μC++ Annotated Reference Manual

Version 3.7

Peter A. Buhr and Richard A. Stroobosscher

June 9, 1993

Contents

Preface

The goal of this work is to introduce concurrency into the object-oriented language C++ [Str91]. To achieve this goal a set of important programming language abstractions were adapted to C++, producing a new dialect called μC++. These abstractions were derived from a set of design requirements and combinations of elementary execution properties, different combinations of which categorized existing programming language abstractions and suggested new ones. The set of important abstractions contains those needed to express concurrency, as well as some that are not directly related to concurrency. Therefore, while the focus of this work is on concurrency, all the abstractions produced from the elementary properties are discussed. While the abstractions have been as extensions to C++, the requirements and elementary properties are generally applicable to other object-oriented languages such as Eiffel [Mey88], Simula [Sta87] and Smalltalk [GR83].

This manual does not discuss how to use the new constructs to build complex concurrent systems. The manual is strictly a reference manual for μC++. A reader should have an intermediate knowledge of control flow and concurrency issues to understand the ideas presented in this manual as well as some experience programming in C++.

This manual contains annotations set off from the normal discussion in the following way:

□ Annotation discussion like this is quoted with quads. □

An annotation provides rationale for design decisions or additional implementation information. Also a chapter or section may end with a commentary section, which contains major discussion about design alternatives and/or implementation issues. Since this organizational structure is taken from the Ellis and Stroustrup annotated C++ book [ES90], we hope we will not be sued for look and read violations.

Each chapter of the manual does *not* begin with an insightful quotation. Feel free to add your own.

Abridged Manual

This manual has an abridged form that removes the multiprocessor and UNIX I/O material. The abridged manual is useful for introductory teaching of μC++. To generate the abridged manual change the following line, which appears at the beginning of the source file for this manual, and reformat the manual.

```
\notabridgedtrue                % change true to false for abridged manual and reformat
```

Chapter 1

μC++ Extensions

μC++ extends the C++ programming language [Str91] in somewhat the same way that C++ extends the C programming language. The extensions introduce new objects that augment the existing panoply of control flow facilities and provide for light-weight concurrency on uniprocessor and parallel execution on multiprocessor computers running the UNIX[1] operating system. The following discussion is the rational for the particular extensions that were chosen.

1.1 Design Requirements

The following requirements directed this work:

- All communication among the new kinds of objects must be statically type checkable. We believe that static type checking is essential for early detection of errors and efficient code generation. (As well, this requirement is consistent with the fact that C++ is a statically typed programming language.)
- Interaction among the different kinds of objects should be possible, and in particular, interaction among concurrent objects, called tasks, should be possible. This allows a programmer to choose the kind of object best suited to the particular problem without having to cope with communication restrictions.

  This is in contrast to schemes where some objects, such as tasks, can only interact indirectly through another non-task object. For example, many programming languages that support monitors [Bri75, MMS79, Hol92] require that all communication among tasks be done indirectly through a monitor; similarly, the Linda system [CG89] requires that all communication take place through one or possibly a small number of tuple spaces. This increases the number of objects in the system; more objects consume more system resources, which slows the system. As well, communication among tasks is slowed because of additional synchronization and data transfers with the intermediate object.
- All communication between objects is performed using routine calls; data is transmitted by passing arguments to parameters and results are returned as the value of the routine call. We believe it is confusing to have additional forms of communication in a language, such as message passing, message queues, or communication ports.
- Any of the new kinds of objects should have the same declaration scopes and lifetimes as existing objects. That is, any object can be declared at program startup, during routine and block activation, and on demand during execution, using a `new` operator.
- All mutual exclusion must be implicit in the programming language constructs and all synchronization should be limited in scope. It is our experience that requiring users to build mutual exclusion out of synchronization mechanisms, e.g. locks, often leads to incorrect programs. Further, we have noted that reducing the scope in which synchronization can be used, by encapsulating it as part of programming language constructs, further reduces errors in concurrent programs.

[1] UNIX is a registered trademark of AT&T Bell Laboratories

- Both synchronous and asynchronous communication are needed in a concurrent system. However, we believe that the best way to support this is to provide synchronous communication as the fundamental mechanism; asynchronous mechanisms, such as buffering or futures [Hal85], can then be built when that is appropriate. Building synchronous communication out of asynchronous mechanisms requires a protocol for the caller to subsequently detect completion. This is error prone because the caller may not obey the protocol (e.g. never retrieve a result). Further, asynchronous requests require the creation of implicit queues of outstanding requests, each of which must contain a copy of the arguments of the request. This creates a storage management problem because different requests require different amounts of storage in the queue. We believe asynchronous communication is too complicated a mechanism to be hidden in a system.
- An object that is accessed concurrently must have some control over which requester it services next. There are two distinct approaches: control can be based on the kind of request, for example, selecting a requester from the set formed by calls to a particular entry point; or control can be based on the identity of the requester. In the former case, it must be possible to give priorities to the sets of requesters. This is essential for high priority requests, such as a time out or a termination request. (This is to be differentiated from giving priority to elements within a set or execution priority.) In the latter case, selection control is very precise as the next request must only come from the specified requester. Currently, we see a need for only the former case because we believe that the need for the latter case is rare.
- There must be flexibility in the order that requests are completed. This means that a task can accept a request and subsequently postpone it for an unspecified time, while continuing to accept new requests. Without this ability, certain kinds of concurrency problems are quite difficult to implement, e.g. disk scheduling, and the amount of concurrency is inhibited as tasks are needlessly blocked [Gen81].

We have satisfied all of these requirements in μC++.

1.2 Elementary Execution Properties

Extensions to the object concept were developed based on the following execution properties:

thread – is execution of code that occurs independently of and possibly concurrently with other execution; the execution resulting from a thread is sequential. A thread's function is to advance execution by changing execution state. Multiple threads provide concurrent execution. A programming language must provide constructs that permit the creation of new threads and specify how threads are used to accomplish computation. Further, there must be programming language constructs whose execution causes threads to block and subsequently be made ready for execution. A thread is either blocked or running or ready. A thread is **blocked** when it is waiting for some event to occur. A thread is **running** when it is executing on an actual processor. A thread is **ready** when it is eligible for execution but not being executed.

execution-state – is the state information needed to permit concurrent execution. An execution-state is either **active** or **inactive**, depending on whether or not it is currently being used by a thread. In practice, an execution-state consists of the data items created by an object, including its local data, local block and routine activations, and a current execution location, which is initialized to a starting point. The local block and routine activations are often maintained in a contiguous stack, which constitutes the bulk of an execution-state and is dynamic in size, and is the area where the local variables and execution location are preserved when an execution-state is inactive. A programming language determines what constitutes an execution-state, and therefore, execution-state is an elementary property of the semantics of a language. (An execution-state *starts* a new continuation [RC86] and is related to the notion of a process continuation [HD90].) When a thread transfers from one execution-state to another, it is called a **context switch**.

mutual exclusion – is the mechanism that permits an action to be performed on a resource without interruption by other actions on the resource. In a concurrent system, mutual exclusion is required to guarantee consistent generation of results, and cannot be trivially or efficiently implemented without appropriate programming language constructs.

The first two properties represent the minimum needed to perform execution, and seem to be fundamental in that they are not expressible in machine-independent or language-independent ways. For example, creating a new thread requires creation of system runtime control information, and manipulation of execution-states requires machine specific operations (modifying stack and frame pointers). The last property, while expressible in terms of simple language statements, can only be done by algorithms that are error-prone and inefficient, e.g. Dekker-like algorithms, and therefore we believe that mutual exclusion must also be provided as an elementary execution property.

A programming language designer could attempt to provide these 3 execution properties as basic abstractions in a programming language [BLL88], allowing users to construct higher-level constructs from them. However, some combinations might be inappropriate or potentially dangerous. Therefore, all combinations are examined, analyzing which ones make sense and are appropriate as higher-level programming language constructs. What is interesting is that enumerating all combination of these elementary execution properties produces many existing high-level abstractions and suggests new ones that we believe merit further examination.

The three execution properties are properties of objects. Therefore, an object may or may not have a thread, may or may not have an execution-state, and may or may not have mutual exclusion. Different combinations of these three properties produce different kinds of objects. If an object has mutual exclusion, this means that execution of certain member routines are mutually exclusive of one another. Such a member routine is called a **mutex member**. In the situation where an object does not have the minimum properties required for execution, i.e. thread and execution-state, those of its user (caller) are used.

Table 1.1 shows the different abstractions possible when an object possesses different execution properties. Case 1 is an object, such as a **free routine** (a routine not a member of an object) or an object with member

object properties		object's member routine properties	
thread	execution-state	no implicit mutual exclusion	implicit mutual exclusion
no	no	1 class-object	2 monitor
no	yes	3 coroutine	4 coroutine-monitor
yes	no	5 (rejected)	6 (rejected)
yes	yes	7 (rejected)	8 task

Table 1.1: Fundamental Abstractions

routines that have none of the execution properties, called a **class-object**. In this case, the caller's thread and execution-state are used to perform the execution. Since this kind of object provides no mutual exclusion, it is normally accessed only by a single thread. If such an object is accessed by several threads, explicit locking may be required, which violates a design requirement. Case 2 is like Case 1 but deals with the concurrent access problem by implicitly ensuring mutual exclusion for the duration of each computation by a member routine. This abstraction is a **monitor** [Hoa74]. Case 3 is an object that has its own execution-state but no thread. Such an object uses its caller's thread to advance its own execution-state and usually, but not always, returns the thread back to the caller. This abstraction is a **coroutine** [Mar80]. Case 4 is like Case 3 but deals with the concurrent access problem by implicitly ensuring mutual exclusion; we have adopted the name **coroutine-monitor** for it. Cases 5 and 6 are objects with a thread but no execution-state. Both cases are rejected because the thread cannot be used to provide additional concurrency. First, the object's thread cannot execute on its own since it does not have an execution-state, so it cannot perform any independent actions. Second, if the caller's execution-state is used, assuming the caller's thread can be blocked to ensure mutual exclusion of the execution-state, the effect is to have two threads successively executing portions of a single computation, which does not seem useful. Case 7 is an object that has its own thread and execution-state. Because it has both a thread and execution-state it is capable of executing on its own; however, it lacks mutual exclusion. Without mutual exclusion, access to the object's data is unsafe; therefore, servicing of requests would, in general, require explicit locking, which violates a design requirement. Further, there is no performance advantage over case 8. For these reasons, we have rejected this case. Case 8 is like Case 7 but deals with the concurrent access problem by implicitly ensuring mutual exclusion, called a **task**.

The abstractions suggested by this categorization come from fundamental properties of execution and not ad hoc decisions of a programming language designer. While it is possible to simplify the programming language design by only supporting the task abstraction [SBG+90], which provides all the elementary execution properties, this would unnecessarily complicate and make inefficient solutions to certain problems. As will be shown, each of the non-rejected abstractions produced by this categorization has a particular set of problems that it can solve, and therefore, each has a place in the programming language. If one of these abstractions is not present, a programmer may be forced to contrive a solution for some problems that violates abstraction or is inefficient.

Chapter 2

μC++ Translator

The μC++ translator reads a program containing language extensions and transforms each extension into one or more C++ statements, which are then compiled by an appropriate C++ compiler and linked with a concurrency runtime library. Because μC++ is only a translator and not a compiler, some restrictions apply that would be unnecessary if the extensions were part of the C++ programming language. Similar, but less extensive translators have been built: MC [RH87] and Concurrent C++ [GR88].

2.1 Extending C++

Operations in μC++ are expressed explicitly, i.e. the abstractions derived from the elementary properties are used to structure a program into a set of objects that interact, possibly concurrently, to complete a computation. This is to be distinguished from implicit schemes such as those that attempt to *discover* concurrency in an otherwise sequential program, for example, by parallelizing loops and access to data structures. While both schemes are complementary, and hence, can appear together in a single programming language, we believe that implicit schemes are limited in their capacity to *discover* concurrency, and therefore, the explicit scheme is essential. Currently, μC++ only supports the explicit approach, but nothing in its design precludes implicit approaches.

The abstractions in Table 1.1 are expressed in μC++ using two new type specifiers, uCoroutine and uTask, which are extensions of the class construct, and hence, define new types. In this manual, the types defined by the class construct and the new constructs are called **class types**, **monitor types**, **coroutine types**, **coroutine-monitor types** and **task types**, respectively. The terms **class-object**, **monitor**, **coroutine**, **coroutine-monitor** and **task** refer to the objects created from such types. The term **object** is the generic term for any instance created from any type. All objects can be declared externally, in a block, or using the new operator. Two new type qualifiers, uMutex and uNoMutex, are also introduced to specify the presence or absence of mutual exclusion on the member routines of a type (see Table 2.1). The default qualification values have been chosen based on the expected frequency of use of the new types. Several new statements are added to the language: uSuspend, uResume, uAccept, uWait and uSignal. Each is used to affect control in objects created by the new types. (The prefix "u" followed by a capital letter for the new keywords is an attempt to avoid current and future conflicts with UNIX routine names, e.g. accept, wait, signal, and C++ library names, e.g. task.) Appendix A shows the grammar for the μC++ extensions.

μC++ executes on uniprocessor and multiprocessor shared-memory computers. On a uniprocessor, concurrency is achieved by interleaving execution to give the appearance of parallel execution. On a multiprocessor computer, concurrency is accomplished by a combination of interleaved execution and true parallel execution. Further, μC++ uses a **single-memory model**. This single memory may be the address space of a single UNIX process or a memory shared among a set of UNIX processes. A memory is populated by routine activations, class-objects, coroutines, monitors, coroutine-monitors and concurrently executing tasks, all of which have the same addressing scheme for accessing the memory. Because these entities use the same memory they can be **light-weight**, so there is a low execution cost for creating, maintaining and communicating among them. This has its advantages as well as its disadvantages. Communicating objects do not have to

object properties		object's member routine properties	
thread	execution-state	no implicit mutual exclusion	implicit mutual exclusion
no	no	[uNoMutex]† class	uMutex class
no	yes	[uNoMutex] uCoroutine	uMutex uCoroutine
yes	yes	N/A	[uMutex] uTask

† [] implies default qualification if not specified

Table 2.1: New Type Specifiers

send large data structures back and forth, but can simply pass pointers to data structures. However, this technique does not lend itself to a distributed environment with separate address-spaces.

□ Currently, we are looking at the approaches taken by distributed shared-memory systems to see if they provide the necessary implementation mechanisms to make the non-shared memory case similar to the shared-memory case. □

Commentary

μC++ tasks are not implemented as UNIX processes for two reasons. First, UNIX processes have a high runtime cost for creation and context switching. Second, each UNIX process is allocated as a separate address space (or perhaps several) and if the system does not allow memory sharing among address spaces, tasks would have to communicate using pipes and sockets. Pipes and sockets are runtime expensive. If shared memory is available, there is still the overhead of entering the UNIX kernel, page table creation, and management of the address space of each process. Therefore, UNIX processes are called **heavy-weight** because of the high runtime cost and space overhead in creating a separate address space for a process, and the possible restrictions on the forms of communication among them. μC++ provides access to UNIX processes only indirectly through virtual processors (see Section 2.3.2). A user is not prohibited from creating UNIX processes explicitly, but such processes are not administrated by the μC++ runtime environment.

2.2 Compile Time Structure of a μC++ Program

A μC++ program is constructed exactly like a normal C++ program with one exception: the main (starting) routine is a member of an initial task called uMain, which has the following structure (Section 2.11 details the task construct):

```
uTask uMain {
  private:
    int argc;                       // number of arguments on the shell command line
    char *argv[];                   // pointers to tokens on the shell command line
    char *envp[];                   // pointers to shell environment variables
    int &uResult;                   // return value to the shell
    void main();                    // user provides body for this routine
  public:
    uMain( int argc, char *argv[], char *envp[] ) {
        uMain::argc = argc;
        uMain::argv = argv;
        uMain::envp = envp;
    }
};
```

A μC++ program must define the body for the main member routine of this initial task, as in:

```
... // normal C++ declarations and routines

void uMain::main() {                    // body for initial task uMain
    ...
}
```

μC++ supplies and uses the free routine main to initialize the μC++ runtime environment and creates the task uMain of which routine uMain::main is a member. Member uMain::main has available as local variables the same three arguments that are passed to the free routine main: argc, argv, and envp. To return a value back to the shell, set the variable uResult and return from uMain::main; uResult is initialized to zero.

2.3 Runtime Structure of a μC++ Program

The dynamic structure of an executing μC++ program is significantly more complex than a normal C++ program. In addition to the five kinds of objects introduced by the elementary properties, μC++ has two more runtime entities that are used to control the amount of concurrent execution.

2.3.1 Cluster

A cluster is a collection of tasks and virtual processors (discussed next) that execute those tasks. The purpose of a cluster is to control the amount of parallelism that is possible among tasks, where **parallelism** is defined as execution which occurs simultaneously. This can only occur when multiple processors are present. **Concurrency** is execution that, over a period of time, appears to be parallel. For example, a program written with multiple tasks has the potential to take advantage of parallelism but it can execute on a uniprocessor, where it may *appear* to execute in parallel because of the rapid speed of context switching.

A cluster uses a single-queue multi-server queueing model for scheduling its tasks on its processors. This results in automatic load balancing of tasks on processors. Figure 2.1 illustrates the runtime structure of a μC++ program. An executing task is illustrated by its containment in a processor. Because of appropriate defaults for clusters, it is possible to begin writing μC++ programs after learning about coroutines or tasks. More complex concurrency work may require the use of clusters. If several clusters exist, tasks can be explicitly migrated from one cluster to another. No automatic load balancing among clusters is performed by μC++.

When a μC++ program begins execution, it creates two clusters: a system cluster and a user cluster. The system cluster contains a processor that does not execute user tasks. Instead, the system cluster handles system related operations, such as, catching errors that occur on the user clusters, printing appropriate error information, and shutting down μC++. A user cluster is created to contain the user tasks; the first task created in the user cluster is uMain, which begins executing the member routine uMain::main. Having all tasks execute on the one cluster often maximizes utilization of processors, which minimizes runtime. However, because of limitations of the underlying operating system or because of special hardware requirements, it is sometimes necessary to have more than one cluster. Partitioning into clusters must be used with care as it has the potential to inhibit parallelism when used indiscriminately. However, in some situations it will be shown that partitioning is essential. For example, on some systems concurrent UNIX I/O operations are only possible by exploiting the clustering mechanism.

2.3.2 Virtual Processor

A μC++ virtual processor is a "software processor" that executes threads. A virtual processor is implemented as a UNIX process (or kernel thread) that is subsequently scheduled for execution on a hardware processor by the underlying operating system. On a multiprocessor, UNIX processes are usually distributed across the hardware processors and so some UNIX processes are able to execute in parallel. This, in turn, means the tasks executing on them execute in parallel. μC++ uses virtual processors instead of hardware processors so that programs do not actually allocate and hold hardware processors. Programs can be written to run using a number of virtual processors and execute on a machine with a smaller number of hardware processors. Thus, the way in which μC++ accesses the parallelism of the underlying hardware is through an intermediate

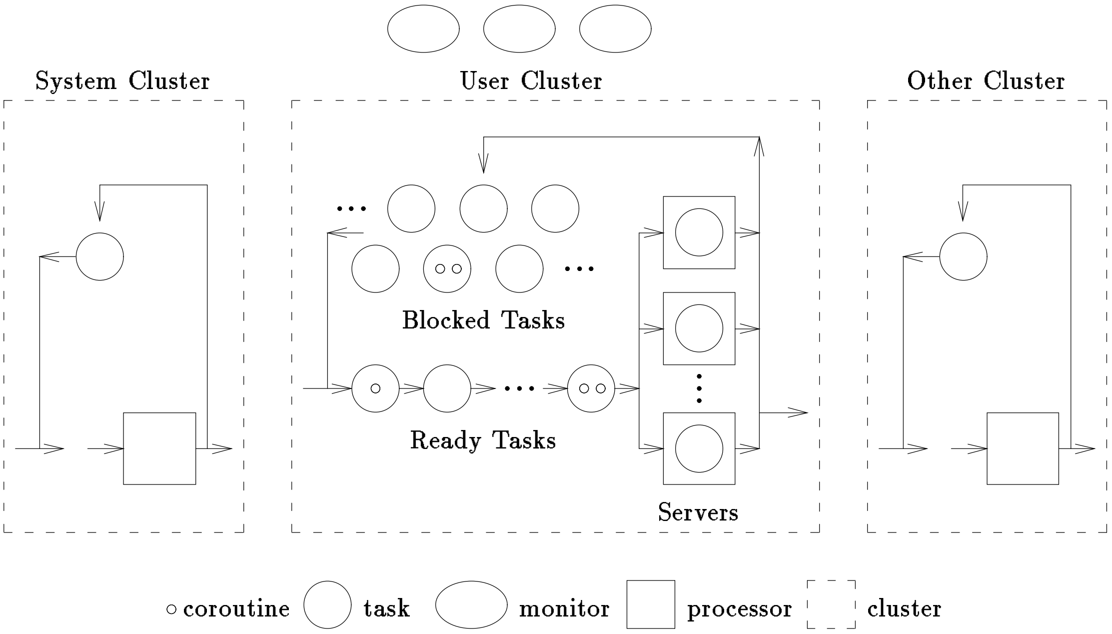

Figure 2.1: Runtime Structure of a μC++ Program

resource, the UNIX process (or kernel thread). In this way, μC++ is kept portable across uniprocessor and different multiprocessor hardware designs.

When a virtual processor is executing, μC++ controls scheduling of tasks on it. Thus, when UNIX schedules a virtual processor for a runtime period, μC++ may further subdivide that period by executing one or more tasks. When multiple virtual processors are used to execute tasks, the μC++ scheduling may automatically distribute tasks among virtual processors and, thus, indirectly among hardware processors. In this way, parallel execution occurs.

