate large arrays in heap when in coroutine main, due to stack size.

10 Full Coroutine

Acts symmetrically by explicitly activating a member of another coroutine which directly or indirectly activates the original coroutine (activation cycle).

A full coroutine can perform semi-coroutine operations since it subsumes a semi-coroutine.

Suspend inactivates the current active coroutine and activates the last resumer. Resume inactivates the current active coroutine and activates the current object (`this`). Hence, the current object must be a non-terminated coroutine.

The last resumer is not changed when resuming self.

Phases to any full coroutine program:

1. starting the cycle
2. executing the cycle
3. stopping the cycle (returning to program main)

Starting the cycle requires each coroutine to know at least one other coroutine, but this can cause mutually recursive references.

Note that a coroutine should go back to its starter. For full coroutines, the starter is often not the last resumer, so coroutine main does not appear to implicitly `suspend` on termination. The starter stack always gets back to program main.

11 Coroutine Languages

Coroutine implementations can be:

- **stackless**: use the caller's stack and fixed-size local state
  - cannot call other routines and then suspend
  - generators and iterators using `yield`
- **stackful**: separate stack and a fixed-size local state (from a class)