Doc No: X3J16/91-0130
WG21/N0063
Date: 4 November 1991
Reply to: Dag Brück

Comments on C++ Concurrency

Dag M. Brück

Department of Automatic Control
Lund Institute of Technology
Box 118, S-221 00 Lund, Sweden
dag@control.lth.se

This paper gives a brief description of μC++, an extended version of C++ with primitives for concurrent programming, and evaluates the Waterloo proposal's suitability for standardization. There is also a more general discussion of some fundamental requirements for concurrent programming in C++.

1. Summary

Some sort of standard C++ concurrency library is desirable, although there are several opinions of what it should look like. The preliminary proposal from University of Waterloo [Buhr *et al.*, 1991] describes an interesting system called μC++, and a revised version of μC++ is described in [Buhr and Stroobosscher, 1991]. There are two drawbacks of the proposal that I believe will prevent it from being accepted as a standard in its present form:

- The proposal relies on substantial extensions of the basic C++ language, both new syntax and new semantics. Most importantly, the changes will have an impact on all C++ programs, not only those that use concurrency. The increase in complexity also affects every C++ implementation.
- The proposal is written based on one particular view of concurrency. Although I think the proposal is reasonable, it will surely not correspond to every interpretation of the word concurrency. Because the proposal is based on language extensions and not only library extensions, there is no obvious way to substitute a different concurrency scheme.

The authors hope that their proposal can form the basis for further discussions of concurrency in C++. The analysis in the paper provides valuable input to X3J16: it defines three fundamental execution properties and shows how common concurrency mechanisms can be expressed as combination of the execution properties.

It is clear that better designs and implementations of some constructs can be provided if the language is carefully revised. Some problems that arise in concurrent programming lend themselves to general solutions that deserve closer study, e.g., assertion of class invariants. Other concurrency primitives cannot be implemented in an efficient and type-safe way without major extensions to C++.

2. Overview of μC++

μC++ provides support for several forms of concurrent programming through a combination of language extension and library extension. Synchronous communication between concurrent objects* is performed using function calls; asynchronous communication was deemed too complicated to be provided as a built-in feature. μC++ extends C++ with some new keywords:

uCoroutine	uSuspend	uWhen
uTask	uResume	uOr
uMutex		uAccept
uNoMutex		uWait
		uSignal

uCoroutine or **uTask** replace the keyword **class** to indicate a concurrent object. **uMutex** and **uNoMutex** are new type qualifiers that specify the presence or absence of mutual exclusion of public member functions. The other new keywords control synchronization. For example,

```
uMutex uCoroutine Foo {
public:
    Foo();
    void Set(int);
    uNoMutex void Get(int &);
};
```

defines a coroutine where public members are protected from multiple access, except **Get** which is explicitly declared to have no protection.

The current implementation translates each extension into one or more C++ statements. μC++ uses a single-memory model and executes on uniprocessors and multiprocessor shared-memory computers running the UNIX operating system. So far, μC++ has never been used in a stand-alone configuration for hard real-time applications.

In the current version, tasks do not have priorities to distinguish operations running at different time scales, and the authors believe that in future versions of μC++, two priority levels are sufficient in practical applications — certainly a controversial position. The two-level scheme will be implemented in future versions of μC++.

---

* *Concurrent object* is a deliberately loose term representing objects used for coroutines or autonomous tasks (processes). The proposal uses better terminology, which is too specific to use here.

3. Some fundamental requirements

I think the proposal is most valuable in that it identifies three dimensions of the design space for concurrent systems. Most (maybe all) mechanisms for concurrent programming can be identified as combinations of the following three execution properties:

Thread. Execution that occurs independently of other execution. Conceptually this is a virtual processor whose function is to advance execution by changing execution state.

Execution state. The state information needed to permit concurrent execution. An execution state is either active or passive, depending on whether or not it is currently being executed by a thread.

Mutual exclusion. An action on a resource that takes place without interruption by other actions on the resource. In concurrent systems, mutual exclusion is needed to guarantee consistent generation of results. □

Common mechanisms for concurrent programming can then be expressed as combinations of the fundamental execution properties:

Mechanism	Thread	Execution state	Mutual exclusion
Class object	no	no	no
Monitor	no	no	yes
Coroutine	no	yes	no
Coroutine-monitor	no	yes	yes
Task	yes	yes	yes

Other combinations are either contradictory (e.g., thread without execution state), or essentially useless (task without mutual exclusion).

I believe the identification of these three execution properties will be useful for discussing concurrency in C++. It will enable analysis of capability, flexibility and complexity of concurrency proposals.

Threads

The notion of threads is intimately associated with scheduling, i.e., some sort of algorithm that decides which thread should be executed next. Scheduling is one of the main characterizations of a real-time system, and many design issues depend on the chosen scheduler. Different forms of scheduling are needed depending on the application, and flexible implementations allow user-defined schedulers.

Consequently, I do not think scheduling should be standardized by a language committee. Given a standardized coroutine facility (see below), different forms of scheduling can be implemented in a portable way. The portable implementation may be inefficient, but efficient specialized implementations are also possible.