2.4 μC++ Kernel

After a μC++ program has been translated and compiled, a runtime concurrency library is linked in with the resulting program, called the μC++ kernel. There are two versions of the μC++ kernel: the unikernel, which is designed to use a single processor (the system, user and any other clusters are automatically combined); and the multikernel, which is designed to use several processors. Thus, the unikernel is sensibly used on systems with a single hardware processor or nonshared memory; the multikernel is sensibly used on systems that have multiple hardware processors and that permit memory to be shared among UNIX processes. Table 2.2 shows the situations where each kernel can be used. The unikernel can be used in a system with multiple hardware processors and shared memory but does not take advantage of either of these capabilities. The multikernel can be used on a system with a single hardware processor and shared memory but performs less efficiently than the unikernel because it uses multi-processor mutual exclusion unnecessarily.

The μC++ kernel has a debugging form, which performs a number of runtime checks. For example, the μC++ kernel provides no support for automatic growth of stack space for coroutines and tasks because this would require compiler support. The debugging kernel checks for stack overflow whenever context switches occur among coroutines and tasks. This catches most stack overflows; however, stack overflow can still occur if insufficient stack area is provided, which can cause an immediate error or unexplainable results. Many other runtime checks are performed in the debugging kernel.

	non-shared memory among UNIX processes	shared memory among UNIX processes
single processor	unikernel, yes multikernel, no	unikernel, yes multikernel, yes, but inefficient
multiple processors	unikernel, yes multikernel, no	unikernel, yes, but no parallelism multikernel, yes

Table 2.2: When to Use the Unikernel and Multikernel

2.5 Using the μC++ Translator

To use the concurrency extensions in a C++ program, include the file:

```
#include <uC++.h>
```

at the beginning of each source file.

2.5.1 Compiling a μC++ Program

The command u++ is used to compile μC++ program(s). This command works just like the AT&T CC or GNU g++ command to compile C++ programs (the default C++ compiler is GNU C++ [Tie90]), for example:

u++ *[C++ options] yourprogram.cc [assembler and loader files]*

The following additional options are available on the u++ command:

-debug The program is linked with the debug version of the unikernel or multikernel. The debug version performs runtime checks to help during the debugging phase of a μC++ program. This slows the execution of the program. The runtime checks should only be removed after the program is completely debugged. **This is the default.**

-nodebug The program is linked with the non-debug version of the unikernel or multikernel. **No runtime checks are performed so errors usually result in abnormal program termination.**

-delay When the program is translated, a random number of context switches occur at the beginning of each member routine so that during execution there is a better simulation of parallelism. This is in addition to random context switching due to time slicing (see Section 3.7.5). The extra delays help during the debugging phase of a μC++ program, but slows the execution of the program.

-nodelay Context switches are not inserted in member routines. **This is the default.**

-verify When the program is translated, a check to verify that the stack has not overflowed occurs at the beginning of each member routine. (This is in addition to checks on each context switch provided by the -debug option.) Verifying the stack has not overflowed is important during the debugging phase of a μC++ program, but slows the execution of the program.

-noverify Stack checking is not inserted in member routines. **This is the default.**

-inline When the program is translated, as much μC++ kernel code as possible is inlined to decrease runtime at the cost of increased compilation time, increased program size, and possibly poorer debugging capabilities.

-noinline None of the μC++ kernel code is inlined, which reduces compilation cost, decreases program size, and possibly makes debugging easier at the cost of increased runtime. **This is the default.**

-trace When the program is translated, additional runtime code is inserted in objects to generate events for several μC++ operations. Events for a particular object form a trace of the operations performed by or to that object. (See the manual "μC++ Monitoring, Visualization and Debugging Annotated Reference Manual" for information about displaying event traces.)

-notrace No event trace code is generated. **This is the default.**

-multi The program is linked with the multikernel.

-nomulti The program is linked with the unikernel. **This is the default.**

-quiet The printing of the μC++ compilation message at the beginning of the compilation is suppressed.

-noquiet The μC++ compilation message is printed at the beginning of the compilation. **This is the default.**

-U++ Only the C preprocessor and the μC++ translator are performed and the transformed program is written to stdout. This makes it possible to examine the code generated by the μC++ translator.

-compiler *name* This specifies the name of the compiler used to compile the μC++ program(s). This allows compilers other than the default GNU C++ compiler to be used to compile a μC++ program using u++. **Currently, only the GNU C++ compiler works.**

-cpp *name* This specifies the name of the C preprocessor used to preprocess the μC++ program. This allows a preprocessor other than the default GNU C++ preprocessor to be used to compile a μC++ program using u++.

When multiple conflicting options appear on the command line, e.g. -delay followed by -nodelay, the last option takes precedence. The u++ command is available by including /u/usystem/software/u++-3.7/bin in your command search path, which is usually located in the .cshrc file.

2.5.2 Preprocessor Variables

When programs are compiled using u++, the following translator variables are available during preprocessing. The translator variable __U_CPLUSPLUS__ is always available during preprocessing. If the -debug compilation option is specified, the translator variable __U_DEBUG__ is available during preprocessing. If the -delay compilation option is specified, the translator variable __U_DELAY__ is available during preprocessing. If the -verify compilation option is specified, the translator variable __U_VERIFY__ is available during preprocessing. If the -inline compilation option is specified, the translator variable __U_INLINE__ is available during preprocessing. If the -trace compilation option is specified, the translator variable __U_TRACE__ is available during preprocessing. If the -multi compilation option is specified, the translator variable __U_MULTI__ is available during preprocessing. This allows conditional compilation of programs that must work differently in these situations. For example, to allow a normal C/C++ program to be compiled using μC++ the following is necessary:

```
...
#ifdef __U_CPLUSPLUS__
void uMain::main() {
#else
int main( int argc, char *argv[] ) {
#endif
    ...
}
```

This conditionally includes the correct definition for main if the program is compiled using u++.

2.6 Coroutine

A coroutine is an object with its own execution-state so its execution can be suspended and resumed. Execution of a coroutine is suspended as control leaves it, only to carry on from that point when control returns at some later time. This means that a coroutine is not restarted at the beginning on each activation and that its local variables are preserved. Hence, a coroutine solves the class of problems associated with finite-state machines and push-down automata, which are logically characterized by the ability to retain state between invocations. In contrast, a free routine or member routine always executes to completion before returning so that its local variables only persist for a particular invocation. A coroutine executes serially, and hence there is no concurrency implied by the coroutine construct. However, the ability of a coroutine to suspend its execution-state and later have it resumed is the precursor to true tasks but without concurrency problems; hence, a coroutine is also useful to have in a programming language for teaching purposes because it allows incremental development of these properties [Yea91].

A coroutine type has all the properties of a class. The general form of the coroutine type is the following:

```
[uNoMutex] uCoroutine coroutine-name {
  private:
    ...                     // these members are not visible externally
  protected:
    ...                     // these members are visible to descendants
    void main();            // starting member
  public:
    ...                     // these members are visible externally
};
```

The coroutine type has one distinguished member, named main. Instead of allowing direct interaction with main, its visibility is private or protected; therefore, a coroutine can only be activated indirectly by one of the coroutine's member routines. A user interacts with a coroutine indirectly through its member routines. This allows a coroutine type to have multiple public member routines to service different kinds of requests that are statically type checked. main cannot have parameters, but the same effect can be accomplished indirectly by passing arguments to the constructor for the coroutine and storing these values in the coroutine's variables, which can be referenced by main.

A coroutine can suspend its execution at any point by activating another coroutine. This can be done in two ways. First, a coroutine can implicitly reactivate the coroutine that previously activated it. Second, a coroutine can explicitly invoke a member of another coroutine, which causes activation of the coroutine that it is a member of. Hence, two different styles of coroutine are possible, based on whether implicit activation is used. A **semi-coroutine** always activates the member routine that activated it; a **full coroutine** calls member routines in other coroutines that cause execution of that coroutine to be activated.

□ Coroutines can be simulated without using a separate execution-state, e.g. using a class, but this is difficult and error-prone for more than a small number of activation points. All data needed between activations must be local to the class and the coroutine structure must be written as a series of cases, each ending by recording the next case that will be executed on re-entry. Simulating a coroutine with a subroutine requires retaining data in variables with global scope or variables with static storage-class between invocations. However, retaining state in these ways violates the principle of abstraction and does not generalize to multiple instances, since there is only one copy of the storage in both cases. Simulating a coroutine with a task, which also has an execution-state, is non-trivial because the organizational structure of a coroutine and task are different. Further, simulating full coroutines that form a cyclic call-graph is not possible with tasks because of a task's mutual exclusion, which would cause deadlock. Finally, a task is inefficient for this purpose because of the higher cost of switching both a thread and execution-state as opposed to just an execution-state. In this implementation, the cost of communication with a coroutine is, in general, less than half the cost of communication with a task, unless the communication is dominated by transferring large amounts of data. □

2.6.1 Coroutine Creation and Destruction

A coroutine is the same as a class-object with respect to creation and destruction, as in:

```
uCoroutine C {
    void main() ...
  public:
    void r( ... ) ...
};
C *cp;                             // pointer to a C coroutine
{ // start a new block
    C c, ca[3];                    // local creation
    cp = new C;                    // dynamic creation
    ...
    c.r( ... );                    // call a member routine that activates the coroutine
    ca[1].r( ... );                // call a member routine that activates the coroutine
    cp->r( ... );                  // call a member routine that activates the coroutine
    ...
} // c, ca[0], ca[1] and ca[2] are destroyed
...
delete cp;                         // cp's instance is destroyed
```

When a coroutine is created the following occurs. The appropriate coroutine constructor and any base-class constructors are executed in the normal order. The stack component of the coroutine's execution-state is then created and the starting point (activation point) is initialized to the coroutine's main routine; however, the main routine does not start execution until the coroutine is activated by one of its member routines. The location of a coroutine's variables—in the coroutine's data area or in member routine main—depends on whether the variables must be accessed by member routines other than main. Once main is activated, it executes until it activates another coroutine or terminates. The coroutine's point of last activation may be outside of the main routine because main may have called another routine; the routine called could be local to the coroutine or in another coroutine.

A coroutine terminates when its main routine terminates. When a coroutine terminates, it activates the coroutine or task that caused main to *start* execution. This choice was made because the start sequence is a tree, i.e. there are no cycles. A thread can move in a cycle among a group of coroutines but termination always proceeds back along the branches of the starting tree. This choice for termination does impose certain requirements on the starting order of coroutines, but it is essential to ensure that cycles can be broken at termination. An attempt to communicate with a terminated coroutine is an error.

Like a routine or class, a coroutine can access all the external variables of a C++ program and the heap area. Also, any static member variables declared within a coroutine are shared among all instances of that coroutine type. If a coroutine makes global references or has static variables and is instantiated by different tasks, there is the general problem of concurrent access to these shared variables.

2.6.2 Inherited Members

Each coroutine type, if not derived from some other coroutine type, is implicitly derived from the coroutine type uBaseCoroutine, as in:

```
uCoroutine coroutine-name : public uBaseCoroutine {
    ...
};
```

where the interface for the base class uBaseCoroutine is as follows:

```
uCoroutine uBaseCoroutine {
  public:
    uBaseCoroutine();
    uBaseCoroutine( int stackSize );
    void uVerify();
    void uSetName( const char *name );
    const char *uGetName() const;
};
```

The overloaded constructor routine uBaseCoroutine has the following forms:

uBaseCoroutine() – creates the coroutine on the current cluster with the cluster's default stack size.

uBaseCoroutine(int stackSize) – creates the coroutine on the current cluster with the specified stack size (in bytes).

A coroutine type can be designed to allow declarations to specify the size of the stack by doing the following:

```
uCoroutine C {
  public:
    C() : uBaseCoroutine( 8192 ) {};            // default 8K stack
    C( int s ) : uBaseCoroutine(s) {};          // user specified stack size
    ...
};

C x, y( 16384 );                // x has an 8K stack, y has a 16K stack
```

The member routine uVerify checks whether the current coroutine has overflowed its stack. If it has, the program terminates. A call to uVerify might be included after each set of declarations, as in the following example:

```
void main() {
   ...                        // declarations
   uVerify();                 // check for stack overflow
   ...                        // code
}
```

Thus, after a coroutine has allocated its local variables, a verification is made that the stack was large enough to contain them.

□ When the -verify option is used, calls to uVerify are automatically inserted at the beginning of each member routine, but not after each set of declarations. □

The member routine uSetName associates a name with a coroutine. The member routine uGetName returns the string name associated with a coroutine. If the coroutine has not been assigned a name, uGetName returns the type name of the coroutine instance. μC++ uses the name when printing any error message, which is helpful in debugging.

The free routine:

```
uBaseCoroutine &uThisCoroutine();
```

is used to determine the identity of the current coroutine. Because it returns a reference to the base coroutine type, uBaseCoroutine, this reference can only be used to access the public routines of type uBaseCoroutine. For example, a free routine can check whether the allocation of its local variables has overflowed the stack of a coroutine that called it by performing the following:

```
int FreeRtn( ... ) {
    ...                              // declarations
    uThisCoroutine().uVerify();      // check for stack overflow
    ...                              // code
}
```

2.6.3 Coroutine Control and Communication

Control flow among coroutines is specified by the uResume and uSuspend statements. The uResume statement is used only in member routines; It always activates the coroutine in which it is specified, and consequently, causes the caller of the member routine to become inactive. The uSuspend statement is used only within the coroutine body or local members called directly or indirectly from the coroutine body. it causes the coroutine to become inactive, and consequently, to activate the caller that most recently activated the coroutine. In terms of the execution properties, these statements redirect a thread to a different execution-state. The execution-state can be that of a coroutine or the current task, i.e. a task's thread can execute using its execution-state, then several coroutine execution-states, and then back to the task's execution-state. Therefore, these statements activate and deactivate execution-states, and do not block and make ready threads.

A semi-coroutine is characterized by the fact that it always activates its caller, as in the producer-consumer example of Figure 2.2. Notice the explicit call from Prod's main routine to delivery and then the return back when delivery completes. delivery always activates its coroutine, which subsequently activates delivery. Appendix B.1 shows a complex binary insertion sort implemented as a semi-coroutine.

Producer

```
uCoroutine Prod {
    Cons *c;
    int N, status;
    void main() {
        int p1, p2;
        // 1st resume starts here
        for ( int i = 1; i <= N; i += 1 ) {
            ... // generate a p1 and p2
            status = c->delivery( p1, p2 );
            if ( status == ... ) ...
        } // for
        c->stop();
    } // main
  public:
    Prod( Cons *c ) { Prod::c = c; };
    void start( int N ) {
        Prod::N = N;
        uResume;                // restart Prod::main
    } // start
}; // Prod

void uMain::main() {
    Cons cons;                  // create consumer
    Prod prod( &cons );         // create producer
    prod.start( 10 );           // start producer
} // main
```

Consumer

```
uCoroutine Cons {
    int p1, p2, status, done;
    void main() {
        // 1st resume starts here
        while ( !done ) {
            // consume p1 and p2
            status = ...;
            uSuspend;           // restart Cons::delivery
        } // while
    } // main
  public:
    Cons() { done = 0; };
    int delivery( int p1, int p2 ) {
        Cons::p1 = p1;
        Cons::p2 = p2;
        uResume;                // restart Cons::main
        return status;
    } // delivery
    void stop() {
        done = 1;
        uResume;
    } // stop
}; // Cons
```

Figure 2.2: Semi-Coroutine Producer-Consumer

A full coroutine is characterized by the fact that it never activates its caller; instead, it activates another coroutine by invoking one of its member routines. Thus, full coroutines activate one another often in a cyclic fashion, as in the producer-consumer example of Figure 2.3. Notice the uResume statements in routines payment and delivery. The uResume in routine payment activates the execution-state associated with Prod::main and that execution-state continues in routine Cons::delivery. Similarly, the uResume in routine delivery activates the execution-state associated with Cons::main and that execution-state continues in Cons::main initially and subsequently in routine Prod::payment. This cyclic control flow and the termination control flow is illustrated in Figure 2.4.

2.7 Mutex Types

A mutex type consists of a set of variables and a set of mutex members, which operate on the variables. **A mutex type has at least one mutex member.** Objects instantiated from mutex types have the property

Producer	Consumer

<pre>uCoroutine Prod {
    Cons *c;
    int N, money, status, receipt;
    void main() {
        int p1, p2;
        // 1st resume starts here
        for (int i = 1; i <= N; i += 1) {
            ... // generate a p1 and p2
            status = c->delivery(p1, p2);
            if (status == ...) ...
        } // for
        c->stop();
    } // main
  public:
    int payment(int money) {
        Prod::money = money;
        ... // process money
        uResume; // restart prod in Cons::delivery
        return receipt;
    } // payment
    void start(int N, Cons *c) {
        Prod::N = N;
        Prod::c = c;
        uResume;
    } // start
}; // Prod
 	
void uMain::main() {
    Prod prod;   // create producer
    Cons cons(&prod);   // create consumer
    prod.start(10, &cons); // start producer
} // main</pre>
	<pre>uCoroutine Cons {
	    Prod *p;
	    int p1, p2, status, done;
	    void main() {
	        int money, receipt;
	        // 1st suspend starts here
	        while (!done) {
	            // consume p1 and p2
	            status = ...
	            receipt = p->payment(money);
	        } // while
	    } // main
	  public:
	    Cons(Prod *p) { Cons::p = p; done = 0; }
	    int delivery(int p1, int p2) {
	        Cons::p1 = p1;
	        Cons::p2 = p2;
	        uResume; // restart cons in Cons::main 1st
	        // time and cons in Prod::payment afterwards
	        return status;
	    } // delivery
	    void stop() {
	        done = 1;
	        uResume;
	    } // stop
	}; // Cons</pre>

Figure 2.3: Full-Coroutine Producer-Consumer

that mutex members are executed with mutual exclusion; that is, only one task at a time can be executing in the mutex members. This task is termed the **active** task. Mutual exclusion is enforced by **locking** the mutex object when execution of a mutex member begins and **unlocking** it when the active task voluntarily gives up control of the mutex object. If another task invokes a mutex member while a mutex object is locked, the task is blocked until the mutex type becomes unlocked.

When uMutex or uNoMutex qualifies a type, as in:

```
uMutex class M {
  private:
    char  w( ... );
  public:
    M();
    ~M();
    int x( ... );
    float y( ... );
    void z( ... );
};
```

it defines the default form of mutual exclusion on *all* public member routines, including the constructor and destructor. Hence, public member routines M, ~M, x, y, and z of monitor type M are mutex members executing

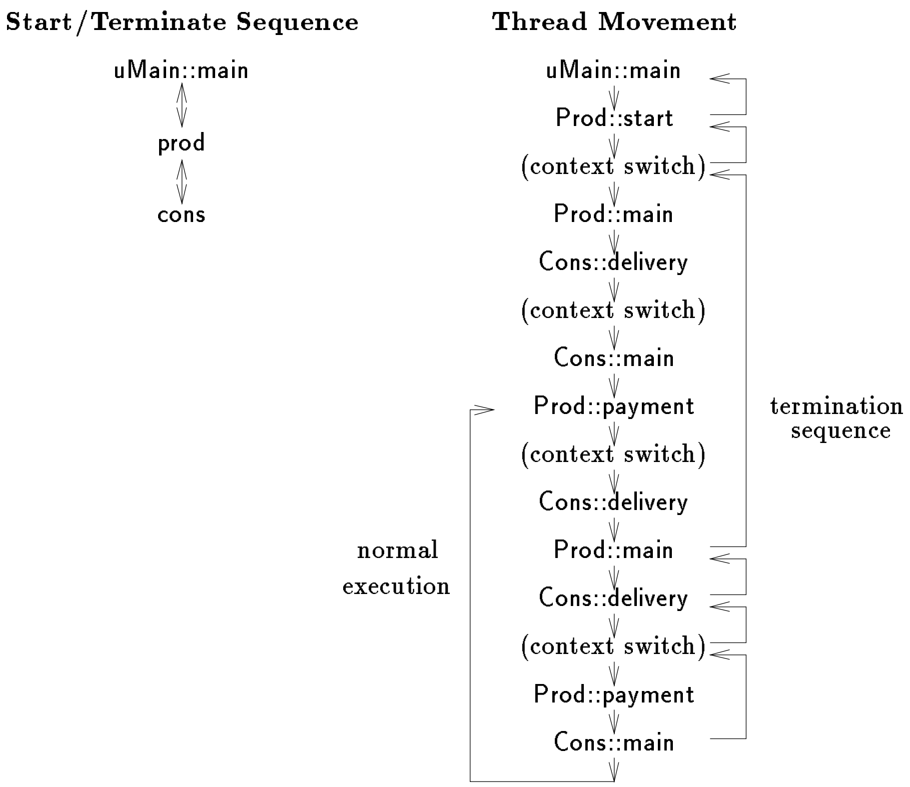

Figure 2.4: Cyclic Control Flow in Full Coroutine

mutually exclusively of one another. protected and private member routines are *always* implicitly uNoMutex, except for main in coroutine-monitors and tasks. Because the destructor of a mutex object is executed mutually exclusively, the termination of a block containing a mutex object or deleting a dynamically allocated one may block if the destructor cannot be executed immediately. **In μC++, a mutex member cannot call another mutex member in the same object without the executing thread deadlocking with itself.**

A mutex qualifier may be needed for protected and private member routines in mutex types, as in:

```
uMutex class M {
  private:
    uMutex char w( ... );            // explicitly qualified member routine
    ...
};
```

because another thread may need access to these member routines. For example, when a friend task calls a protected or private member routine, these calls may need to provide mutual exclusion.

A public member of a mutex type can be explicitly qualified with uNoMutex. Such a routine is, in general, error-prone in concurrent situations because the lack of mutual exclusion permits concurrent updating to object variables. However, there are two situations where a non-mutex public member are useful: first, for read-only member routines where execution speed is of critical importance; and second, to encapsulate a sequence of calls to several mutex members to establish a protocol, which ensures that a user cannot violate the protocol since it is part of the object's definition.

The general structure of a mutex object is shown in Figure 2.5. All the implicit and explicit data structures associated with a mutex object are discussed in the following sections. Notice that each mutex member has a queue associated with it on which calling tasks wait if the mutex object is locked. A no-mutex member has no queue.

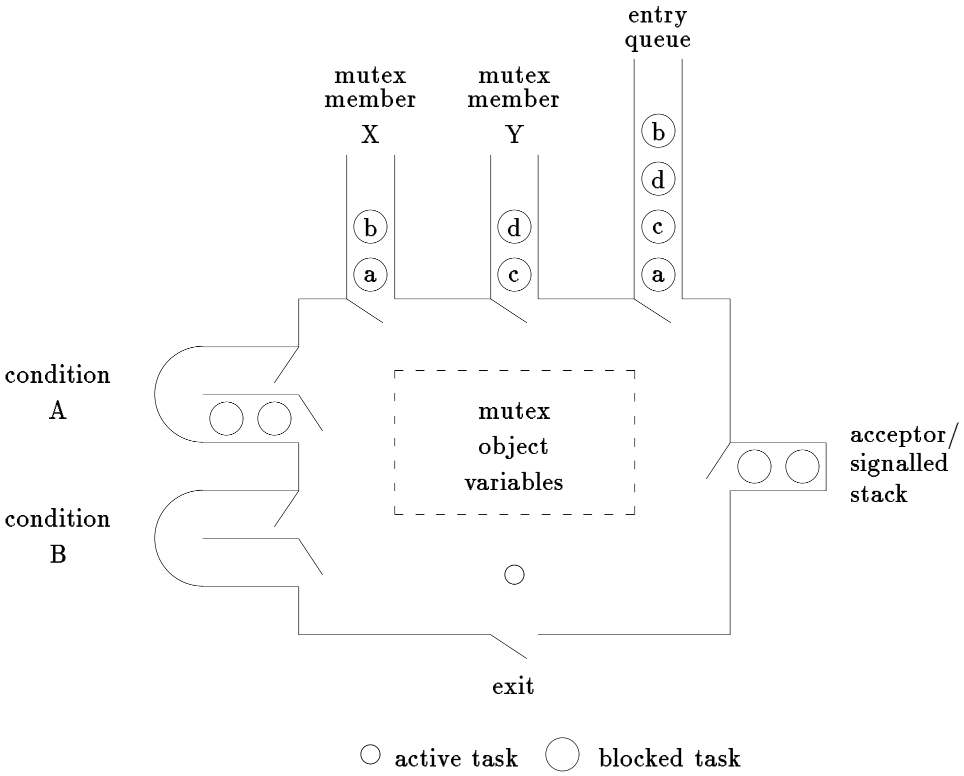

Figure 2.5: General Mutex Object

2.8 Thread Control Statements

For many purposes, the mutual exclusion that is provided automatically by mutex members is all that is needed, e.g. an atomic counter, as in:

```
uMutex class atomicounter {
    int cnt;
  public:
    atomicounter() { cnt = 0; }
    inc() { cnt += 1; }                // atomically increment counter
}
```

However, it is sometimes necessary to synchronize with tasks calling or executing within the mutex object. For this purpose, a task in a mutex object can block until a particular external or internal event occurs. At some point after a task has blocked, it must be reactivated by another (active) task.

2.8.1 Implicit Scheduling

Implicit scheduling occurs when a mutex object becomes unlocked because the active task blocks or exits a mutex member. The next task to use the mutex object is then chosen from one of a number of lists internal to the mutex object. Figure 2.5 shows a mutex object with a set of tasks using or waiting to use it. When a calling task finds the mutex object locked, it is added to both the "mutex member queue" of the member routine it called and the "entry queue"; otherwise it enters the mutex object and locks it. The entry queue is a list of all the calling tasks in chronological order, which is important for selecting a task when there is no active task in a mutex object. When a task is blocked implicitly or is reactivated by another (active) task, it is added to the top of the "acceptor/signalled stack".

When a mutex object becomes unlocked, the next task to execute is selected by an internal scheduler. In the following thread control statements, the internal scheduler may be directed to select from a specific set of queues; hence, there is no choice with regard to which queues are examined. In other cases, the internal scheduler may make a choice among all the queues. When a choice is possible, the internal scheduler for μC++ makes selections based on the results presented in [BCF] to give the user the greatest possible control and produce efficient performance. This means the scheduler follows these selection rules:

1. Select tasks that have entered the mutex object, blocked, and now need to continue execution over tasks that have called and are waiting on entry queues.
2. When one task reactivates a task that was previously blocked in the mutex object, the restarting task always continues execution and the reactivated task continues to wait until it is selected for execution by rule 1.

All other tasks must wait until the mutex object is again unlocked. Therefore, when selection is done implicitly, the next task to resume is not under direct user control.

2.8.2 External Control

A mutex object can control the kind of external request it serves next using an accept statement.

2.8.2.1 Accept Statement

A uAccept statement is provided to dynamically choose which mutex member executes next. This indirectly controls which caller is accepted next, that is, the next caller to the accepted mutex member. The simple form of the uAccept statement is as follows:

```
uWhen ( conditional-expression )                // optional guard
    uAccept( mutex-member-name );
```

with the restriction that the constructor, new and delete, and uNoMutex members are excluded from being accepted. The first three member routines are excluded because these routines are essentially part of the implicit memory-management runtime support. uNoMutex members are excluded because, in this implementation, they contain no code that could affect the caller or acceptor.

> □ While it is possible to block the caller to and the acceptor of a uNoMutex member and then continue both the caller and acceptor when they synchronize, we do not believe that such a facility is particularly useful. □

The syntax for accepting a mutex operator member, such as operator =, is as follows:

```
uAccept( operator = );
```

Currently, there is no way to accept a particular overloaded member. Instead, when an overloaded member name appears in a uAccept statement, calls to any member with that name are accepted.

> □ A consequence of this design decision is that once one routine of a set of overloaded routines becomes mutex, all the overload routines in that set become mutex members. The rationale is that members with the same name should perform essentially the same function, and therefore, they all should be eligible to accept a call. □

A uWhen guard is considered true if it is omitted or its *conditional-expression* evaluates to non-zero. Before the uAccept statement is executed, the guard must be true and an outstanding call to the corresponding member must exist. If the guard is true and there is no outstanding call to that member, the task is accept-blocked until a call to the appropriate member is made. If the guard is false, the program is aborted; hence, the uWhen clause can act as an assertion of correctness in the simple case.

When a uAccept statement is executed, the acceptor is blocked and pushed on the top of the implicit acceptor/signalled stack and the mutex object is unlocked. The internal scheduler then schedules a task from the specified mutex member queue, possibly waiting until an appropriate call occurs. The accepted member is then executed like a member routine of a conventional class by the caller's thread. If the caller is expecting a return value, this value is returned using the return statement in the member routine. When the caller's thread exits the mutex member (or waits, as will be discussed shortly), the mutex object is unlocked. Because the internal scheduler gives priority to tasks on the acceptor/signalled stack of the mutex object over calling tasks, the acceptor is popped from the acceptor/signalled stack and made ready. When the acceptor becomes active, it has exclusive access to the object. Hence, the execution order between acceptor and caller is stack order, as for a traditional routine call.

The extended form of the uAccept statement can be used to accept one of a group of mutex members, as in:

```
uWhen ( conditional-expression )               // optional guard
   uAccept( mutex-member-name )
      statement                                // statement
uOr uWhen ( conditional-expression )           // optional guard
   uAccept( mutex-member-name )
      statement                                // statement
...
   ...
      ...
uElse                                          // optional default clause
   statement
```

Before a uAccept clause is executed, the guard must be true and an outstanding call to the corresponding member must exist. If there are several mutex members that can be accepted, the uAccept clause nearest the beginning of the statement is executed. Hence, the order of the uAccepts indicates their relative priority for selection if there are several outstanding calls. Once the accepted call has completed, the statement after the accepting uAccept clause is executed. If there is a uElse clause and no uAccept can be executed immediately, the uElse clause is executed instead. Hence, the uElse clause allows a conditional attempt to accept a call without the acceptor blocking. If there is no uElse clause and the guards are all false, the program is aborted. If some guards are true and there are no outstanding calls to these members, the task is accept-blocked until a call to one of these members is made.

□ Note that the syntax of the uAccept statement precludes the caller's argument values from being accessed in the *conditional-expression* of a uWhen. However, this deficiency is handled by the ability of a task to postpone requests (see Section 2.8.3.1). □

□ The uOr clause seems superfluous because it could be replaced by the uElse clause (or vice versa). However, the following situation could cause significant confusion if uElse was used to separate the uAccept clauses:

```
uAccept( fred );                      uAccept( fred );
uElse uAccept( mary );                uElse { uAccept( mary ) };
```

The left example accepts a call to either member fred or mary. The right example accepts a call to member fred if one is currently available, otherwise it accepts a call to member mary. The syntactic difference is subtle, and yet, their execution is significantly different. We believe that a less subtle syntactic difference between these two case, as in:

```
uAccept( fred );                      uAccept( fred );
uOr uAccept( mary );                  uElse uAccept( mary );
```

helps to prevent the problem. □

2.8.2.2 Accepting the Destructor

Accepting the destructor in a uAccept statement is used to terminate a mutex object when it is de-allocated (like the terminate clause of the select statement in Ada [Uni83, Section 9.4, 9.7.1]). The destructor is accepted in the same way as a mutex member, as in:

```
for ( ;; ) {
    uAccept( ~DiskScheduler ) {                    // request to terminate DiskScheduler
        break;
    } uOr uAccept( WorkRequest ) {                 // request from disk
    } uOr uAccept( DiskRequest ) {                 // request from clients
    } // uAccept
} // for
// clean up code
```

However, the semantics for accepting a destructor are different from accepting a normal mutex member. When the call to the destructor occurs, the caller blocks immediately because a mutex object's storage cannot be deallocated if it is being used by a thread. When the destructor is accepted, the caller is blocked and pushed onto the acceptor/signalled stack instead of the acceptor. Therefore, control restarts at the accept statement *without* executing the destructor member. This allows a mutex object to clean up before it terminates. Only when the caller to the destructor is popped off the acceptor/signalled stack by the internal scheduler is the destructor execute. Once the destructor is accepted, it is not possible to reactivate blocked tasks below it on the acceptor/signalled stack. It is the programmers responsibility to ensure that the acceptor/signalled stack is empty before accepting the destructor.

> □ While a mutex object can always be setup so that the destructor does all the cleanup, this can force variables that logically belong in member routines into the mutex object. Further, the fact that control would not return to the uAccept statement when the destructor is accepted seemed more confusing than having special semantics for accepting the destructor. □

Accepting the destructor can be used by a mutex object to know when to stop without having to accept a special call. For example, by allocating tasks in a specific way, a server task for a number of clients can know when the clients are finished and terminate without having to be explicitly told, as in:

```
{
    DiskScheduler ds;                              // start DiskScheduler task
    {
        Clients c1(ds), c2(ds), c3(ds);            // start clients, which communicate with ds
    } // wait for clients to terminate
} // implicit call to DiskScheduler's destructor
```

Commentary

In contrast to Ada, a uAccept statement in μC++ places the code to be executed in a mutex member; thus, it is specified separately from the uAccept statement. An Ada-style accept specifies the accept body as part of the accept statement, requiring the accept statement to provide parameters and a routine body. Since we have found that having more than one accept statement per member is rather rare, our approach gives essentially the same capabilities as Ada. As well, accepting member routines also allows virtual routine redefinition, which is not possible with accept bodies. Finally, an accept statement with parameters and a routine body does not fit with the design of C++ because it is like a nested routine definition, and since routines cannot be nested in C++, there is no precedent for such a facility. It is important to note that anything that can be done in Ada-style accept statements can be done within member routines, possibly with some additional code. If members need to communicate with the block containing the uAccept statements, it can be done by leaving "memos" in the mutex types variables. In cases where there would be several different Ada-style accept statements for the same entry, accept members would have to start with switching logic to determine which case applies. While neither of these solutions is particularly appealing, the need to use them seems to arises only rarely.

2.8.3 Internal Control

Tasks within a mutex object may need to wait and synchronize with one another during the servicing of their request. For that purpose, **condition variables** are provided, with the associated operations **wait** and **signal**.

2.8.3.1 Condition Variables and the Wait and Signal Statements

The type uCondition creates a queue object on which tasks can be blocked and reactivated in first-in first-out order, and is defined:

```
class uCondition {
  public:
    int uEmpty();
};

uCondition DiskNotIdle;
```

The member routine uEmpty() returns 0 if there are tasks blocked on the queue and 1 otherwise. It is *not* meaningful to read or to assign to a condition variable, or copy a condition variable (e.g. pass it as a value parameter), or use a condition variable outside of the mutex object in which it is declared.

It is common to associate with each condition variable an assertion about the state of the mutex object. For example, in a disk-head scheduler, a condition variable might be associated with the assertion "the disk head is idle". Waiting on that condition variable would correspond to waiting until the condition is satisfied, that is, until the disk head is idle. Correspondingly, the active task would reactivate tasks waiting on that condition variable only when the disk head became idle. The association between assertions and condition variables is implicit and not part of the language.

To join such a queue, the active task executes a uWait statement, for example,

```
uWait DiskNotIdle;
```

This causes the active task to block on condition DiskNotIdle, which unlocks the mutex object and invokes the internal scheduler. The internal scheduler first attempts to pop a task from the acceptor/signalled stack. If there are no tasks on the acceptor/signalled stack, the internal scheduler selects a task from the entry queue or waits until a call occurs if there are no tasks; hence, the next task to enter is the one blocked the longest. If the internal scheduler did not accept a call at this point, deadlock would occur.

A task is reactivated from a condition variable when another (active) task executes a uSignal statement, for example,

```
uSignal DiskNotIdle;
```

The effect of a uSignal statement is to remove one task from the specified condition variable and push it on the acceptor/signalled stack. The signaller continues execution and the signalled task is scheduled by the internal scheduler when the mutex object is nexted unlocked. This is different from the uAccept statement, which always blocks the acceptor; **the signaller does not block.**

> □ The uAccept, uWait and uSignal statements can be executed by any routine of a mutex type. Even though these statements block the current task, they can be allowed in member routines because member routines are executed by their caller, not the task of which they are members. This is to be contrasted to Ada where the use of a statement like a uWait in an accept body would cause the task to deadlock. □

> □ Ultimately, we want to restart tasks blocked within a terminating mutex object—on entry queues and condition variables—by raising an exception to notify them that their call failed. We are currently examining how to incorporate exceptions among tasks within the proposed C++ exception model [BMZ92]. □

Commentary

The ability to postpone a request is an essential requirement of a programming language's concurrency facilities. Postponement may occur multiple times during the servicing of a request while still allowing an object to accept new requests.

In simple cases, the uWhen construct can be used to accept only requests that can be completed without postponement. However, when the selection criteria become complex, e.g. when the parameters of the request are needed to do the selection or information is needed from multiple queues, it is simpler to unconditionally accept a request and subsequently postpone it if it does not meet the selection criteria. This avoids complex selection expressions and possibly their repeated evaluation. In addition, this allows all the programming language constructs and data structures to be used in making the decision to postpone a request, instead of some fixed selection mechanism provided in the programming language, as in SR [AOC+88] and Concurrent C++ [GR88].

Regardless of the power of a selection facility, none can deal with the need to postpone a request after it has been accepted. In a complex concurrent system, a task may have to make requests to other tasks as part of servicing a request. Any of these further requests can indicate that the current request cannot be completed at this time and must be postponed. Thus, we believe that it is essential that a request be able to be postponed even after it is accepted so that the acceptor can make this decision while the request is being serviced. Therefore, condition variables seem essential to support this facility.

An alternative approach to condition variables is to send the request to be postponed to another (usually non-public) mutex member of the object (like Ada 9X's requeue statement). This action re-blocks the request on that mutex member's entry queue, which can be subsequently accepted when the request can be restarted. However, there are problems with this approach. First, the postponed request cannot be sent directly from a mutex member to another mutex member or deadlock may occur because of synchronous communication. (Asynchronous communication solves this problem, but as stated earlier, imposes a substantial system complexity and overhead.) The only alternative is to use a non-mutex member, which calls a mutex member to start the request and checks its return code to determine if the request must be postponed. If the request is to be postponed, another mutex member is invoked to block the current request until it can be continued. Unfortunately, structuring the code in this fashion becomes complex for non-trivial cases and there is little control over the order that requests are processed. In fact, the structuring problem is similar to the one when simulating a coroutine using a class or subroutine, where the programmer must explicitly handle the different execution states. Second, any mutex member working on the request may accumulate temporary results. If the request must be postponed, the temporary results must be returned and bundled with the initial request that are forwarded to the mutex member that handles the next step of the processing; alternatively, the temporary results can be re-computed at the next step if that is possible. In contrast, waiting on a condition variable automatically saves the execution location and any partially computed state.

2.9 Monitor

A monitor is an object with mutual exclusion and so it can be accessed simultaneously by multiple tasks. A monitor provides a mechanism for indirect communication among tasks and is particularly useful for managing shared resources. A monitor type has all the properties of a class. The general form of the monitor type is the following:

```
uMutex class monitor-name {
  private:
    ...                     // these members are not visible externally
  protected:
    ...                     // these members are visible to descendants
  public:
    ...                     // these members are visible externally
};
```

The macro name uMonitor is defined to be "uMutex class" in uC++.h.

2.9.1 Monitor Creation and Destruction

A monitor is the same as a class-object with respect to creation.

```
uMutex class M {
  public:
    void r( ... ) ...
};
M *mp;                          // pointer to a M
{ // start a new block
    M m, ma[3];                 // local creation
    mp = new M;                 // dynamic creation
    ...
    m.r( ... );                 // call a member routine that must be accepted
    ma[1].r( ... );             // call a member routine that must be accepted
    mp->r( ... );               // call a member routine that must be accepted
    ...
} // wait for m, ma[0], ma[1] and ma[2] to terminate and then destroy
...
delete mp;                      // wait for mp's instance to terminate and then destroy
```

Because a monitor is a mutex object, the execution of its destructor waits until it can gain access to the monitor, just like the other public members of the monitor, which can delay the termination of the block containing a monitor or the deletion of a dynamically allocated monitor.

2.9.2 Monitor Control and Communication

In μC++, the uAccept statement can be used to control which member(s) can be executed next in a monitor. This ability makes μC++ monitors more general than conventional monitors [Hoa74] because they specify a mutex member, while uSignal specifies only a condition variable. This gives the ability to restrict which member can be called, instead of having to accept all calls and subsequently handle or block them, as for conventional monitors. Figure 2.6 compares a traditional style monitor using explicit condition variables to one that uses accept statements. The problem is the exchange of values (telephone numbers) between two kinds of tasks (girls and boys). (uAccept allows the removal of all condition variables in this case, but that is not always possible.)

2.10 Coroutine-Monitor

The coroutine-monitor is a coroutine with mutual exclusion and so it can be accessed simultaneously by multiple tasks. A coroutine-monitor type has a combination of the properties of a coroutine and a monitor, and can be used where a combination of these properties are needed, such as a finite-state machine that is used by multiple tasks. A coroutine-monitor type has all the properties of a class. The general form of the coroutine-monitor type is the following:

```
uMutex uCoroutine coroutine-name {
  private:
    ...                  // these members are not visible externally
  protected:
    ...                  // these members are visible to descendants
    void main();         // starting member
  public:
    ...                  // these members are visible externally
};
```

Currently, we have little experience in using a coroutine-monitor, nevertheless we believe it merits further examination.

2.10.1 Coroutine-Monitor Creation and Destruction

A coroutine-monitor is the same as a monitor with respect to creation and destruction.

Traditional Method	New Method

<pre>uMonitor DatingService {
    int GirlPhoneNo, BoyPhoneNo;
    uCondition GirlWaiting, BoyWaiting, Confirmed;
  public:
    int Girl(int PhoneNo) {
            if (BoyWaiting.uEmpty()) {
                uWait GirlWaiting;
                GirlPhoneNo = PhoneNo;
                uSignal Confirmed;
            } else {
                GirlPhoneNo = PhoneNo;
                uSignal BoyWaiting;
                uWait Confirmed;
            } // if
            return BoyPhoneNo;
    } // Girl
    int Boy(int PhoneNo) {
            if (GirlWaiting.uEmpty()) {
                uWait BoyWaiting;
                BoyPhoneNo = PhoneNo;
                uSignal Confirmed;
            } else {
                BoyPhoneNo = PhoneNo;
                uSignal GirlWaiting;
                uWait Confirmed;
            } // if
            return GirlPhoneNo;
    } // Boy
}; // DatingService</pre>
	<pre>uMonitor DatingService {
	    int GirlPhoneNo, BoyPhoneNo;
	  public:
	    DatingService() {
	            GirlPhoneNo = BoyPhoneNo = -1;
	    }; // DatingService
	    int Girl(int PhoneNo) {
	            GirlPhoneNo = PhoneNo;
	            if (BoyPhoneNo == -1) {
	                uAccept(Boy);
	            } // if
	            int temp = BoyPhoneNo;
	            BoyPhoneNo = -1;
	            return temp;
	    }; // Girl
	    int Boy(int PhoneNo) {
	            BoyPhoneNo = PhoneNo;
	            if (GirlPhoneNo == -1) {
	                uAccept(Girl);
	            } // if
	            int temp = GirlPhoneNo;
	            GirlPhoneNo = -1;
	            return temp;
	    }; // Boy
	}; // DatingService</pre>

Figure 2.6: Traditional versus New Monitor Control

2.10.2 Coroutine-Monitor Control and Communication

A coroutine-monitor can make use of uSuspend, uResume, uAccept and uCondition variables, uWait and uSignal to move a task among execution-states and to block and restart tasks that enter it. When creating a cyclic call-graph using a coroutine-monitor, it is the programmer's responsibility to ensure that at least one of the members in the cycle is a uNoMutex member or deadlock occurs because of the mutual exclusion.