Execution state

The notion of execution state is not difficult to describe with an abstract base class, although implementations will of course be tailored to the architecture. The definition must prescribe some mechanism for context switching, such as, **uSuspend**

and **uResume** in μC++ or **Transfer** in Modula-2. Coroutine-style programming would then be possible in C++.

Execution state can be defined as a pure library extension and need not affect the language definition. Particular implementations may handle context switching more efficiently, just like some C/C++ implementations implement **strcmp** and **memcpy** by inline code.

Mutual exclusion

A fundamental problem in implementing a monitor, or some other protection mechanism, is to *guarantee* that the resource is protected. In current C++ real-time systems the programmer must explicitly use the correct mechanism, for example, wait/signal on a semaphore or a lock object. It is possible to provide functions that enforce mutual exclusion in a base class, but it is not possible to guarantee that they will be called.

Mutual exclusion can be regarded as a special case of class invariant. A general mechanism for asserting class invariants would most likely be able to handle mutual exclusion.

The predecessor to C++, C with Classes [Stroustrup, 1982], had such a mechanism. Given two classes written in an extended C++,

```
class Monitor {
    call Monitor();        // Lock protected region
    return Monitor();      // Free protected region
};

class Mailbox : public Monitor {
    void Send(int);
};
```

a call to **Mailbox::Send()** would implicitly invoke the inherited member functions that guarantee mutual exclusion:

```
void Mailbox::Send()
{
    Monitor::call Monitor();
    ....
    Monitor::return Monitor();
}
```

Extending the language with proper call/return functions will probably not affect existing C++ programs and implementations, and will not introduce any cost when not used. An extended version of C++ that provides similar features (and much more) is described in [Seliger, 1990]. An alternative approach is to provide a tool that inserts lock objects in the appropriate places (wherever that is).

Some concurrency primitives are inherently difficult, or impossible, to implement without major language extensions. For example, the **uAccept** statement of μC++ (similar to Ada's rendez-vous) provides efficient type-safe communication with selection from multiple queues.

Two-phase construction

Another problem we have encountered is two-phase construction, i.e., the need to let the constructor of a base class do some additional initialization after the object is constructed. In the following example we want to make a process eligible for scheduling when it has been completely constructed.

```
class Process {
    char* stack;
protected:
    Process(int stacksize);
    virtual void Main() = 0;
};

extern void Schedule(Process *p);

Process::Process(int stacksize)
    : stack(new char[stacksize])
{
    Schedule(this);
}
```

The problem is that the object is not completely constructed when the process is made ready to run. In a typical implementation, the "vtbl" does not contain a valid entry for the virtual **Main** function. In the destructor case, the derived part of the process object has been destroyed before the base class destructor suspends the process.

These problems can of course be "solved" by the application programmer; the correct calls to schedule the process must be inserted in the constructors of the derived classes, which means that all derived classes need intimate knowledge of the process implementation. It is essential that only leaf classes call **Schedule**, so complete knowledge of the inheritance hierarchy is also needed.

This is one of the most important difficulties when realizing concurrency with a library based approach; μC++ uses some sort of magic to handle two-phase construction. One possible solution for C++ is to re-invent the **inner** concept of Simula-67 [Birtwistle *et al.*, 1973]:

```
Process::Process(int stacksize)
    : stack(new char[stacksize])
{
    inner;
    Schedule(this);
}
```

Construction of the base class is done in two phases. Firstly, the base class constructor is run up to the **inner** statement. Secondly, derived classes are constructed. Thirdly, the rest of the base class constructor is run. In this example, the process is not scheduled until construction (process initialization) is completed. The main advantage is that the desired behaviour can be guaranteed by the base class implementor. There are, of course, several fine points that are not discussed here.

Low-level requirements

We need some simple mechanism for implementing the mutual exclusion primitive. A C++ equivalent to the test-and-set or test-and-add instructions found on many processors would be useful. This is also a library extension and will not affect the language definition.

Most forms of concurrent programming require re-entrancy. We cannot require that every standard library function be re-entrant, but the C++ standard should enable the programmer to avoid functions that are not re-entrant. I suggest that the standard will require every implementation to list all functions that are not re-entrant.

References

BIRTWISTLE, G. M., O.-J. DAHL, B. MYHRHAUG, and K. NYGAARD (1973): *SIMULA BEGIN.* Auerbach Publishers Inc., Philadelphia, PA, USA. Also published by Studentlitteratur.

BUHR, P. A., G. DITCHFIELD, R. A. STROOBOSSCHER, B. M. YOUNGER, and C. R. ZARNKE (1991): "μC++: Concurrency in the object-oriented language C++." *Software — Practice and Experience.* ANSI document X3J16/91-0133, ISO document WG21/N0066.

BUHR, P. A. and R. A. STROOBOSSCHER (1991): *μC++ Annotated Reference Manual, Version 3.0.* Department of Computer Science, University of Waterloo. Preliminary Draft.

SELIGER, R. (1990): "Extending C++ to support remote procedure call, concurrency, exception handling, and garbage collection." In *Proc. USENIX C++ Conference,* San Francisco, CA, USA. USENIX Association. April 9–11, 1990.

STROUSTRUP, B. (1982): "Classes: an abstract type facility for the C language." *SIGPLAN Notices,* 17:1, pp. 42–51.