2.11 Task

A task is an object with its own thread of control and execution-state, and whose public member routines provide mutual exclusion. A task type has all the properties of a class. The general form of the task type is the following:

```
uTask task-name {
  private:
    ...                      // these members are not visible externally
  protected:
    ...                      // these members are visible to descendants
    void main();             // starting member
  public:
    ...                      // these members are visible externally
};
```

The task type has one distinguished member, named main, in which the new thread starts execution. Instead of allowing direct interaction with main, its visibility is private or protected. A user then interacts with a task's main member indirectly through its member routines. This allows a task type to have multiple public member routines to service different kinds of requests that are statically type checked. main cannot have parameters, but the same effect can be accomplished indirectly by passing arguments to the constructor for the task and storing these values in the task's variables, which can be referenced by main.

2.11.1 Task Creation and Destruction

A task is the same as a class-object with respect to creation and destruction, as in:

```
uTask T {
    void main() ...
  public:
    void r( ... ) ...
};
T *tp;                          // pointer to a T
{ // start a new block
    T t, ta[3];                 // local creation
    tp = new T;                 // dynamic creation
    ...
    t.r( ... );                 // call a member routine that must be accepted
    ta[1].r( ... );             // call a member routine that must be accepted
    tp->r( ... );               // call a member routine that must be accepted
    ...
} // wait for t, ta[0], ta[1] and ta[2] to terminate and then destroy
...
delete tp;                      // wait for tp's instance to terminate and then destroy
```

When a task is created, the appropriate task constructor and any base-class constructors are executed in the normal order by the creating thread. Then a new thread of control and execution-state are created for the task, which are used to begin execution of the main routine visible by the inheritance scope rules from the task type. From this point, the creating thread executes concurrently with the new task's thread. main executes until its thread blocks or terminates.

A task terminates when its main routine terminates. When a task terminates, so does the task's thread of control and execution-state. A task's destructor is invoked by the destroying thread when the block containing the task declaration terminates or by an explicit delete statement for a dynamically allocated task. Because a task is a mutex object, a block containing one cannot terminate until all tasks declared in the block terminate. Similarly, deleting a task on the heap must also wait until the task being deleted has terminated. An attempt to communicate with a terminated task is an error.

While a task that creates another task is conceptually the parent and the created task its child, μC++ makes no implicit use of this relationship nor does it provide any facilities based on this relationship. Once a task is declared it has no special relationship with its declarer other than what results from the normal scope rules.

Like a coroutine, a task can access all the external variables of a C++ program and the heap area. However, because tasks execute concurrently, there is the general problem of concurrent access to such shared variables. Further, this problem may also arise with static member variables within a task that is instantiated multiple times. Therefore, it is suggested that these kinds of references be used with extreme caution.

> □ A coroutine is not owned by the task that creates it. It can be "passed off" to another task. However, to ensure that only one thread is executing a coroutine at a time, the passing around of a coroutine must involve a protocol among its users. This is the same sort of protocol that is required when multiple tasks share a data structure. □

2.11.2 Inherited Members

Each task type, if not derived from some other task type, is implicitly derived from the task type uBaseTask, as in:

```
uTask task-name : public uBaseTask {
    ...
};
```

where the interface for the base class uBaseTask is as follows:

```
uTask uBaseTask : public uBaseCoroutine {            // inherit from coroutine base type
  public:
    uBaseTask();
    uBaseTask( int stackSize );
    uBaseTask( uCluster &cluster() );
    uBaseTask( uCluster &cluster, int stackSize );

    uCluster &uMigrate( uCluster &cluster );
    void uDelay( int times = 1 );
};
```

The public member routines uVerify, uSetName and uGetName are inherited from uBaseCoroutine and have the same functionality.

The overloaded constructor routine uBaseTask has the following forms:

uBaseTask() – creates the task on the current cluster with the cluster's default stack size (same as uBaseCoroutine()).

uBaseTask(int stackSize) – creates the task on the current cluster with the specified stack size (in bytes) (same as uBaseCoroutine(int stackSize)).

uBaseTask(uCluster &cluster) – creates the task on the specified cluster with that cluster's default stack size.

uBaseTask(uCluster &cluster, int stackSize) – creates the task on the specified cluster with the specified stack size (in bytes).

A task type can be designed to allow declarations to specify the cluster on which creation occurs and the size of the stack by doing the following:

```
uTask T {
  public:
    T() : uBaseTask( 8192 ) {};                      // current cluster, 8K stack
    T( int s ) : uBaseTask( s ) {};                  // current cluster and user stack size
    T( uCluster &c ) : uBaseTask( c ) {};            // user cluster
    T( uCluster &c, int s) : uBaseTask(c,s) {};      // user cluster and stack size
    ...
};
uCluster c;                 // create a new cluster
T x, y( 16384 );            // x has a default stack, y has a 16K stack
T z( c );                   // z created in cluster c with default stack size
T w( c, 16384 );            // w created in cluster c and has a 16K stack
```

The public member routine uMigrate allows a task to move itself from one cluster to another so that it can access resources that are dedicated to that cluster's processor(s).

from-cluster-reference = uMigrate(*to-cluster-reference*)

Although uMigrate is a public member routine, one task cannot migrate another task; a task can only migrate itself. A call such as:

```
x.uMigrate(...)
```

fails, via a dynamic check, if task x is not the same as the currently executing task. This is because a task cannot be migrated unless it is blocked (which can be done trivially when a task migrates itself). If one task could migrate another, the migrated task would have to be interrupted and blocked so that the migration does not occur at an unreasonable time in the future (i.e. waiting until the migrated task happens to block). Further, the ability to perform such a powerful operation on a task without its permission seems unreasonable.

The public member routine routine uDelay gives up control of the virtual processor to another ready task the specified number of times. For example, the routine call uDelay(5) immediately returns control to the μC++ kernel the next 5 times the task is scheduled for execution. If there are no other ready tasks, the delaying task is simply delayed and restarted 5 times. uDelay allows a task to relinquish control when it has no current work to do or when it wants other ready tasks to execute before it performs more work. An example of the former situation is when a task is polling for an event, such as a hardware event. After the polling task has determined the event has not occurred, it can relinquish control to another ready task, e.g. uDelay(1). An example of the latter situation is when a task is creating many other tasks. The creating task may not want to create a large number of tasks before the created tasks have a chance to begin execution. (Task creation occurs so quickly that it is possible to create 10-100 tasks before pre-emptive scheduling occurs.) If after the creation of several tasks the creator yields control, some created tasks will have an opportunity to begin execution before the next group of tasks is created. This facility is not a mechanism to control the exact order of execution of tasks; pre-emptive scheduling and/or multiple processors make this impossible.

Like member routine, uMigrate, one task cannot delay another task; a task can only delay itself. A call such as:

```
x.uDelay(...)
```

fails, via a dynamic check, if task x is not the same as the currently executing task. The reasons for this restriction are basically the same as for the restrictions on uMigrate.

> □ When the -delay option is used, calls to uDelay(random() % 3) are automatically inserted at the beginning of each member routine. □

The free routine:

```
uBaseTask &uThisTask();
```

is used to determine the identity of the current task. Because it returns a reference to the base task type, uBaseTask, of the current task, this reference can only be used to access the public routines of type uBaseTask and uBaseCoroutine. For example, a free routine can migrate or delay execution of the calling task by performing the following:

```
uThisTask().uMigrate(...);
uThisTask().uDelay();
```

2.11.3 Task Control and Communication

A task can make use of uAccept and uCondition variables, uWait and uSignal to block and make ready tasks that enter it. Appendix B.3 shows the archetypical disk scheduler implemented as a task that must process requests in an order other than first-in first-out to achieve efficient utilization of the disk.

Commentary

Initially, we attempted to add the new types and statements by creating a library of class definitions that were used through inheritance and preprocessor macros. This approach has been used to provide coroutine facilities [Sho87, Lab90] and simple parallel facilities [DG87, BLL88]. For example, the library approach involves defining an abstract class, Task, which implements the task abstraction. New task types are created by inheritance from Task, and tasks are instances of these types.

Task creation must be arranged so that the task body does not start execution until all of the task's initialization code has finished. One approach requires the task body (the code that appears in our main member) to be placed at the end of the new class's constructor, with code to start a new thread in Task::Task(). One thread then continues normally, returning from Task::Task() to complete execution of the constructors, while the other thread returns directly to the point where the task was declared. This is accomplished in the library approach by having one thread "diddle" with the stack to find the return address of the constructor called at the declaration. However, this scheme prevents further inheritance; it is impossible to derive a type from a task type if the new type requires a constructor, since the new constructor would be executed only *after* the parent constructor containing the task body. It also seems impossible to write stack diddling code that causes one thread to return directly to the declaration point if the exact number of levels of inheritance is not known. We tried to implement another approach that did not rely on stack diddling while still allowing inheritance and found it was impossible because a constructor cannot determine if it is the last constructor in an inheritance chain. Therefore, it is not possible to determine when all initialization is completed so that the new thread can be started. A mechanism like Simula's [Sta87] inner could be used to ensure that all initialization had been done before the task's thread is started. However, it is not obvious how inner would work in a programming language with multiple inheritance.

PRESTO solved this problem by providing a start() member routine in class Task, which must be called after the creation of a task. Task::Task() would set up the new thread, but start() would set it running. However, this two-step initialization introduces a new user responsibility: to invoke start before invoking any member routines or accessing any member variables.

A similar two-thread problem occurs during deletion when the destructors are called. The destructor of a task can be invoked while the task body is executing, but clean-up code must not execute until the task body has terminated. Therefore, the code needed to wait for a thread's termination cannot simply be placed in Task::~Task(), because it would be executed after all the derived class destructors have executed. Task designers could be required to put the termination code in the new task type's destructor, but that prevents further inheritance. Task could provide a finish() routine, analogous to start(), which must be called before task deletion, but that is error-prone because a user may fail to call finish appropriately, for example, before the end of a block containing a local task.

Communication among tasks also presents difficulties. In library-based schemes, it is often done via message queues. However, a single queue per task is inadequate; the queue's message type inevitably becomes a union of several "real" message types, and static type checking is compromised. (One could use inheritance from a Message class, instead of a union, but the task would still have to perform type tests on messages before accessing them.) If multiple queues are used, some analogue of the Ada select statement is needed to allow a task to block on more than one queue. Furthermore, there is no statically enforceable way to ensure that only one task is entitled to receive messages from any particular queue. Hence the implementation must handle the case of several tasks that are waiting to receive messages from overlapping sets of queues. For example,

```
class TaskType : Task {
  public:
    MsgQueueType A;              // queue associated with each instance of the task
    static MsgQueueType B;       // queue shared among all instances of the task type
  protected:
    void main() {
        ...
        uAccept i = A.front();   // accept from either message queue
        uOr uAccept i = B.front();
        ...
    }
};

TaskType T1, T2;
```

Tasks T1 and T2 are simultaneously accepting from two different queues. While it is straightforward to check for the existence of data in the queues, if there is no data, both T1 and T2 block waiting for data to appear on either queue. To implement this, tasks have to be associated with both queues until data arrives, given data when it arrives, and then removed from both queues. This would be expensive since the addition or removal of a message to/from a queue would have to be an atomic operation across all queues involved in a waiting task's accept statement to ensure that only one data item from the accepted set of queues is given to the accepting task.

If the more natural routine-call mechanism is to be used for communication among tasks, each public member routine would have to have special code at the start and possibly at the exits of each public member, which the programmer would have to provide. Other object-oriented programming languages that support inheritance of routines, such as LOGLAN'88 [CKL+88] and Beta [KMMPN87], or wrapper routines, as in GNU C++ [Tie88], might be able to provide automatically any special member code. Further, we could not find any convenient way to provide an Ada-like select statement without extending the language.

In the end, we found the library approach to be unsatisfactory. We decided that language extensions would better suit our goals by providing more flexible and consistent primitives, and static checking. It is also likely that language extensions can provide greater efficiency than a set of library routines.

2.12 Inheritance

C++ provides two forms of inheritance: "private" inheritance, which provides code reuse, and "public" inheritance, which provides reuse and subtyping (a promise of behavioural compatibility). (These terms must not be confused with C++ visibility terms with the same names.)

Class definitions can inherit from one another using both single and multiple inheritance. In μC++, there are three kinds of types, class, coroutine, and task, so the situation is more complex. The trivial case of single inheritance among homogeneous type specifiers, i.e. a class or coroutine or task type inherits from another class or coroutine or task, respectively, is supported in μC++. Further, multiple inheritance among classes is allowed as long as at most one of the base classes is a mutex class. Multiple inheritance is not allowed among coroutines or tasks. While there are some implementation difficulties with this multiple inheritance, the main reason is that it cannot be implemented efficiently. When coroutines and tasks inherit from other such types, each entity in the hierarchy may specify a main member; the main member specified in the last derived class of the hierarchy is the one that is started when a new instance is created. Clearly, there must be at least one main member specified in the hierarchy. For a task or a monitor type, new member routines that are defined by the derived class can be accepted by statements in a new main routine or in redefined virtual routines.

Inheritance among heterogeneous type specifiers is not supported. While there are some implementation difficulties with certain combinations and potential problems with non-virtual routines, the main reason is a fundamental one. Types are written as a class or a coroutine or a task possibly with mutual exclusion, and we do not believe that the coding styles used in each can be arbitrarily mixed. For example, an object produced by a task that inherits from a class can be passed to a routine expecting instances of the class and the routine might call one of the object's member routines that inadvertently blocks the current thread indefinitely. While this could happen in general, we believe there is a significantly greater chance if users casually combine types of different kinds.

Having mutex classes inherit from non-mutex classes is useful to generate concurrent usable types from existing non-concurrent types, for example, to define a queue that is derived from a simple queue and that can be accessed concurrently. However, there is a fundamental problem with non-virtual members in C++. To change a simple queue to a sharable queue, for example, would require a monitor to inherit from the class Queue and to redefine all of the class's member routines so that the correct mutual exclusion occurred when they are invoked. However, non-virtual routines in the Queue class might be called instead because non-virtual routines are statically bound. Consider, this attempt to create a sharable queue from a non-sharable queue:

```
class Queue {
  public:
    void insert( ... ) ...
    virtual void remove( ... ) ...
};
uMutex class MutexQueue : public Queue {
    virtual void insert( ... ) ...
    virtual void remove( ... ) ...
};
Queue *qp = new MutexQueue;          // subtyping allows assignment
qp->insert( ... );                   // call to a non-virtual member routine, statically bound
qp->remove( ... );                   // call to a virtual member routine, dynamically bound
```

Routine Queue::insert does not provide mutual exclusion because it is a member of the class, while routines MutexQueue::insert and MutexQueue::remove do provide mutual exclusion. Because the pointer variable qp is of type Queue, the call qp->insert calls Queue::insert even though insert was redefined in MutexQueue; so no mutual exclusion occurs. In contrast, the call to remove is dynamically bound, so the redefined routine in the monitor is invoked and appropriate synchronization occurs. The unexpected lack of mutual exclusion would cause many errors. In object-oriented programming languages that have only virtual member routines, this is not a problem. The problem does not occur with private inheritance because no subtype relationship is created and hence the assignment to qp would be invalid.

2.13 Exception Handling Facilities

future work, exceptions must be on a coroutine/task basis

2.14 Implementation Problems

The following restrictions are an artifact of this implementation. In some cases the restriction results from the fact that μC++ is only a translator and not a compiler. In all other cases, the restrictions exists simply because time limitations on this project have prevent it from being implemented.

- Some runtime member routines are publicly visible when they should not be; therefore, μC++ programs should not contain variable names that start with a "u" followed by a capital letter. This is an artifact of μC++ being a translator.

- μC++ allows at most 32 mutex members because a 32 bit mask is used to test for accepted member routines.

  We do not believe this causes practical problems in most programs. Further, this approach does not extend to support multiple inheritance. As is being discovered, multiple inheritance is not as useful a mechanism as it initially seemed [Car90], nor do we believe that the performance degradation required to support multiple inheritance is acceptable.

- When defining a derived type from a base type that is a task or coroutine and the base type has default parameters in its constructor, the default arguments must be explicitly specified if the base constructor is an initializer in the definition of the constructor of the derived type, for example:

```
uCoroutine Base {
  public:
    Base( int i, float f = 3.0, char c = 'c' );
};

uCoroutine Derived : public Base {
  public:
    Derived( int i ) : Base( i, 3.0, 'c' );          // values 3.0 and 'c' must be specified
};
```

  All other uses of the constructor for Base are *not* required to specify the default values. This is an artifact of μC++ being a translator.

- There is no discrimination mechanism in the uAccept statement to differentiate among overloaded mutex member routines. When time permits, a scheme that uses a formal declarer in the uAccept statement to disambiguate overloaded member routines will be implemented, for example:

```
uAccept( mem(int) );
uOr uAccept( mem(float) );
```

  Here, the overloaded member routines mem are completely disambiguated by the type of their parameters because C++ overload resolution does not use the return type.

- uWait and uSignal are statements rather than member routines of type uCondition because of limitations in the translator. Each uCondition variable must be initialized with a pointer to a unique internal lock object created implicitly as part of each mutex object. However, it is non-trivial to locate all instantiations of uCondition variables and initialize them with a pointer to the lock object. As well, the many different forms of initialization in C++ make this complex, assuming all instantiations could be located. Instead, it was simpler to pass the pointer to the lock object to implicit wait and signal members of a uCondition variable. For example, a statement like:

```
uWait DiskNotIdle;
```

  is translated into:

```
DiskNotIdle.uWait( lock-object );
```

  Alternatively, the lock object could have been passed as an implicit first parameter to member uWait, as in:

```
DiskNotIdle.uWait();
```

  would translate into:

```
DiskNotIdle.uWait( lock-object );
```

  However, the translator only builds a partial symbol table so it cannot determine if variable DiskNotIdle is a condition variable. Without this knowledge, the translator would have to transform any call to a member named uWait.

Chapter 3

μC++ Kernel

The μC++ kernel is a library of classes and routines that provide low-level light-weight concurrency support on uniprocessor and multiprocessor computers running the UNIX operating system. On uniprocessors, parallelism is simulated by rapid context switching at non-deterministic points so a programmer cannot rely on order or speed of execution. Many of the following facilities only have an effect on multiprocessor computers but can be called on a uniprocessor so that a program can be seamlessly transported between the two architectures.

The μC++ kernel does not call the UNIX kernel to perform a context switch or to schedule tasks, and uses shared memory for communication. As a result, performance for execution of and communication among large numbers of tasks is significantly increased over UNIX processes. The maximum number of tasks that can exist is restricted only by the amount of memory available in a program. The minimum stack size for an execution-state is machine dependent, but can be as small as 256 bytes. The storage management of all μC++ objects, the scheduling of tasks on virtual processors, and the pre-emptive round-robin scheduling to interleave task execution is performed by the μC++ kernel.

3.1 Pre-emptive Scheduling and Critical Sections

In general, the μC++ kernel and UNIX library routines are *not* reentrant. For example, many random number generators maintain an internal state between successive calls and there is no mutual exclusion on this internal state. Therefore, one task that is executing the random number generator can be pre-empted and the generator state can be modified by another task. This can result in problems with the generated random values or errors. One solution is to supply cover routines for each UNIX function, which guarantees mutual exclusion on calls. In general, this is not practical as too many cover routines have to be created.

Part of this problem can be handled by allowing pre-emption only in user code. When a pre-emption occurs, the handler for it checks if the interrupt location is within user code. If it is not, the interrupt handler resets the timer and returns without rescheduling another task. If the current interrupt point is in user code, the handler causes a context switch to another task. In theory, if the user code is always calling system routines (e.g. I/O routines), pre-emption may never occur.

Determining whether an address is in user code is done by relying on the loader to place programs in memory in a particular order. μC++ programs are compiled using a command that invokes the C++ compiler and includes all necessary include files and libraries. The program also brackets all user modules between two precompiled routines, uBeginUserCode and uEndUserCode, which contain no code. We then rely on the loader to load all object code in the order specified in the compile command. This results in all user code lying between the address of routines uBeginUserCode and uEndUserCode. The pre-emption interrupt handler simply checks if the interrupt address is between the address of uBeginUserCode and uEndUserCode to determine if the interrupt occurred in user code.

Unfortunately, this does not work when μC++ kernel routines are inlined into user code to reduce execution cost. This is handled by setting and resetting a critical section flag on entry and exit to inlined μC++ routines. This flag is also tested by the pre-emption interrupt handler to determine if the interrupt occurred in user code.

Allowing pre-emption only in user code is sufficient to deal with critical sections on uniprocessors. On multiprocessors, it is also necessary to arbitrate among the multiple tasks attempting to access a critical section. This is done using spin locks built from atomic instructions. Once a task has acquired a spin lock and entered a critical section, it may still be pre-empted during the execution of the critical section. If the critical section had called a UNIX library routine when the pre-emption occurred, the pre-emption would be disallowed; otherwise, another task is executed and the critical section remains locked.

3.2 User Specified Context

The following facilities allow users to specify additional coroutine and task context to be saved and restored during a context switch. This facility should only be used to save and restore processor specific data, for example, co-processor or graphics hardware data that is specific to each processor's execution. This facility does *not* allow a shared resource, like a single graphics device, to be accessed mutually exclusively by multiple tasks in a multiprocessor environment. In a multiprocessing environment, tasks executing in parallel will corrupt the shared resource because their context switches overlap. To share a resource in a multiprocessor environment requires proper mutual exclusion, for example, by using a server task. In a uniprocessor environment, this facility can be used to guarantee mutual exclusion to a shared resource because only one task is executing at a time so the context of the shared resource is saved and restored on each context switch. We *strongly* discourage using this facility for mutual exclusion of a non-processor specific resource because it does not scale to a multiprocessor environment.

The user-context facility has two parts: the definition of a context save-area, containing the storage for the context and routines to save and restore the context, and the declaration and initialization of a context save-area. The association of the additional context with a coroutine or task depends on which execution-state is in use when the declaration of the context save-area occurs.

A context area *must* be derived from the abstract class uContext:

```
class uContext {
  public:
    uContext( void *context, void (*save)( void * ), void (*restore)( void * ) );
};
```

The constructor routine uContext has the following parameters:

context is a pointer to a data area in which a context is saved and restored.

save is a free routine that is passed the address of the user context area and which stores the context into this area.

restore is a free routine that is passed the address of the user context area and which restores the context from this area.

Multiple context areas can be declared, and hence, associated with a coroutine or task. Context areas are only associated with an execution state if the address of the save and restore routines are unique. This prevents the same context from being associated multiple times with a particular coroutine or task. Figure 3.1 shows how the context of a hardware co-processor can be saved and restored as part of the context of task worker.

3.2.1 Floating Point Context

In most systems, e.g. UNIX, the entire state of the actual processor is saved during a context switch between execution-states because there is no way to determine if a particular object is using only a subset of the actual processor state. All objects use the fixed-point registers, while only some use the floating-point registers. Because there is a significant execution cost in saving and restoring the floating-point registers, they are not saved automatically. If a coroutine or task performs floating-point operations, saving the floating-point registers must become part of the context-switching action for the execution-state of that coroutine or task.

```
class CoProcessorCxt : public uContext {
    // use static routines so global name space is not polluted
    static void Save( void *context );
    static void Restore( void *context );
    int reg[3];                          // co-processor has 3 integer registers
  public:
    CoProcessorCxt() : uContext( reg, Save, Restore ) {};
};

void CoProcessor::Save( void *context ) {
    // assembler code to save co-processor registers into context area
}

void CoProcessor::Restore( void *context ) {
    // assembler code to restore co-processor registers from context area
}

uTask worker {
    ...
    void main() {
        CoProcessorCxt cpcxt;            // associate additional context with task
        ...
    }
    ...
};
```

Figure 3.1: Saving Co-processor Context

To cause this action to occur, declare a single instance of type uFloatingPointContext in the scope of the floating-point computations, such as the beginning of the coroutine's or task's main member, as in:

```
uCoroutine C {
    void main() {
        uFloatingPointContext fpcxt;     // the name of the variable is not significant
        ...          // floating-point computations can be performed safely in this scope
    }
    ...
};
```

Once main starts, both the fixed-point and floating-point registers are restored or saved during a context switch to or from instances of coroutine C.

> □ **WARNING:** The member routines of a coroutine or task are executed using the execution-state of the caller. Therefore, if floating point operations occur in a member routine, including the constructor, the caller must also save the floating-point registers. Only a coroutine's or task's main routine and the routines called by main use the coroutine's or task's execution state, and therefore, only these routines can safely perform floating point operations. □

Additional context can be associated with a coroutine or task in a free routine, member routine, or as part of a class instance to temporarily save a particular context. For example, the floating-point registers are saved when an instance of the following class is declared:

```
class c {
  private:
    uFloatingPointContext fpcxt;
  public:
    void func() {
        // perform floating-point computations
    }
};
```

When a coroutine or task declares an instance of c, its context switching is augmented to save the floating-point registers for the duration of the instance. This allows the implementor of c to ensure that the integrity of its floating point calculations are not violated by another coroutine or task performing floating point operations. It also frees the user from having to know that the floating-point registers must be saved when using class c. If the floating-point registers are already being saved, the additional association is ignored.

3.3 Explicit Mutual Exclusion and Synchronization

The following locks are low-level mechanisms for providing mutual exclusion of critical sections and synchronization among tasks. In general, explicit locks are unnecessary to build highly concurrent systems; the mutual exclusion provided by monitors, coroutine-monitors and tasks, and the synchronization provided by uAccept, uWait and uSignal are sufficient. Nevertheless, several low-level lock mechanisms are provided for teaching purposes and for building special experiments.

3.3.1 Counting Semaphore

A semaphore in μC++ is implemented as a counting semaphores as described by Dijkstra [Dij68]. A counting semaphore has two parts: a counter and a list of waiting tasks. Both the counter and the list of waiting tasks is managed by the semaphore.

The type uSemaphore defines a semaphore:

```
class uSemaphore {
  public:
    uSemaphore( int value = 0 );
    void uP( int times = 1 );
    void uV( int times = 1 );
    int uEmpty();
};

uSemaphore x, y(1), *z;
z = new uSemaphore(4);
```

This creates three variables that are semaphores and initializes them to the value 0, 1, and 4, respectively.

The constructor routine uSemaphore has the following form:

uSemaphore(int value) – this form specifies an initialization value for the semaphore counter. Appropriate count values are values ≥ 0. The default value is 0.

The public member routines uP and uV are used to perform the classical counting semaphore operations. uP decrements the semaphore counter if the value of the semaphore counter is greater than zero and continues; if the semaphore counter is equal to zero, the calling task blocks. If uP is passed a positive integer value, the semaphore is Ped that many times. The overloaded routine uV wakes up the task blocked for the longest time if there are tasks blocked on the semaphore and increments the semaphore counter. If uV is passed a positive integer value, the semaphore is Ved that many times. The public member routine uEmpty() returns 0 if there are threads blocked on the semaphore and 1 otherwise.

It is *not* meaningful to read or to assign to a semaphore variable, or copy a semaphore variable (e.g. pass it as a value parameter).

□ wait and signal operations on conditions are very similar to P and V operations on counting semaphores. The wait statement can block a task's execution while a signal statement can cause resumption of another task. There are, however, differences between them. The P operation does not necessarily block a task, since the semaphore counter may be greater than zero. The wait statement, however, always blocks a task. The signal statement can make ready a blocked task on a condition just as a V operation makes ready a blocked task on a semaphore. The difference is that a V operation always increments the semaphore counter; thereby affecting a subsequent P operation. A signal statement on an empty condition does not affect a subsequent wait statement. Another difference is that multiple tasks blocked on a semaphore can resume execution without delay if enough V operations are performed. In the mutex type case, multiple signal statements do unblocked multiple tasks, but only one of the these tasks will be able to execute because of the mutual exclusion property of the mutex type. □

3.3.2 Owner Lock

An owner lock is owned by the task that acquires it; all other tasks wanting the lock wait until the owner releases it. The owner of an owner lock can acquire the lock multiple times, but a matching number of releases must occur or the lock remains in the owners possession and other task cannot acquire it.

The type uOwnerLock defines an owner lock:

```
uMonitor uOwnerLock {
  public:
    uOwnerLock();
    void uAcquire();
    void uRelease();
};

uOwnerLock x, y, *z;
z = new uOwnerLock();
```

This creates three variables that are owner locks and initializes them to open.

The public member routines uAcquire and uRelease are used to atomically acquire and release the owner lock, respectively. uAcquire acquires the lock if it is not currently owned, otherwise the calling task waits until the lock is released by the current owner. uRelease releases the lock.

It is *not* meaningful to read or to assign to a lock variable, or copy a lock variable (e.g. pass it as a value parameter).

□ Owner locks are used in the implementation of the non-blocking I/O stream library. □

3.3.3 Lock

A lock is either open or closed and tasks compete to acquire the lock after is has been released. Unlike a semaphore, which blocks tasks that cannot continue execution immediately, a lock may allow tasks to loop attempting to acquire the lock (busy wait). Locks do not ensure that tasks competing to acquire it are served in any particular order; in theory, starvation can occur, in practice, it is not a problem.

The type uLock defines a lock:

```
class uLock {
  public:
    uLock();
    void uAcquire();
    void uRelease();
};

uLock x, y, *z;
z = new uLock();
```

This creates three variables that are locks and initializes them to opened.

The public member routines uAcquire and uRelease are used to atomically acquire and release the lock, closing and opening it, respectively. uAcquire acquires the lock if it is open, otherwise the calling task spins waiting. uRelease releases the lock, which allows any waiting tasks to race to acquire the lock.

It is *not* meaningful to read or to assign to a lock variable, or copy a lock variable (e.g. pass it as a value parameter).

3.3.4 Barrier

future work

3.4 Memory Management

All data that μC++ manipulates must reside in memory that is accessible by all UNIX processes started by μC++. In the unikernel case, there is a single address space accessed by the process that owns it. In the multikernel case, several address-spaces exist, one for each UNIX process. These address-spaces have private memory accessible only by a single process and shared memory that is accessible by all the UNIX processes. In μC++, all user data is located in the shared memory of the UNIX processes.

In order to make memory management operations portable across both versions of μC++ and make memory sharable, the memory management operators new and delete are redefined to allocate and free memory correctly for each version of μC++. These free operators provide identical functionality to the C++ operators new and delete. Further, the μC++ versions of these memory management operators provide mutual exclusion on calls to them.

If direct access is required to the system routines malloc and free, they can be accessed through the cover routines uMalloc and uFree, for example:

```
struct fred *fp = uMalloc( sizeof(fred) );
...
uFree( fp );
```

These free routines ensure memory returned is sharable and calls to them are mutually exclusive.

3.5 Program Termination

To terminate a program with a status code to the invoking shell, use the free routine uExit:

```
void uExit( int status );
```

This routine is the same as the UNIX routine _exit, except that it works correctly in both unikernel and multikernel. Notice, that global destructors are *not* invoked when exiting because there may be outstanding local coroutines and tasks that would generate errors from kernel destructors.

> □ When the routine uMain::main terminates, the current rule is that *all* other tasks are automatically terminated. It is not possible to start tasks that continue to execute after uMain::main terminates. Therefore, uMain::main must only terminate when the entire application program has completed. This rule was chosen because we found that managing multiple UNIX processes running independently of uMain::main required too much knowledge from novice users. However, there is nothing in the design of μC++ that precludes supporting this feature at some future time. □

3.6 Errors

Errors in μC++ are divided into the following categories:

- The mandatory way to stop all execution and print an error message while running within μC++ is to call the free routine uAbort. The UNIX routine abort is designed for a single process program and does not work as expected in the multikernel.

  The routine uAbort prints a user specified string, which is presumably a message describing the error, and then prints the name of the currently executing task type, possibly naming the type of the currently executing coroutine if the task's thread is not executing on its own execution-state at the time of the call.

  ```
  void uAbort( char *format, ... )
  ```

  format is a format string containing text to be printed and printf style format codes which describe how to print the following variable number of arguments.

  ... is a list of arguments to be formatted and printed on standard output. The number of elements in this list must match with the number of format codes.

- A user task executes some code that causes the virtual processor to fault. The death of the UNIX process is caught by a task executing on the parent process of the terminating process. In general, this task calls routine uAbort. For example, if a task tries to divide by zero or access memory out of the address space currently available to the application, these errors are trapped. In such situations, the UNIX signal number of the terminating process is displayed in the error message. Hence, when μC++ displays a message saying that a UNIX process died, the cause of that UNIX process's death can be determined.

Errors in I/O usually produce a return code from the particular I/O routine. A detailed reason for the error may be placed in a global variable called errno. To facilitate the traditional UNIX style of testing return codes and examining errno, errno is made part of an execution-state. Hence, after an I/O operation, it is safe to examine errno as it will not be changed by another coroutine or task executing an I/O operation.

3.7 Cluster

A cluster is a collection of μC++ tasks and processors; it provides a runtime environment for a task's execution. This environment contains a number of variables that can be modified to affect how coroutines, tasks and processors behave in a cluster. These environment variables are used implicitly, unless overridden, when creating an execution-state on a cluster and by processors associated with that cluster:

stack size is the default stack size, in bytes, used when coroutines or tasks are created on a cluster.

time slice duration is the longest time, in milliseconds, a task on this cluster can hold a processor before a switch to another task is attempted.

spin duration is the number of checks of the cluster's ready queue that an idle processor performs before the processor goes to sleep.

Each of these variables is either explicitly set or implicitly assigned a μC++-dependent default value when the cluster is created.

3.7.1 Cluster Default Values

1¡ The μC++ default values are initially set to reasonable values for the particular CPU/OS combination on which μC++ is running. When a cluster is created, it is these machine-dependent default values that initialize the cluster variables unless overridden in the declaration. The default values come from 3 free routines:

```
int uDefaultStackSize();     // returns the default stack size for this application
int uDefaultTimeSlice();     // returns the default time slice for this application
int uDefaultSpin();          // returns the default spin time for this application
```

which are called by a cluster during its initialization.

The default routines can be replaced by the user simply by defining a routine with the same name in the application, as in:

```
int uDefaultStackSize() {
    return 64 * 1024;                    // 64K default stack size
}
```

If the value of a global variable is returned instead of a constant, the application can change the application default value dynamically; clusters created at different times would be initialized with different default values (unless overridden on the declaration of the cluster). However, the global variable must be statically initialized because its value is used for the stack size of task uMain, which is created *before* any user code is executed in uMain::main.

3.7.2 Cluster Type and Objects

The cluster's constructors and member routines provide the ability to set and read the cluster environment variables:

```
class uCluster {
  public:
    uCluster( int stackSize = uDefaultStackSize(), int timeSlice = uDefaultTimeSlice(),
                    int spinTime = uDefaultSpin(), const char *name = "" );
    uCluster( const char *name );

    void uSetStackSize( int stackSize );
    int uGetStackSize();
    void uSetTimeSlice( int timeSlice );
    int uGetTimeSlice();
    void uSetSpin( int spinTime );
    int uGetSpin();
    void uSetName( const char *name );
    const char *uGetName() const;
};

uCluster clus( 8196, 0 ) // 8K default stack size, 0 time slice duration
```

The overloaded constructor routine uCluster has the following forms:

uCluster(int stackSize = uDefaultStackSize(), int timeSlice = uDefaultTimeSlice(), int spinTime = uDefaultSpin(), const char *name = "") – this form uses the user specified stack size, time-slice duration, processor-spin duration and cluster name. If no stack size or time-slice duration or processor-spin are given, they are specified by the current cluster's default values, a machine dependent value obtained from the appropriate free routines. If no cluster name is given, the name is set to the NULL string;

uCluster(const char *name) – this form uses the user specified name for the cluster and the current cluster's default values for stack size, time-slice duration and processor-spin duration.

As stated previously, there are two versions of the μC++ kernel: the unikernel, which is designed to use a single processor (the system, user and any other clusters are automatically combined); and the multikernel, which is designed to use several processors. While the interfaces to the unikernel and multikernel are identical, there are several differences between them, which all result from the unikernel having only one virtual processor. In particular, the semantics of the cluster are different for each kernel. In the unikernel, operations to increase or decrease the number of virtual processors are ignored. Further, while a new cluster instance is created, it refers to the initial cluster; hence, the system and user clusters are combined into a single cluster. The uniform interface allows almost all concurrent applications to be designed and tested on the unikernel, and then run on the multikernel after re-linking.

3.7.3 Cluster Creation and Destruction

A cluster object contains the values of the cluster environment variables and a set of processors that are associated with the cluster. A number of routines are available to modify a cluster's environment variables. A cluster can be used in operations like task creation to specify the cluster on which the task is to be created. After a cluster is created, it the users responsibility to associate at least one processor with the cluster so that it can execute tasks.

When a cluster terminates, it must have no tasks executing on it and all processors associated with it must be freed. It is the user's responsibility to ensure that no tasks are executing on a cluster when it terminates; therefore, a cluster can only be destroyed by a task on another cluster. If tasks are executing on a cluster when it is destroyed, they block and are inaccessible.

The free routine:

```
uCluster &uThisCluster();
```

is used to determine the identity of the current cluster a task resides on.

3.7.4 Default Stack Size

The public member routine uSetStackSize is used to set the default stack size value for the stack portion of each execution-state allocated on a cluster. The new stack size is specified in bytes. For example, the call clus.uSetStackSize(8000) sets the default stack size to 8000 bytes.

The public member routine uGetStackSize is used to read the value of the default stack size for a cluster. For example, the statement i = clus.uGetStackSize() sets i to the value 8000.

3.7.5 Implicit Task Scheduling

Pre-emptive scheduling is enabled by default on both unikernel and multikernel. Each processor is periodically interrupted in order to schedule another task to be executed. Note that interrupts are not associated with a task but with a processor; hence, a task does not receive a time slice and it may be interrupted immediately after starting execution because the processor's time slice ended and another task is scheduled. A task is pre-empted at a non-deterministic location in its execution when the processor's time-slice expires. All processors on a cluster have the same time slice but the interrupts are not synchronized. The default processor time-slice is machine dependent but is approximately 0.1 seconds on most machines. The effect of this pre-emptive scheduling is to simulate parallelism. This simulation is usually accurate enough to detect most situations on a uniprocessor where a program may depend on the order or the speed of execution of tasks.

The public member routine uSetTimeSlice is used to set the default time slice for each processor on the current cluster. The new time duration between interrupts is specified in milliseconds. For example, the call clus.uSetTimeSlice(50) sets the default time slice to 0.05 seconds for each processor on this cluster. To turn pre-emption off, call clus.uSetTimeSlice(0).

> □ On many systems the minimum time slice may be 10 milliseconds (0.01 of a second). Setting the duration to an amount less than this simply sets the interrupt time interval to this minimum value. □

> □ The overhead of pre-emptive scheduling depends on the frequency of the interrupts. Further, because interrupts involve entering the UNIX kernel, they are relatively expensive if they occur frequently. We have found that an interrupt interval of 0.05 to 0.1 seconds gives adequate concurrency and increases execution cost by less than 1% for most programs. □

The public member routine uGetTimeSlice is used to read the current default time slice for a cluster. For example, the statement i = clus.uGetTimeSlice() sets i to the value 50.

3.7.6 Idle Virtual Processors

When there are no ready tasks for a processor to execute, the idle processor has to spin in a loop or sleep or both. In the μC++ kernel, an idle processor spins for a user specified number of checks of the cluster's ready queue before it sleeps. During the spinning, the processor is constantly checking for ready tasks, which would be made ready by other processors. An idle processor is ultimately put to sleep so that machine resources are not wasted. The reason that the idle processor spins is because the sleep/wakeup time can be large in comparison to the execution of tasks in a particular application. If an idle processor goes to sleep immediately upon finding no ready tasks, the next executable task will have to wait for completion of a UNIX system call to restart the processor. If the idle processor spins for a short period of time any task that becomes ready during the spin duration will be processed immediately. Selecting a spin time is application dependent and it can have a significant affect on performance for certain applications.

The public member routine uSetSpin is used to set the default spin-duration for each processor on the current cluster. The new spin duration is specified in the number of times the cluster's ready queue is checked for an available task to execute. For example, the call clus.uSetSpin(50000) sets the default spin-duration to 50000 checks for each processor on this cluster. To turn spinning off, call clus.uSetSpin(0).

The public member routine uGetSpin is used to read the current default spin-duration for a cluster. For example, the statement i = clus.uGetSpin() sets i to the value 50000.

3.8 Processors

A μC++ virtual processor is a "software processor" that executes threads. A virtual processor is implemented as a UNIX process (or kernel thread) that is subsequently scheduled for execution on a hardware processor by the underlying operating system. On a multiprocessor, UNIX processes are usually distributed across the hardware processors and so some UNIX processes are able to execute in parallel. The maximum number of virtual processors that can be created is indirectly limited by the number of UNIX processes a UNIX user can create, as the sum of the virtual processors on all clusters cannot exceed this limit.

The processor's constructors allow it to be created with a runtime environment different from the cluster's environment variables.

```
class uProcessor {
  public:
    uProcessor();
    uProcessor( int timeSlice );
    uProcessor( int timeSlice, int spinTime );
    uProcessor( uCluster &cluster );
    uProcessor( uCluster &cluster, int timeSlice );
    uProcessor( uCluster &cluster, int timeSlice, int spinTime );

    void uSetTimeSlice( int timeSlice );
    int uGetTimeSlice() const;
    void uSetSpin( int spinTime );
    int uGetSpin() const;
};

uProcessor proc( clus );  // processor is attached to cluster clus
```

The overloaded constructor routine uProcessor has the following forms:

uProcessor() – creates the processor on the current cluster with that cluster's default time-slice and processor-spin duration.

uProcessor(int timeSlice) – creates the processor on the current cluster with the user specified time-slice duration and the current cluster's processor-spin duration.

uProcessor(int timeSlice, int spinTime) – creates the processor on the current cluster with the user specified time-slice and processor-spin duration.

uProcessor(uCluster &cluster) – creates a processor on the specified cluster using that cluster's default time slice and spin duration.

uProcessor(uCluster &cluster, int timeSlice) – creates a processor on the specified cluster with the user specified time-slice duration and that cluster's processor-spin duration.

uProcessor(uCluster &cluster, int timeSlice, int spinTime) – creates a processor on the specified cluster using the user specified time-slice and processor-spin duration.

The routines uSetTimeSlice, uGetTimeSlice, uSetSpin and uGetSpin set and return the individual processor's environment values.

The free routine:

```
uBaseProcessor &uThisProcessor();
```

is used to determine the identity of the current processor a task is executing on.

The following are points to consider when deciding how may processors to create for a cluster. First, there is no advantage in creating significantly more processors than the average number of simultaneously active tasks on the cluster. For example, if on average three tasks are eligible for simultaneous execution, creating significantly more than three processors will not achieve any execution speed up and wastes resources. Second, the processors of a cluster are really virtual processors for the hardware processors, and there is usually a performance penalty in creating more virtual processors than hardware processors. Having more virtual processors than hardware processors can result in extra context switching of the heavy-weight UNIX processes used to implement a virtual processor, which is runtime expensive. This same problem can occur among clusters. If a computational problem is broken into multiple clusters and the total number of virtual processors exceeds the number of hardware processors, extra context switching of the UNIX processes will occur. Finally, a μC++ program usually shares the hardware processors with other user programs. Therefore, the overall UNIX system load will affect how many processors should be allocated to avoid unnecessary context switching of UNIX processes.

> □ Changing the number of processors is expensive, since a request is made to UNIX to allocate or deallocate UNIX processes or kernel threads. This operation often takes at least an order of magnitude more time than task creation. Further, there is often a small maximum number of UNIX processes (e.g. 20–40) that can be created by a UNIX user. Therefore, processors should be created judiciously, normally at the beginning of a program. □

3.8.1 Blocking Virtual Processors

To ensure maximum parallelism, it is desirable that a task not execute an operation that causes the processor it is executing on to block. It is also essential that all processors in a cluster be interchangeable, since task execution may be performed by any of the processors of a cluster. When tasks or processors cannot satisfy these conditions, it is essential that they be grouped into appropriate clusters in order to avoid adversely affecting other tasks or guarantee correct execution. Each of these points will be examined.

There are two forms of blocking that can occur in μC++:

heavy blocking which is done by UNIX on a virtual processor as a result of certain system requests (e.g. I/O operations).

light blocking which is done by the μC++ kernel on a task as a result of certain μC++ operations (e.g. uAccept, uWait and calls to a mutex routine).

The problem is that heavy blocking removes a virtual processor from use until the operation is completed; for each virtual processor that blocks, the potential for parallelism decreases on that cluster. In those situations where maintaining a constant number of virtual processors for computation is desirable, tasks should block lightly rather than heavily. This can be accomplished by keeping the number of tasks that block heavily to a minimum and also relegated to a separate cluster. This can be accomplished in two ways. First, tasks that would otherwise block heavily instead make requests to a task on a separate cluster which then blocks heavily. Second, tasks migrate to the separate cluster and perform the operation that blocks heavily. This maintains a constant number of virtual processors for concurrent computation in a computational cluster, such as the user cluster.

On some multiprocessor computers not all hardware processors are equal. For example, not all of the hardware processors may have the same floating-point units; some units may be faster than others. Therefore, it may be necessary to create a cluster whose processors are attached to these specific hardware processors. (The mechanism for attaching virtual processors to hardware processors is operating system specific and not part of μC++. For example, the Dynix operating system from Sequent provides a routine tmp_affinity to lock a UNIX process on a processor.) All tasks that need to perform high-speed floating-point operations can be created/placed on this cluster. This still allows tasks that do only fixed-point calculations to continue on another cluster, potentially increasing parallelism, but not interfering with the floating-point calculations.

Chapter 4

Input/Output

4.1 Interaction with the UNIX File System

As explained in Section 3.7.2, it is desirable to avoid heavy blocking of virtual processors. UNIX I/O operations can be made to be nonblocking, but this requires special action as the I/O operations do not restart automatically when the operation completes. Instead, it is necessary to perform polling for I/O completions and possibly blocking of the UNIX process if all tasks are directly or indirectly blocked waiting for I/O operations to complete. To simplify the complexity of performing nonblocking I/O operations, μC++ supplies a library of I/O routines that perform the nonblocking operations and polling (detailed below).

4.1.1 Unikernel File Operations

To retain concurrency in the unikernel during I/O operations, the μC++ I/O routines check the ready queue before performing their corresponding UNIX I/O operation. If there are no tasks waiting to execute, the single virtual processor is blocked because all tasks in the system must be directly or indirectly waiting for an I/O operation to complete. If there are tasks to execute, a nonblocking I/O operation is performed. A task performing an I/O operation is chosen to polling for completion of any I/O operation, yielding control of the processor if no I/O operation has completed. When an I/O operation completes, the task waiting for that operation is unblocked and if necessary another task performing an I/O operation is chosen to poll for completion of the next I/O operation. This scheme allows other tasks to progress with only a slight degradation in performance due to the polling task.

Unfortunately, UNIX does not provide adequate facilities to ensure that signals sent to wake up sleeping UNIX processes are always delivered. There is a window between sending a signal and blocking using a UNIX select operation that cannot be closed. Therefore, each processor blocked waiting for I/O completion polls once a second for the rare event that a signal sent to wake it up was missed.

4.1.2 Multikernel File Operations

In the multikernel, not only is there the blocking I/O problem, but some UNIX systems associate the internal information needed to access a file (i.e. a file descriptor) with a virtual processor (i.e. UNIX process) in a non-shared way. This means that if a task opens a file on one virtual processor it will not be able to read or write the file if the task is scheduled for execution on another virtual processor. Both problems can be solved by creating a separate cluster that has a single virtual processor containing the file descriptor. Any task that wants to access the file migrates to the I/O cluster to perform the operation. In this manner, a task performing an I/O operation can access the private UNIX file descriptor.

In detail, a cluster with one processor is automatically created when a file is opened. Hence, each open file has a corresponding UNIX process. The exception to this rule is the standard I/O files, called uCin, uCout and uCerr in μC++, which are opened implicitly on the system cluster. A user task performs I/O operations by executing the equivalent μC++ cover routine, which migrates the task to the cluster containing the file descriptor and performs the appropriate operation. When the user task closes the file, the cluster and all of its resources are released. To ensure that multiple tasks are not performing I/O operations simultaneously on the same file descriptor, each μC++ file descriptor is implemented as a monitor that provides mutual exclusion of I/O operations. Figure 4.1 illustrates the runtime structures created for accessing a file. Depending on the kind of I/O, there may be one or several tasks on the I/O cluster.

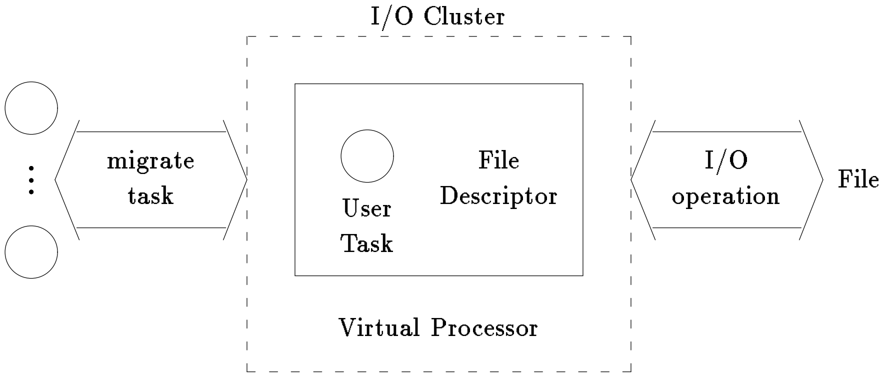

Figure 4.1: UNIX File I/O Cluster

An I/O cluster is created for each open file. Therefore, if a file is opened multiple times, each opening creates a new and independent cluster, processor and file descriptor that refers to the same file on disk. Access across these I/O clusters is not serialized, while access to a particular I/O cluster is serialized. This is not a problem if all tasks are reading the file, but will not work, in general, if some tasks read and some write to the same file on different I/O clusters to the same file on disk.

□ **WARNING**: μC++ introduces substantial overhead for each I/O call because of the non-blocking I/O. Therefore, always read and write in large blocks (e.g. 4K-8K) when possible. □

4.2 μC++ Stream Library

The following interface has been provided to use C++ streams. The standard C++ stream objects cin, cout and cerr have derived objects uCin, uCout and uCerr, respectively. These μC++ objects behave identically to the C++ stream objects, and in addition each I/O operation is performed mutually exclusively and in a nonblocking manner. To use the interface, include the file:

```
#include <uIOStream.h>
```

at the beginning of each source file.

Mutual exclusion on streams ensures that characters being generated by the insertion operator (<<) and/or the extraction operator >> in different tasks are not interspersed; hence, each execution of the operators << and >> is atomic. However, this does not prevent the results of multiple insertion and extraction operators from being interspersed. For example, if two tasks execute the following:

$task_1$
```
uCout << "abc " << "def ";
```

$task_2$
```
uCout << "uvw " << "xyz ";
```

some of the different outputs that could appear are:

```
abc def uvw xyz
abc uvw def xyz
abc uvw xyz def
uvw abc def xyz
uvw abc xyz def
```

To ensure that I/O output is not interspersed, a stream may be explicitly acquired and released using the manipulator routines uAcquire and uRelease, as in:

$task_1$
```
uCout << uAcquire << "abc " << "def " << uRelease;
```

$task_2$
```
uCout << uAcquire << "uvw " << "xyz " << uRelease;
```

which can produce only two different outputs:

```
abc def uvw xyz
uvw xyz abc def
```

Once a task has acquired the I/O lock for a stream, it owns the stream until it unlocks it. Therefore, multiple I/O statements can be performed atomically, as in:

```
uCout << uAcquire;              // acquire the lock for stream uCout
uCout << "abc";
uCout << "def";
uCout << uRelease;              // release the lock for stream uCout
```

□ **WARNING:** Deadlock can occur if routines are called in an I/O sequence that might block, as in:

```
uCout << uAcquire << "data:" << Monitor.rtn(...) << endl << uRelease;
```

The problem occurs if the task executing the I/O sequence blocks in the monitor when it is holding the I/O lock for stream uCout. Any other task that attempts to write on uCout will block until the task holding the lock is unblocked and releases it. This can lead to deadlock if the task that is going to unblock the task waiting in the monitor first writes to uCout. One simple precaution is to factor the call to the monitor routine out of the I/O sequence, as in:

```
int data = Monitor.rtn(...);
uCout << uAcquire << "data:" << data << endl << uRelease;
```

□

4.2.1 uIStream and uOStream I/O

The types uIStream and uOStream are similar to the C++ classes istream and ostream, with the following exceptions:

- All of the member routines have mutual exclusion through an owner lock so a task controls the stream lock.
- Additional manipulator routines are defined for controlling mutual exclusion of the stream across a sequence of operations:

  ```
  uIStream &uAcquire( uIStream & );
  uIStream &uRelease( uIStream & );
  uOStream &uAcquire( uOStream & );
  uOStream &uRelease( uOStream & );
  ```

- The stream member routine:

  ```
  uOStream &form( const char *fmt, ... );
  ```

  is not available for uOStream because there is no way to pass the cover routine's ellipse argument to the corresponding base class routine's ellipse parameter (GNU only).

4.2.2 uIFStream and uOFStream I/O

The types uIFStream and uOFStream are similar to the C++ classes ifstream and ofstream, with the same exceptions as for types uIStream and uOStream. To use the uFStream I/O interface, include the file:

```
#include <uFStream.h>
```

at the beginning of each source file.

4.3 UNIX File I/O

The following interface has been provided to use UNIX files. To use the interface, include the file:

```
#include <uFile.h>
```

at the beginning of each source file. uFile.h also includes the following UNIX system files: <sys/types.h>, <sys/file.h>.

A file is a passive object that has information written into and read from it by tasks; therefore, a file is like a monitor, which provides indirect communication among tasks. The difference between a file and a monitor is that the file is on secondary storage, and hence, is not directly accessible by the computer's processors; a file must be made explicitly accessible before it can be used in a program. Furthermore, a file may have multiple accessors—although it is up to UNIX to interpret the meaning of these potentially concurrent accessors—so there is a many to one relationship between a file and its accessors. This relationship is represented in a μC++ program by a declaration for a file and subsequent declarations for each accessor.

Traditionally, access to a file is explicit and is achieved procedurally by a call to "open" and a subsequent call to "close" to terminate the access. In μC++, the declaration of a special **access object** performs the equivalent of the traditional open and its deallocation performs the equivalent of the traditional close. In many cases, the access object is a local variable so that the duration of access is tied to the duration of its containing block. However, by dynamically allocating an access object and passing its pointer to other blocks, the equivalent access duration provided by traditional "open" and "close" can be achieved.

In μC++, a connection to a UNIX file is made by declaration of a uFile object, as in:

```
uFile infile( "abc" ), outfile( "xyz" );
```

which creates two connection variables, infile and outfile, connected to UNIX files abc and xyz, respectively. The operations available on a file object are:

```
class uFile {
  public:
    uFile( const char *name );
    void status( struct stat *buf );
};
```

The parameter for the constructor uFile is the UNIX path name of the file, which is connected to the program. It is *not* meaningful to read or to assign to a uFile object, or copy a uFile object (e.g. pass it as a value parameter).

The parameter for member status is explained in the UNIX manual entry for stat. (The first parameter to the UNIX stat routine is unnecessary, as it is provided implicitly by the uFile object.) Because the file object is not accessible after the connection is made, there are no member routines to access its contents.

4.3.1 File Access

Once a connection has been made to a UNIX file, its contents can be accessed by an access object, as in:

```
uFileAccess input( infile, O_RDONLY ), output( outfile, O_CREAT | O_WRONLY );
```

which creates one access object to read from the connection to file abc and one object to write to the connection made to file xyz. The operations available on an access object are:

```
class uFileAccess {
  public:
    uFileAccess( uFile &f, int flags, int mode = 0644 );
    int read( void *buf, int len );
    int write( void *buf, int len );
    off_t lseek( off_t offset, int whence );
    int fsync();
};
```

The parameters for the constructor uFileAccess are as follows. The f parameter is a uFile object that is to be opened for access. The flags and mode parameters are explained in the UNIX manual entry for open. It is *not* meaningful to read or to assign to a uFileAccess object, or copy a uFileAccess object (e.g. pass it as a value parameter).

The parameters and return value for members read, write, lseek and fsync are explained in their corresponding UNIX manual entries. (The first parameter to these UNIX routines is unnecessary, as it is provided implicitly by the uFileAccess object.) Appendix B.4 shows reading and writing to UNIX files.

4.4 BSD Sockets

The following interface has been provided to use BSD sockets. To use the interface in a μC++ program, include the file:

```
#include <uSocket.h>
```

at the beginning of each source file. uSocket.h also includes the following UNIX system files: <sys/types.h>, <sys/file.h>, <sys/socket.h>.

A socket is an end points in bidirectional communicate among tasks in different UNIX processes. A socket endpoint is accessed either as a client, which is one to one with its socket endpoint, or as a server, which is many to one with its socket endpoint (like a file and its accessors). A server may accept connections from multiple clients, creating an acceptor object to deal with each client's communication. Therefore, a socket provides either a connection capability for a client to a server or a de-multiplexing capability for a server to manage its multiple clients. This relationship is shown in Figure 4.2. A server socket has a name (a character string and/or port number) that clients must know to connect to the server socket. A task using a server socket is normally in a loop accepting connections from clients. Each acceptance creates an acceptor object, which can be passed to a worker task to perform the communication with the client in parallel with the server accepting more clients. These relationships are represented in a μC++ program by declarations of client, server and acceptor objects, respectively.

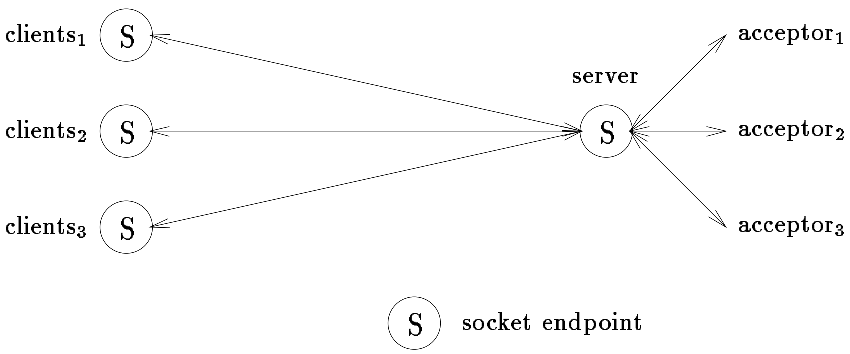

Figure 4.2: Client/Server Sockets

4.4.1 Client

In μC++, a client, its socket endpoint and a connection to a server are made by declaration of a uSocketClient object, for example:

```
uSocketClient client( "abc" );
```

which creates a client variable, client, connected to the UNIX server socket abc. The operations provided by uSocketClient are:

```
class uSocketClient {
  public:
    uSocketClient( char *name, int domain = AF_UNIX, int type = SOCK_STREAM, int protocol = 0 );
    uSocketClient( unsigned short port, char *name, int type = SOCK_STREAM, int protocol = 0 );
    int getsockname( struct sockaddr *name, int *len );
    int getpeername( struct sockaddr *name, int *len );
    int read( void *buf, int len );
    int write( void *buf, int len );
    int send( void *buf, int len, int flags );
    int sendto( void *buf, int len, int flags, void *to, int tolen );
    int sendmsg( void *msg, int flags );
    int recv( void *buf, int len, int flags );
    int recvfrom( void *buf, int len, int flags, void *from, int *fromlen );
    int recvmsg( void *msg, int flags );
};
```

The parameters for the first constructor uSocketClient are as follows. The name parameter to uSocketClient is the name of an existing UNIX socket that the client is connecting to. The domain, type and protocol parameters are explained in the UNIX manual entry for socket; they specify the type of the server socket and control the kind of communication with the server. The parameters for the second constructor of uSocketClient are as follows. The port parameter is the port number of an INET port on a host machine specified by the name parameter. The domain parameter is always AF_INET and the remaining parameters are the same as above. It is *not* meaningful to read or to assign to a uSocketClient object, or copy a uSocketClient object (e.g. pass it as a value parameter).

The parameters and return value for the communication members are explained in their corresponding UNIX manual entries. (The first parameter to these UNIX routines is unnecessary, as it is provided implicitly by the uSocketClient object.) Appendix B.5.1 shows a client connecting to a server and communicating with an acceptor.

4.4.2 Server

In μC++, a server and its socket endpoint are created by declaration of a uSocketServer object, for example:

```
uSocketServer server( "abc" );
```

which creates a server variable, server, and a UNIX socket endpoint named abc. The operations provided by uSocketServer are:

```
class uSocketServer {
  public:
    uSocketServer( char *name, int domain = AF_UNIX, int type = SOCK_STREAM, int protocol = 0,
                int backlog = 5 );
    uSocketServer( unsigned short port, char *name, int type = SOCK_STREAM, int protocol = 0,
                int backlog = 5 );
    int getsockname( struct sockaddr *name, int *len );
    int getpeername( struct sockaddr *name, int *len );
};
```

The parameters for the first constructor of uSocketServer are as follows. The name parameter is the name of the new UNIX server socket. The domain, type and protocol parameters are explained in the UNIX manual entry for socket; they specify the type of the server socket and control the kind of server communication. The backlog parameters is explained in the UNIX manual entry for listen; it specifies a limit on the number of incoming connections from clients. The parameters for the second constructor of uSocketServer are as follows. The port parameter is the port number of an INET port on a host machine specified by the name parameter. The domain parameter is always AF_INET and the remaining parameters are the same as above. It is *not* meaningful to read or to assign to a uSocketServer object, or copy a uSocketServer object (e.g. pass it as a value parameter).

The parameters and return value for members getsockname and getpeername are explained in their corresponding UNIX manual entries. (The first parameter to these UNIX routines is unnecessary, as it is provided implicitly by the uSocketServer object.)

4.4.3 Server Acceptor

After a server socket is created, it is possible to accept connections from clients by declaring an acceptor object, for example:

```
uSocketAccept acceptor( server );
```

which creates an acceptor object, acceptor, that blocks until a client connects to the UNIX socket, abc, represented by server object server. The operations available on an acceptor object are:

```
class uSocketAccept {
  public:
    uSocketAccept( uSocketServer &s, void *adr = 0, int *len = 0 );
    int getsockname( struct sockaddr *name, int *len );
    int getpeername( struct sockaddr *name, int *len );
    int read( void *buf, int len );
    int write( void *buf, int len );
    int send( void *buf, int len, int flags );
    int sendto( void *buf, int len, int flags, void *to, int tolen );
    int sendmsg( void *msg, int flags );
    int recv( void *buf, int len, int flags );
    int recvfrom( void *buf, int len, int flags, void *from, int *fromlen );
    int recvmsg( void *msg, int flags );
};
```

The parameters for the constructor uSocketAccept are as follows. The s parameter is a uSocketServer object through which a connection to a client is to be made. The adr and len parameters are explained in the UNIX manual entry for accept; they are used to determine information about the client that the acceptor is connected to. It is *not* meaningful to read or to assign to a uSocketAccept object, or copy a uSocketAccept object (e.g. pass it as a value parameter).

The parameters and return value for the communication members are explained in their corresponding UNIX manual entries. (The first parameter to these UNIX routines is unnecessary, as it is provided implicitly by the uSocketAccept object.) Appendix B.5.2 shows a server and accepting connections from clients.

> □ μC++ does *not* support out-of-band data on sockets. Out-of-band data requires the ability to install a signal handler on the server's I/O cluster (see Section 4.1). Currently, there is no facility to do this. □

Chapter 5

μC++ Concurrency Techniques

The following shows how different features of μC++ can be used in writing concurrent programs. (This section will be expanded as time permits.)

5.1 Asynchronous Communication

While μC++ provides only synchronous communication facilities, it is possible to build asynchronous facilities out of the synchronous ones. The following discussion shows some techniques for doing this.

5.1.1 Asynchronous Call

When a call occurs to a mutex member, the caller blocks until the mutex member completes. If the purpose of the call is to only transfer data to be manipulated by the receiver, the caller should not have to wait until the receiver performs the manipulation. This can be handled by moving code from the member routine to the statement executed after the member is accepted, as in:

```
uTask t1 {
  public:
    void xxx(...) { code1 }
    void yyy(...) { code2 }
    void main() {
        ...
        uAccept( xxx );
        uOr uAccept( yyy );
    }
}
```

```
uTask t2 {
  public:
    void xxx(...) { copy parameters }
    void yyy(...) { copy parameters }
    void main() {
        ...
        uAccept( xxx ) { code1 }
        uOr uAccept( yyy ) { code2 };
    }
}
```

In task t1, a caller of member routines xxx or yyy is blocked until code *code1* or *code2* is executed, respectively. In task t2, control returns back to the caller after the parameters have been copied into local variables of the task. Then the local variables are manipulated in *code1* or *code2* of the accept statement, respectively, by the task.

5.1.2 Buffers

The previous technique provides buffering of size one between the caller and the receiving task. If a call is made before the receiving task can execute the code in the statements of the accept statement, the call waits. To allow the caller to get further ahead, a buffer must be used to store the arguments of the call until the receiver processes them. Appendix B.2 shows several ways that generic bounded buffers can be built. However, it is worth noting that unless the average time for production and consumption of a message is approximately equal with only a small variance, the buffer will always be either full or empty making communication virtually synchronous.

5.1.3 Futures

Neither of the previous cases can provide any asynchrony if a result is returned as part of the call. To achieve asynchrony when a results is required, the single call must be transformed into two calls: the first call to transmit the arguments and a subsequent call to retrieve the results. The time between the two calls allows the calling task to execute asynchronously with the task performing the operation on the caller's behalf. If the result is not ready when the second call is made, the caller blocks. However, this requires a protocol so that when the caller makes the second call the correct results can be returned.

One form of protocol is the use of a token or ticket. The first part of the protocol transmits the arguments specifying the desired work and a ticket (like a laundry ticket) is returned immediately. The second call passes the ticket to retrieve the result. The ticket is matched with a result, and the result is returned if available or the caller is blocked until the result for that ticket is available. However, protocols are error prone because the caller may not obey the protocol (e.g. never retrieve a result or use the same ticket twice).

A **future** is a mechanism to provide the same asynchrony as above but without an explicit protocol. The protocol becomes implicit between the future and the task generating the result. Further, it removes the difficult problem of when the caller should try to retrieve the results. In detail, a future is an object that is a subtype of the result type expected by the caller. Instead of two calls as before, a single call is made, passing the appropriate arguments, and a future is returned. In general, the future is return immediately and it is empty. In other words, the caller "believes" the call completed and continues execution with an empty result value. The future is filled in after the actual result is calculated. If the caller tries to use the future before its value is filled in, the caller is implicitly blocked.

A simple future can be constructed out of a semaphore and link field, as in:

```
class future {
    friend uTask server;                    // allow server to access internal state

    uSemaphore resultAvailable;
    future *link;
    ResultType result;
  public:
    future() : resultAvailable( 0 ) {}

    ResultType get() {
        resultAvailable.uP();               // wait for result
        return result;
    }
};
```

The semaphore is used to block the caller if the future is empty and the link field is used to chain the future onto a work list of the task generating results. Unfortunately, the syntax for retrieving the value of the future is awkward as it requires a call to the get routine.

□ Separating assignment into two routines, get and put, as in CLU [LAB+81], would have solved this syntactic problem. □

Chapter 6

Miscellaneous

6.1 Symbolic Debugging

The symbolic debugging tools (e.g. dbx, gdb) do not work perfectly with μC++. This is because each coroutine and task has its own stack, and the debugger does not know that there are multiple stacks. When a program terminates with an error, only the stack of the coroutine or task in execution at the time of the error is understood by the debugger. Further, in the multiprocessor case, there are multiple UNIX processes that are not necessarily handled well by all debuggers. Nevertheless, it is possible to use many debuggers on programs compiled with the unikernel. At the very least, it is usually possible to examine some of the variables, externals and ones local to the current coroutine or task, and to discover the statement where the error occurred.

For most debuggers it is necessary to tell them to let the μC++ runtime system handle certain UNIX signals. For the unikernel, signal SIGALRM is handled by μC++ to perform pre-emptive scheduling. In gdb, the following debugger command allows the application program to handle signal SIGALRM:

```
handle SIGALRM nostop noprint pass ignore
```

For the multikernel on Sun multiprocessors, signals SIGSEGV and SIGBUS are handled by μC++. In gdb, the following debugger commands allow the application program to handle signals SIGSEGV and SIGBUS:

```
handle SIGSEGV nostop noprint pass ignore
handle SIGBUS  nostop noprint pass ignore
```

6.2 Monitoring Multiprocessor Execution

When executing a multiprocessor μC++ program, it is possible to monitor its execution at a very high level using the UNIX ps command. The following is an example of the output from ps for a program that has a computational cluster with 4 virtual processors and 3 open files. The user cluster is used for the computational cluster, which sets the number of processors and then opens 3 files. During execution of this program, called a.out, the following output might appear from ps (the right hand column is annotation information for the explanation and not part of the output from ps):

13166	p1	S	0:00 a.out	*virtual processor for system cluster*
13167	p1	R	0:07 a.out	*virtual processor for user cluster*
13168	p1	R	0:07 a.out	"
13169	p1	R	0:07 a.out	"
13170	p1	R	0:08 a.out	"
13171	p1	D	0:02 a.out	*virtual processor for open file 1*
13173	p1	D	0:01 a.out	*virtual processor for open file 2*
13174	p1	D	0:00 a.out	*virtual processor for open file 3*

The lowest numbered process (13166) (unless the process numbers wrapped around at zero) is always the virtual processor for the system cluster. This virtual processor has a runtime of 0:00, which indicates that there has been less than 1 second of activity on the system cluster. If a program is not performing input or output from/to uCin or uCout or uCerr, the system cluster has little activity (i.e. just a small amount of polling, hence the process status will be S which means sleeping for less than about 20 seconds). The next 4 UNIX processes (13167-70) are the virtual processors for the computational cluster (user cluster). As long as there is work for the virtual processors and they are not interrupted frequently by the operating system, they will execute at approximately the same rate. Here there are 3 virtual processors that have executed for at least 7 seconds and one that is slightly ahead at 8 seconds. These processes have status R, which means a runnable process. The last 3 UNIX processes (13171,13173-4) are for the 3 open files–one cluster with a virtual processor for each open file. The execution times for the 3 files varies with the amount of I/O activity to the file. In this case, file 1 has the most activity (0:02 seconds), then file 2 (0:01 seconds) and finally file 3 (less than 0:01 second). These processes have status D, which means they are in a disk wait. Using this mechanism it is possible to monitor the execution of a program to ensure that it is making progress. If all the computational virtual processors have status S or I (sleeping longer than about 20 seconds), the system may be deadlocked (however, they might also be blocked waiting for a terminal I/O operation to complete).

6.3 Installation Requirements

μC++ comes configured to run on any of the following platforms: sequent-i386 (both single and multiple processor), sun-sparc (both single and multiple processor), sun-m68k, dec-vax, dec-mips, mips-mips, sgi-mips (both single and multiple processor), ibm-rs6000, hp-hppa. If you wish to port μC++ to another platform, it has been know to work on UNIX BSD 4.3, UNIX System V that has BSD system calls setitimer and a sigcontext passed to signal handlers which contains the location of the interrupted program, versions of QNIX and LINIX for PCs, SUN OS, ULTRIX, DYNIX and IRIX.

μC++ requires at least GNU C++ 2.3.3 [Tie90]. This compiler can be obtained free of charge. Currently, μC++ will **NOT** compile using other compilers but we hope to be able to use other C++ compilers shortly. To run multiprocessor on a Sequent GNU C++ must use the Sequent assembler and loader, which means using collect. This is because the GNU assembler does not handle the assembler directives generated from GNU C++ when the -fshared-data flag is used.

6.4 Installation

μC++ requires at least version 3.8 of dmake, which is available by anonymous ftp from the following location (remember to set your ftp mode to "binary"):

```
plg.uwaterloo.ca:pub/dmake/dmake38.tar.Z
```

Execute the following command to unpack the source:

```
% zcat dmake38.tar.Z | tar -xf -
```

The _install file contains instructions on how to build dmake.

The current version of μC++ can be obtained by anonymous ftp from the following location (remember to set your ftp mode to "binary"):

```
plg.uwaterloo.ca:pub/uSystem/u++-3.7.tar.Z
```

Execute the following command to unpack the source:

```
% zcat u++-3.7.tar.Z | tar -xf -
```

The README file contains instructions on how to build μC++.

6.5 Reporting Problems

If you have problems or questions or suggestions, send e-mail to usystem@maytag.uwaterloo.ca or mail to:

μSystem Project
c/o Peter A. Buhr
Dept. of Computer Science
University of Waterloo
Waterloo, Ontario
N2L 3G1
CANADA

6.6 Contributors

While many people have made numerous suggestions, the following people were instrumental in turning this project from an idea into reality. The original design work, Version 1.0, was done by Peter Buhr, Glen Ditchfield and Bob Zarnke [BDZ89], with additional help from Jan Pachl on the train to Wengen. Brian Younger built Version 1.0 by modifying the AT&T 1.2.1 C++ compiler [You91]. Version 2.0 was designed by Peter Buhr, Glen Ditchfield, Rick Stroobosscher and Bob Zarnke [BDS+92]. Version 3.0 was designed by Peter Buhr, Rick Stroobosscher and Bob Zarnke. Rick Stroobosscher built both Version 2.0 and 3.0 translator and kernel. Peter Buhr wrote the documentation and built the non-blocking I/O library as well as doing other sundry coding. As always, Bob Zarnke made sure it was all done correctly.

The indirect contributers are Richard Stallman for providing emacs so that we could accomplish useful work in UNIX and Michael D. Tiemann for building GNU C++.

Appendix A

μC++ Grammar

The grammar for μC++ is an extension of the grammar for C++ given in [ES90, Chapter 17]. The ellipsis in the following rules represent the productions elided from the C++ grammar.

type-specifier:
    ...
    uMutex
    uNoMutex
class-key:
    ...
    uCoroutine
    uTask
statement:
    ...
    uWait *expression* ;
    uSignal *expression* ;
    uSuspend ;
    uResume ;
    accept-statement ;
accept-statement:
    when-clause$_{opt}$ **uAccept** (*dname*) *statement*
    when-clause$_{opt}$ **uAccept** (*dname*) *statement* **uOr** *accept-statement*
    when-clause$_{opt}$ **uAccept** (*dname*) *statement* **uElse** *statement*
when-clause:
    uWhen (*expression*)

Appendix B

Example Programs

B.1 Coroutine Binary Insertion Sort

The coroutine BinarySort inputs positive integer values to be sorted and sorts them using the binary insertion sort method. For each integer in the set to be sorted, BinarySort is restarted with the integer as the argument. The end of the set of integers to be sorted is signalled with the value -1. When the coroutine receives a value of -1, it stops sorting and prepares to return the sorted integers one at a time. To retrieve the sorted integers, BinarySort is restarted once for each integer in the sorted set. Each restart returns as its result the next integer of the sorted set. The last value returned by BinarySort is -1, which denotes the end of the sorted set, and then BinarySort terminates.

If the set of integers contains more than one value, BinarySort sorts them by creating two more instances of BinarySort, and having each of them sort some of the integers. Each of the two new coroutines may eventually have to create two more coroutines in turn. The result is a binary tree of coroutines. No error checks are made to ensure that member routine output is not called during the sorting phase and that member routine input is not called during the output phase.

```
//                          -*- Mode: C++ -*-
//
// uC++ Version 3.7, Copyright (C) Peter A. Buhr 1990
//
// BinaryInsertSort.cc -- Binary Insertion Sort, semi-coroutines
//
// Author           : Peter A. Buhr
// Created On       : Thu Aug  2 11:53:37 1990
// Last Modified By : Peter A. Buhr
// Last Modified On : Wed Mar 31 21:08:24 1993
// Update Count     : 43
//

#include <uC++.h>
#include <uIOStream.h>

uCoroutine BinarySort {
  private:
    int in, out;
    void main();
  public:
    void input( int );
    int output();
}; // BinarySort

void BinarySort::main() {
```

```
    int pivot;

    pivot = in;                                     // first value is the pivot value
    if ( pivot == -1 ) {                            // no data values
        uSuspend;                                   // acknowledge end of input
        out = -1;
        return;                                     // terminate output
    } // if

    BinarySort less, greater;                       // create siblings

    for ( ;; ) {
        uSuspend;                                   // get more input
      if ( in == -1 ) break;
        if ( in <= pivot ) {                        // direct value along appropriate branch
            less.input( in );
        } else {
            greater.input( in );
        } // if
    } // for

    less.input( -1 );                               // terminate input
    greater.input( -1 );                            // terminate input
    uSuspend;                                       // acknowledge end of input

    // return sorted values

    for ( ;; ) {
        out = less.output();                        // retrieve the smaller values
      if ( out == -1 ) break;                       // no more smaller values ?
        uSuspend;                                   // return smaller values
    } // for

    out = pivot;
    uSuspend;                                       // return the pivot

    for ( ;; ) {
        out = greater.output();                     // retrieve the larger values
      if ( out == -1 ) break;                       // no more larger values ?
        uSuspend;                                   // return larger values
    } // for

    out = -1;
    return;                                         // terminate output
} // BinarySort::main

void BinarySort::input( int val ) {
    in = val;
    uResume;
} // BinarySort::input

int BinarySort::output() {
    uResume;
    return out;
} // BinarySort::output

void uMain::main() {
    const int NoOfValues = 20;
```

```
    BinarySort bs;
    int value;
    int i;

    // sort values

    uCout << "unsorted values:" << endl;
    for ( i = 1; i <= NoOfValues; i += 1 ) {
        value = random() % 100;
        uCout << value << " ";
        bs.input( value );
    } // for
    uCout << endl;
    bs.input( -1 );

    // retrieve sorted values

    uCout << "sorted values:" << endl;
    for ( ;; ) {
        value = bs.output();                    // retrieve values
      if ( value == -1 ) break;                 // no more values ?
        uCout << value << " ";                  // print values
    } // for
    uCout << endl;
} // uMain::main

// Local Variables: //
// compile-command: "u++ BinaryInsertionSort.cc" //
// End: //
```

B.2 Bounded Buffer

Two processes communicate through a unidirectional queue of finite length. Semaphores are used to ensure that should the queue fill, the producer waits until some free queue element appears, and if the queue is empty, the consumer waits until an element appears. Also, a lock is used to ensure mutually exclusive access to the front and back of the queue for removals and insertions.

B.2.1 Using Monitor Accept

```
//                              -*- Mode: C++ -*-
//
// uC++ Version 3.7, Copyright (C) Peter A. Buhr 1990
//
// MonAcceptBB.cc -- Generic bounded buffer problem using a monitor and uAccept
//
// Author           : Peter A. Buhr
// Created On       : Thu Aug  2 11:35:05 1990
// Last Modified By : Peter A. Buhr
// Last Modified On : Sat May  1 10:45:17 1993
// Update Count     : 107
//

#include <uC++.h>
#include <uIOStream.h>

template<class ELEMTYPE> uMonitor BoundedBuffer {
```

```
    const int size;                                          // number of buffer elements
    int front, back;                                         // position of front and back of queue
    int count;                                               // number of used elements in the queue
    ELEMTYPE *Elements;
  public:
    BoundedBuffer( const int size = 10 ) : size( size ) {
        front = back = count = 0;
        Elements = new ELEMTYPE[size];
    } // BoundedBuffer::BoundedBuffer

    ~BoundedBuffer() {
        delete [] Elements;
    } // BoundedBuffer::~BoundedBuffer

    uNoMutex int query() {
        return count;
    } // BoundedBuffer::query

    void insert( ELEMTYPE elem );
    ELEMTYPE remove();
}; // BoundedBuffer

template<class ELEMTYPE> inline void BoundedBuffer<ELEMTYPE>::insert( ELEMTYPE elem ) {
    if ( count == size ) {                                   // buffer full ?
        uAccept( remove );                                   // only allow removals
    } // if

    Elements[back] = elem;
    back = ( back + 1 ) % size;
    count += 1;
}; // BoundedBuffer::insert

template<class ELEMTYPE> inline ELEMTYPE BoundedBuffer<ELEMTYPE>::remove() {
    ELEMTYPE elem;

    if ( count == 0 ) {                                      // buffer empty ?
        uAccept( insert );                                   // only allow insertions
    } // if

    elem = Elements[front];
    front = ( front + 1 ) % size;
    count -= 1;

    return( elem );
}; // BoundedBuffer::remove

#include "ProdConsDriver.i"

// Local Variables: //
// compile-command: "u++ MonAcceptBB.cc" //
// End: //
```

B.2.2 Using Monitor Condition

```
//                              -*- Mode: C++ -*-
//
// uC++ Version 3.7, Copyright (C) Peter A. Buhr 1990
```

```
//
// MonConditionBB.cc -- Generic bounded buffer problem using a monitor and condition variables
//
// Author           : Peter A. Buhr
// Created On       : Thu Aug  2 11:35:05 1990
// Last Modified By : Peter A. Buhr
// Last Modified On : Fri May  7 21:26:17 1993
// Update Count     : 47
//

#include <uC++.h>
#include <uIOStream.h>

template<class ELEMTYPE> uMonitor BoundedBuffer {
    const int size;                                  // number of buffer elements
    int front, back;                                 // position of front and back of queue
    int count;                                       // number of used elements in the queue
    ELEMTYPE *Elements;
    uCondition BufFull, BufEmpty;
  public:
    BoundedBuffer( const int size = 10 ) : size( size ) {
        front = back = count = 0;
        Elements = new ELEMTYPE[size];
    } // BoundedBuffer::BoundedBuffer

    ~BoundedBuffer() {
        delete [] Elements;
    } // BoundedBuffer::~BoundedBuffer

    uNoMutex int query() {
        return count;
    } // BoundedBuffer::query

    void insert( ELEMTYPE elem ) {
        if ( count == size ) {
            uWait BufFull;
        } // if

        Elements[back] = elem;
        back = ( back + 1 ) % size;
        count += 1;

        uSignal BufEmpty;
    }; // BoundedBuffer::insert

    ELEMTYPE remove() {
        ELEMTYPE elem;

        if ( count == 0 ) {
            uWait BufEmpty;
        } // if

        elem = Elements[front];
        front = ( front + 1 ) % size;
        count -= 1;

        uSignal BufFull;
        return elem;
```

```
    }; // BoundedBuffer::remove
}; // BoundedBuffer

#include "ProdConsDriver.i"

// Local Variables: //
// compile-command: "u++ MonConditionBB.cc" //
// End: //
```

B.2.3 Using Task

```
//                              -*- Mode: C++ -*-
//
// uC++ Version 3.7, Copyright (C) Peter A. Buhr 1990
//
// TaskAcceptBB.cc -- Generic bounded buffer using a task
//
// Author           : Peter A. Buhr
// Created On       : Sun Sep 15 20:24:44 1991
// Last Modified By : Peter A. Buhr
// Last Modified On : Fri May  7 21:27:33 1993
// Update Count     : 55
//

#include <uC++.h>
#include <uIOStream.h>

template<class ELEMTYPE> uTask BoundedBuffer {
    const int size;                                    // number of buffer elements
    int front, back;                                   // position of front and back of queue
    int count;                                         // number of used elements in the queue
    ELEMTYPE *Elements;
  public:
    BoundedBuffer( const int size = 10 ) : size( size ) {
        front = back = count = 0;
        Elements = new ELEMTYPE[size];
    } // BoundedBuffer::BoundedBuffer

    ~BoundedBuffer() {
        delete [] Elements;
    } // BoundedBuffer::~BoundedBuffer

    uNoMutex int query() {
        return count;
    } // BoundedBuffer::query

    void insert( ELEMTYPE elem ) {
        Elements[back] = elem;
    } // BoundedBuffer::insert

    ELEMTYPE remove() {
        return Elements[front];
    } // BoundedBuffer::remove
  protected:
    void BoundedBuffer::main() {
        for ( ;; ) {
            uAccept( ~BoundedBuffer ) {
```

```
            break;
        } uOr uWhen ( count != size ) uAccept( insert ) {
            back = ( back + 1 ) % size;
            count += 1;
        } uOr uWhen ( count != 0 ) uAccept( remove ) {
            front = ( front + 1 ) % size;
            count -= 1;
        } // uAccept
    } // for
  } // BoundedBuffer::main
}; // BoundedBuffer

#include "ProdConsDriver.i"

// Local Variables: //
// compile-command: "u++ TaskAcceptBB.cc" //
// End: //
```

B.2.4 Using P/V

```
//                              -*- Mode: C++ -*-
//
// uC++ Version 3.7, Copyright (C) Peter A. Buhr 1990
//
// SemaphoreBB.cc -- Generic bounded Buffer using P and V
//
// Author           : Peter A. Buhr
// Created On       : Thu Aug 15 16:42:42 1991
// Last Modified By : Peter A. Buhr
// Last Modified On : Fri May  7 21:27:19 1993
// Update Count     : 41
//

#include <uC++.h>
#include <uIOStream.h>

template<class ELEMTYPE> class BoundedBuffer {
    const int size;                                 // number of buffer elements
    int front, back;                                // position of front and back of queue
    uSemaphore full, empty;                         // synchronize for full and empty BoundedBuffer
    uSemaphore ilock, rlock;                        // insertion and removal locks
    ELEMTYPE *Elements;
  public:
    BoundedBuffer( const int size = 10 ) : size( size ), full( 0 ), empty( size ), ilock( 1 ), rlock( 1 ) {
        front = back = 0;
        Elements = new ELEMTYPE[size];
    } // BoundedBuffer::BoundedBuffer

    ~BoundedBuffer() {
        delete  Elements;
    } // BoundedBuffer::~BoundedBuffer

    void insert( ELEMTYPE elem ) {
        empty.uP();                                 // wait if queue is full

        ilock.uP();                                 // serialize insertion
        Elements[back] = elem;
```

```
        back = ( back + 1 ) % size;
        ilock.uV();

        full.uV();                                  // signal a full queue space
    } // BoundedBuffer::insert

    ELEMTYPE remove() {
        ELEMTYPE elem;

        full.uP();                                  // wait if queue is empty

        rlock.uP();                                 // serialize removal
        elem = Elements[front];
        front = ( front + 1 ) % size;
        rlock.uV();

        empty.uV();                                 // signal empty queue space
        return( elem );
    } // BoundedBuffer::remove
}; // BoundedBuffer

#include "ProdConsDriver.i"

// Local Variables: //
// compile-command: "u++ SemaphoreBB.cc" //
// End: //
```

B.3 Disk Scheduler

The following example illustrates a fully implemented disk scheduler. The disk scheduling algorithm is the elevator algorithm, which services all the requests in one direction and then reverses direction. A linked list is used to store incoming requests while the disk is busy servicing a particular request. The nodes of the list are stored on the stack of the calling processes so that suspending a request does not consume resources. The list is maintained in sorted order by track number and there is a pointer which scans backward and forward through the list. New requests can be added both before and after the scan pointer while the disk is busy. If new requests are added before the scan pointer in the direction of travel, they are serviced on that scan.

The disk calls the scheduler to get the next request that it services. This call does two things: it passes to the scheduler the status of the just completed disk request, which is then returned from scheduler to disk user, and it returns the information for the next disk operation. When a user's request is accepted, the parameter values from the request are copied into a list node, which is linked in sorted order into the list of pending requests. The disk removes work from the list of requests and stores the current request it is performing in CurrentRequest. When the disk has completed a request, the request's status is placed in the CurrentRequest node and the user corresponding to this request is reactivated.

```
//                          -*- Mode: C++ -*-
//
// uC++ Version 3.7, Copyright (C) Peter A. Buhr 1991
//
// LOOK.cc -- Look Disk Scheduling Algorithm
//
// The LOOK disk scheduling algorithm causes the disk arm to sweep
// bidirectionally across the disk surface until there are no more
// requests in that particular direction, servicing all requests in
// its path.
//
```

```
// Author           : Peter A. Buhr
// Created On       : Thu Aug 29 21:46:11 1991
// Last Modified By : Peter A. Buhr
// Last Modified On : Sun May 16 22:41:38 1993
// Update Count     : 203
//

#include <uC++.h>
#include <uIOStream.h>
#include <unistd.h>

enum boolean { FALSE, TRUE };

typedef char Buffer[50];                                  // dummy data buffer

const int NoOfCylinders = 100;
enum IOStatus { IO_COMPLETE, IO_ERROR };

class IORequest {
  public:
    int track;
    int sector;
    Buffer *bufadr;
    IORequest() {}
    IORequest( int track, int sector, Buffer *bufadr ) {
        IORequest::track = track;
        IORequest::sector = sector;
        IORequest::bufadr = bufadr;
    } // IORequest::IORequest
}; // IORequest

#include "/u/usystem/software/collection/src/Sequence.h"

class WaitingRequest : public Sequable {                  // element for a waiting request list
  public:
    uCondition block;
    IOStatus status;
    IORequest req;
    WaitingRequest( IORequest req ) {
            WaitingRequest::req = req;
    }
}; // WaitingRequest

class Elevator : public Sequence<WaitingRequest> {
    boolean Direction;
    WaitingRequest *Current;
  public:
    Elevator() {
        Direction = TRUE;
    } // Elevator::Elevator
    void orderedInsert( WaitingRequest *np ) {
        for ( WaitingRequest *lp = head();                // insert in ascending order by track number
            lp != 0 && lp->req.track < np->req.track;
            lp = succ( lp ) );
        if ( isEmpty() ) Current = np;                    // 1st client, so set Current
        insert( np, lp );
    } // Elevator::orderedInsert
    WaitingRequest *Remove() {
```

```
        WaitingRequest *temp = Current;                                 // advance to next waiting client
        Current = Direction ? succ( Current ) : pred( Current );
        remove( temp );                                                 // remove request

        if ( Current == 0 ) {                                           // reverse direction ?
            uCout << uAcquire << "Turning" << endl << uRelease;
            Direction = !Direction;
            Current = Direction ? head() : tail();
        } // if
        return( temp );
    } // Elevator::Remove
}; // Elevator

uTask DiskScheduler;

uTask Disk {
    DiskScheduler &scheduler;
    void main();
  public:
    Disk( DiskScheduler &scheduler ) : scheduler( scheduler ) {
    } // Disk
}; // Disk

uTask DiskScheduler {
    Elevator PendingClients;                                            // ordered list of client requests
    uCondition DiskWaiting;                                             // disk waits here if no work
    WaitingRequest *CurrentRequest;                                     // request being serviced by disk
    Disk disk;                                                          // start the disk
    IORequest req;
    WaitingRequest diskterm;                                            // preallocate disk termination request

    void main();
  public:
    DiskScheduler() : disk( *this ), req( -1, 0, 0 ), diskterm( req ) {
    } // DiskScheduler
    IORequest WorkRequest( IOStatus );
    IOStatus DiskRequest( IORequest & );
}; // DiskScheduler

uTask DiskClient {
    DiskScheduler &scheduler;
    void main();
  public:
    DiskClient( DiskScheduler &scheduler ) : scheduler( scheduler ) {
    } // DiskClient
}; // DiskClient

void Disk::main() {
    IOStatus status;
    IORequest work;

    status = IO_COMPLETE;
    for ( ;; ) {
        work = scheduler.WorkRequest( status );
      if ( work.track == -1 ) break;
        uCout << uAcquire << "Disk main, track:" << work.track << endl << uRelease;
        uDelay( 100 );                                                  // pretend to perform an I/O operation
        status = IO_COMPLETE;
```

```
    } // for
} // Disk::main

void DiskScheduler::main() {
    SeqGen<WaitingRequest> gen;                       // declared here because of gcc compiler bug

    CurrentRequest = NULL;                            // no current request at start
    for ( ;; ) {
        uAccept( ~DiskScheduler ) {                   // request from system
            break;
        } uOr uAccept( WorkRequest ) {                // request from disk
        } uOr uAccept( DiskRequest ) {                // request from clients
        } // uAccept
    } // for

    // two alternatives for terminating scheduling server
#if 0
    for ( ; ! PendingClients.isEmpty(); ) {           // service pending disk requests before terminating
        uAccept( WorkRequest );
    } // for
#else
    WaitingRequest *client;                           // cancel pending disk requests before terminating

    for ( gen.over(PendingClients); gen >> client; ) {
        PendingClients.Remove();                      // remove each client from the list
        client->status = IO_ERROR;                    // set failure status
        uSignal client->block;                        // restart client
    } // for
#endif
    // pending client list is now empty

    // stop disk
    PendingClients.orderedInsert( &diskterm );        // insert disk terminate request on list

    if ( !DiskWaiting.uEmpty() ) {                    // disk free ?
        uSignal DiskWaiting;                          // wake up disk to deal with termination request
    } else {
        uAccept( WorkRequest );                       // wait for current disk operation to complete
    } // if
} // DiskScheduler::main

IOStatus DiskScheduler::DiskRequest( IORequest &req ) {
    WaitingRequest np( req );                         // preallocate waiting list element

    PendingClients.orderedInsert( &np );              // insert in ascending order by track number
    if ( !DiskWaiting.uEmpty() ) {                    // disk free ?
        uSignal DiskWaiting;                          // reactivate disk
    } // if

    uWait np.block;                                   // wait until request is serviced

    return( np.status );                              // return status of disk request
} // DiskScheduler::DiskRequest

IORequest DiskScheduler::WorkRequest( IOStatus status ) {
    if ( CurrentRequest != NULL ) {                   // client waiting for request to complete ?
        CurrentRequest->status = status;              // set request status
        uSignal CurrentRequest->block;                // reactivate waiting client
```

```
    } // if

    if ( PendingClients.isEmpty() ) {                      // any clients waiting ?
        uWait DiskWaiting;                                  // wait for client to arrive
    } // if

    CurrentRequest = PendingClients.Remove();              // remove next client's request
    return( CurrentRequest->req );                          // return work for disk
} // DiskScheduler::WorkRequest

void DiskClient::main() {
    IOStatus status;
    IORequest req( random() % NoOfCylinders, 0, 0 );

    uDelay( random() % 50 );                                // delay a random period before making request
    uCout << uAcquire << "enter DiskClient main seeking:" << req.track << endl << uRelease;
    status = scheduler.DiskRequest( req );
    uCout << uAcquire << "enter DiskClient main seeked to:" << req.track << endl << uRelease;
} // DiskClient::main

void uMain::main() {
    const int NoOfTests = 20;
    DiskScheduler scheduler;                                // start the disk scheduler
    DiskClient *p[NoOfTests];
    int i;

    srandom( getpid() );                                    // initialize random number generator

    for ( i = 0; i < NoOfTests; i += 1 ) {
            p[i] = new DiskClient( scheduler );             // start the clients
            uDelay( random() % 10 );
    } // for

    for ( i = 0; i < NoOfTests; i += 1 ) {
        delete p[i];                                        // wait for clients to complete
    } // for

    uCout << "successful execution" << endl;
} // uMain::main

// Local Variables: //
// compile-command: "u++ LOOK.cc" //
// End: //
```

B.4 UNIX File I/O

The following example program reads in a file and copies it into another file.

```
//                              -*- Mode: C++ -*-
//
// uC++ Version 3.7, Copyright (C) Peter A. Buhr 1992
//
// File.cc -- Print multiple copies of the same file to standard output
//
// Author           : Peter A. Buhr
// Created On       : Tue Jan  7 08:44:56 1992
// Last Modified By : Peter A. Buhr
```

```
// Last Modified On : Fri May  7 21:24:00 1993
// Update Count     : 25
//

#include <uC++.h>
#include <uIOStream.h>
#include <uFile.h>

uTask Copier {
    uFile &input;

    void main() {
        uFileAccess in( input, O_RDONLY );
        int count;
        char buf[1];

        for ( ;; ) {                                    // copy in-file to out-file
            count = in.read( buf, sizeof( buf ) );
          if ( count == 0 ) break;                      // eof ?
            uCout << buf[0];
        } // for
    } // Copier::main
  public:
    Copier( uFile &in ) : input( in ) {
    } // Copier::Copier
}; // Copier

void uMain::main() {
    switch ( argc ) {
      case 2:
        break;
      default:
        uCerr << "Usage:  " << argv[0] << " input-file" << endl;
        uExit( -1 );
    } // switch

    uFile input( argv[1] );                             // connect with UNIX files
    {
        Copier c1( input ), c2( input );
    }
} // uMain::main

// Local Variables: //
// compile-command: "u++ File.cc" //
// End: //
```

B.5 UNIX Socket I/O

The following example illustrates bidirectional communication between a client and server socket. A client starts a task to read from standard input and write the data to a server socket. The server's acceptor for that client, reads the data from the client and writes it directly back to the client. The client also starts a task that reads the data coming back from the server's acceptor and writes it onto standard output. Hence, a file is read from standard input and written onto standard output after having made a loop through a server. The server can accept multiple clients.

B.5.1 Socket Client

```
//                              -*- Mode: C++ -*-
//
// uC++ Version 3.7, Copyright (C) Peter A. Buhr 1992
//
// ClientUNIX.cc -- Client connects with server and then communicates with an acceptor.
//
// Author           : Peter A. Buhr
// Created On       : Tue Jan  7 08:42:32 1992
// Last Modified By : Peter A. Buhr
// Last Modified On : Fri May  7 21:23:49 1993
// Update Count     : 48
//

#include <uC++.h>
#include <uIOStream.h>
#include <uSocket.h>

#define EOD '\377'

uTask reader {
    uSocketClient &client;
    void main() {
        char c;
        for ( ;; ) {
            client.read( &c, sizeof( char ) );
          if ( c == EOD ) break;
            uCout << c;
        } // for
    } // reader::main
  public:
    reader( uSocketClient &c ) : client ( c ) {
    } // reader::reader
}; // reader

uTask writer {
    uSocketClient &client;

    void main() {
        char c;
        for ( ;; ) {
            uCin.get( c );
          if ( uCin.eof() ) break;
            client.write( &c, sizeof( char ) );
        } // for
        c = EOD;
        client.write( &c, sizeof( char ) );
    } // writer::main
  public:
    writer( uSocketClient &c ) : client( c ) {
    } // writer::writer
}; // writer

void uMain::main() {
    switch ( argc ) {
      case 2:
```

```
        break;
      default:
        uCerr << "Usage:  " << argv[0] << " socket-name" << endl;
        uExit( -1 );
    } // switch

    uSocketClient client( argv[1] );                    // connection to server
    {
        writer wr( client );                            // emit worker to read from input and write to server
        reader rd( client );                            // emit worker to read from server and write to output
    }
} // uMain::main

// Local Variables: //
// compile-command: "u++ -o Client ClientUNIX.cc" //
// End: //
```

B.5.2 Socket Server

```
//                      -*- Mode: C++ -*-
//
// uC++ Version 3.7, Copyright (C) Peter A. Buhr 1992
//
// SocketServer.cc -- Server accepts multiple connections from clients. Each client then
//                    communicates with an acceptor.
//
// Author           : Peter A. Buhr
// Created On       : Tue Jan  7 08:40:22 1992
// Last Modified By : Peter A. Buhr
// Last Modified On : Fri May  7 21:25:07 1993
// Update Count     : 53
//

#include <uC++.h>
#include <uSocket.h>
#include <uIOStream.h>

#define EOD '\377'

uTask server;                                           // forward declaration

uTask acceptor {
    uSocketServer &sockserver;
    server &ser;

    void main();
  public:
    acceptor( uSocketServer &socks, server &s ) : sockserver( socks ), ser( s ) {
    } // acceptor::acceptor
}; // acceptor

uTask server {
    uSocketServer &sockserver;
    acceptor *terminate;
  public:
    server( uSocketServer &socks ) : sockserver( socks ) {
    } // server::server
```

```
    void connection( void ) {
    } // server::connection

    void complete( acceptor *terminate ) {
        server::terminate = terminate;
    } // server::complete
  private:
    void main() {
        new acceptor( sockserver, *this );                  // create initial acceptor
        for ( ;; ) {
            uAccept( connection ) {
                new acceptor( sockserver, *this );          // create new acceptor after a connection
            } uOr uAccept( complete ) {                     // acceptor has completed with client
                delete terminate;                           // delete must appear here or deadlock
            }; // uAccept
        } // for
    } // server::main
}; // server

void acceptor::main() {
    uSocketAccept acceptor( sockserver );                   // accept a connection from a client

    ser.connection();                                       // tell server about client connection
    for ( ;; ) {
        char c;
        acceptor.read( &c, sizeof(c) );                     // read byte from client
        acceptor.write( &c, sizeof(c) );                    // write byte back to client
      if ( c == EOD ) break;
    } // for
    ser.complete( this );                                   // terminate
} // acceptor::main

void uMain::main() {
    switch ( argc ) {
      case 2:
        break;
      default:
        uCerr << "Usage:  " << argv[0] << " socket-name" << endl;
        uExit( -1 );
    } // switch

    uSocketServer sockserver( argv[1] );                    // create and bind a server socket
    {
        server s( sockserver );                             // does not terminate
    }
} // uMain

// Local Variables: //
// compile-command: "u++ -o Server ServerUNIX.cc" //
// End: //
```

Bibliography

[AOC$^+$88] Gregory R. Andrews, Ronald A. Olsson, Michael Coffin, Irving Elshoff, Kelvin Nilsen, Titus Purdin, and Gregg Townsend. An Overview of the SR Language and Implementation. *ACM Transactions on Programming Languages and Systems*, 10(1):51–86, January 1988.

[BCF] Peter A. Buhr, Michael H. Coffin, and Michel Fortier. Monitors. submitted to ACM Computing Surveys.

[BDS$^+$92] P. A. Buhr, Glen Ditchfield, R. A. Stroobosscher, B. M. Younger, and C. R. Zarnke. μC++: Concurrency in the Object-Oriented Language C++. *Software—Practice and Experience*, 22(2):137–172, February 1992.

[BDZ89] P. A. Buhr, Glen Ditchfield, and C. R. Zarnke. Adding Concurrency to a Statically Type-Safe Object-Oriented Programming Language. *SIGPLAN Notices*, 24(4):18–21, April 1989. Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming, Sept. 26–27, 1988, San Diego, California, U.S.A.

[BLL88] B. N. Bershad, E. D. Lazowska, and H. M. Levy. PRESTO: A System for Object-oriented Parallel Programming. *Software—Practice and Experience*, 18(8):713–732, August 1988.

[BMZ92] Peter A. Buhr, Hamish I. Macdonald, and C. Robert Zarnke. Synchronous and Asynchronous Handling of Abnormal Events in the μSystem. *Software—Practice and Experience*, 22(9):735–776, September 1992.

[Bri75] Per Brinch Hansen. The Programming Language Concurrent Pascal. *IEEE Transactions on Software Engineering*, 2:199–206, June 1975.

[Car90] T. A. Cargill. Does C++ Really Need Multiple Inheritance? In *USENIX C++ Conference Proceedings*, pages 315–323, San Francisco, California, U.S.A., April 1990. USENIX Association.

[CG89] Nicholas Carriero and David Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444–458, April 1989.

[CKL$^+$88] Boleslaw Ciesielski, Antoni Kreczmar, Marek Lao, Andrzej Litwiniuk, Teresa Przytycka, Andrzej Salwicki, Jolanta Warpechowska, Marek Warpechowski, Andrzej Szalas, and Danuta Szczepanska-Wasersztrum. Report on the Programming Language LOGLAN'88. Technical report, Institute of Informatics, University of Warsaw, Pkin 8th Floor, 00-901 Warsaw, Poland, December 1988.

[DG87] Thomas W. Doeppner and Alan J. Gebele. C++ on a Parallel Machine. In *Proceedings and Additional Papers C++ Workshop*, pages 94–107, Santa Fe, New Mexico, U.S.A, November 1987. USENIX Association.

[Dij68] E. W. Dijkstra. The Structure of the "THE"–Multiprogramming System. *Communications of the ACM*, 11(5):341–346, May 1968.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison Wesley, first edition, 1990.

[Gen81] W. Morven Gentleman. Message Passing between Sequential Processes: the Reply Primitive and the Administrator Concept. *Software—Practice and Experience*, 11(5):435–466, May 1981.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[GR88] N. H. Gehani and W. D. Roome. Concurrent C++: Concurrent Programming with Class(es). *Software—Practice and Experience*, 18(12):1157–1177, December 1988.

[Hal85] Robert H. Halstead, Jr. Multilisp: A Language for Concurrent Symbolic Programming. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[HD90] Robert Hieb and R. Kent Dybvig. Continuations and Concurrency. *SIGPLAN Notices*, 25(3):128–136, March 1990. Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practise of Parallel Programming, March. 14–16, 1990, Seattle, Washington, U.S.A.

[Hoa74] C. A. R. Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[Hol92] R. C. Holt. *Turing Reference Manual.* Holt Software Associates Inc., third edition, 1992.

[KMMPN87] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. The BETA Programming Language. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, Computer Systems Series, pages 7–48. MIT Press, 1987.

[LAB+81] Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU Reference Manual*, volume 114 of *Lecture Notes in Computer Science.* Springer-Verlag, 1981.

[Lab90] Pierre Labrèche. Interactors: A Real-Time Executive with Multiparty Interactions in C++. *SIGPLAN Notices*, 25(4):20–32, April 1990.

[Mar80] Christopher D. Marlin. *Coroutines: A Programming Methodology, a Language Design and an Implementation*, volume 95 of *Lecture Notes in Computer Science, Ed. by G. Goos and J. Hartmanis.* Springer-Verlag, 1980.

[Mey88] Bertrand Meyer. *Object-oriented Software Construction.* Prentice Hall International Series in Computer Science. Prentice-Hall, 1988.

[MMS79] James G. Mitchell, William Maybury, and Richard Sweet. Mesa Language Manual. Technical Report CSL–79–3, Xerox Palo Alto Research Center, April 1979.

[RC86] Jonathan Rees and William Clinger. Revised^3 Report on the Algorithmic Language Scheme. *SIGPLAN Notices*, 21(12):37–79, December 1986.

[RH87] A. Rizk and F. Halsall. Design and Implementation of a C-based Language for Distributed Real-time Systems. *SIGPLAN Notices*, 22(6):83–100, June 1987.

[SBG+90] Robert E. Strom, David F. Bacon, Arthur P. Goldberg, Andy Lowry, Daniel M. Yellin, and Shaula Alexander Yemini. Hermes: A Language for Distributed Computing. Technical report, IBM T. J. Watson Research Center, Yorktown Heights, New York, U. S. A., 10598, October 1990.

[Sho87] Jonathan E. Shopiro. Extending the C++ Task System for Real-Time Control. In *Proceedings and Additional Papers C++ Workshop*, pages 77–94, Santa Fe, New Mexico, U.S.A, November 1987. USENIX Association.

[Sta87] Standardiseringskommissionen i Sverige. *Databehandling – Programspråk – SIMULA*, 1987. Svensk Standard SS 63 61 14.

[Str91] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, second edition, 1991.

[Tie88] Michael D. Tiemann. Solving the RPC problem in GNU C++. In *Proceedings of the USENIX C++ Conference*, pages 343–361, Denver, Colorado, U.S.A, October 1988. USENIX Association.

[Tie90] Michael D. Tiemann. User's Guide to GNU C++. Free Software Foundation, 1000 Mass Ave., Cambridge, MA, U. S. A., 02138, March 1990.

[Uni83] United States Department of Defense. *The Programming Language Ada: Reference Manual*, ANSI/MIL-STD-1815A-1983 edition, February 1983. Published by Springer-Verlag.

[Yea91] Dorian P. Yeager. Teaching Concurrency in the Programming Languages Course. *SIGCSE BULLETIN*, 23(1):155–161, March 1991. The Papers of the Twenty-Second SIGCSE Technical Symposium on Computer Science Education, March. 7–8, 1991, San Antonio, Texas, U. S. A.

[You91] Brian M. Younger. Adding Concurrency to C++. Master's thesis, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, 1991.

Index