μC++ Annotated Reference Manual

Version 4.7

Peter A. Buhr and Richard A. Stroobosscher ©[1] 1992
Peter A. Buhr ©[1] 1995, 1996, 1998

May 24, 1999

[1]Permission is granted to make copies for personal or educational use

Contents

Preface **1**

1 μC++ Extensions **3**
1.1 Design Requirements . . . 3
1.2 Elementary Execution Properties . . . 4
1.3 High-level Execution Constructs . . . 5

2 μC++ Translator **7**
2.1 Extending C++ . . . 7
2.2 Compile Time Structure of a μC++ Program . . . 8
2.3 μC++ Runtime Structure . . . 8
2.3.1 Cluster . . . 8
2.3.2 Virtual Processor . . . 9
2.4 μC++ Kernel . . . 10
2.5 Using the μC++ Translator . . . 10
2.5.1 Compiling a μC++ Program . . . 10
2.5.2 Preprocessor Variables . . . 11
2.6 Coroutine . . . 12
2.6.1 Coroutine Creation and Destruction . . . 13
2.6.2 Inherited Members . . . 14
2.6.3 Coroutine Control and Communication . . . 15
2.7 Mutex Types . . . 16
2.8 Scheduling . . . 19
2.8.1 Implicit Scheduling . . . 20
2.8.2 External Scheduling . . . 20
2.8.2.1 Accept Statement . . . 20
2.8.2.2 Breaking a Rendezvous . . . 22
2.8.2.3 Accepting the Destructor . . . 22
2.8.2.4 Commentary . . . 23
2.8.3 Internal Scheduling . . . 23
2.8.3.1 Condition Variables and Wait/Signal Statements . . . 24
2.8.3.2 Commentary . . . 25
2.9 Monitor . . . 26
2.9.1 Monitor Creation and Destruction . . . 26
2.9.2 Monitor Control and Communication . . . 26
2.10 Coroutine-Monitor . . . 26
2.10.1 Coroutine-Monitor Creation and Destruction . . . 27
2.10.2 Coroutine-Monitor Control and Communication . . . 27
2.11 Task . . . 27
2.11.1 Task Creation and Destruction . . . 28
2.11.2 Inherited Members . . . 29
2.11.3 Task Control and Communication . . . 30

2.12 Commentary . . . 31
2.13 Inheritance . . . 32
2.14 Explicit Mutual Exclusion and Synchronization . . . 33
2.14.1 Counting Semaphore . . . 33
2.14.1.1 Commentary . . . 34
2.14.2 Owner Lock . . . 34
2.14.3 Lock . . . 35
2.14.4 Barrier . . . 35
2.15 User Specified Context . . . 36
2.15.1 Predefined Floating Point Context . . . 37
2.16 Implementation Problems . . . 39

3 Input/Output **41**
3.1 Interaction with the File System . . . 41
3.1.1 Unikernel File Operations . . . 41
3.1.2 Multikernel File Operations . . . 41
3.2 μC++ Stream Library . . . 44
3.2.1 uIStream and uOStream I/O . . . 44
3.2.1.1 Commentary . . . 45
3.2.2 uIFStream and uOFStream I/O . . . 46
3.2.3 uIOManip Manipulators . . . 47
3.3 UNIX File I/O . . . 47
3.3.1 File Access . . . 47
3.4 BSD Sockets . . . 48
3.4.1 Client . . . 49
3.4.2 Server . . . 51
3.4.3 Server Acceptor . . . 53
3.4.4 Ancillary Routines . . . 54

4 Abnormal Events and Errors **55**
4.1 AEHM . . . 55
4.2 μC++ AEHM . . . 56
4.3 Abnormal Events . . . 56
4.3.1 Abnormal Event Creation and Destruction . . . 56
4.3.2 Inherited Members . . . 57
4.4 Event Propagation . . . 57
4.4.1 Asynchronous Propagation . . . 58
4.5 Handler . . . 59
4.5.1 Terminating Handler . . . 59
4.5.2 Resuming Handler . . . 60
4.5.2.1 Recursive Resuming . . . 62
4.5.2.2 Preventing Recursive Resuming . . . 62
4.5.2.3 Commentary . . . 64
4.6 Inheritance . . . 66
4.7 Programming with Abnormal Events . . . 66
4.7.1 Throw Event . . . 66
4.7.2 Resume Event . . . 67
4.7.3 Dual Event . . . 67
4.8 Predefined Abnormal Events . . . 67
4.8.1 Implicitly Enabled Events . . . 67
4.8.2 Breaking a Rendezvous . . . 67
4.8.3 Messages . . . 69
4.9 Static (Compile-time) Errors . . . 73
4.9.1 Messages . . . 73

4.10 Dynamic (Run-time) Errors . . . 75
4.10.1 Assertions . . . 75
4.10.2 Termination . . . 75
4.10.3 Messages . . . 76
4.10.3.1 Coroutine . . . 76
4.10.3.2 Task . . . 77
4.10.3.3 Condition Variable . . . 78
4.10.3.4 Locks . . . 79
4.10.3.5 Calendar . . . 79
4.10.3.6 Cluster . . . 80
4.10.3.7 Heap . . . 80
4.10.3.8 I/O . . . 81
4.10.3.9 Processor . . . 81
4.10.3.10 UNIX . . . 82

5 μC++ Kernel 83
5.1 Pre-emptive Scheduling and Critical Sections . . . 83
5.2 Memory Management . . . 84
5.3 Cluster . . . 84
5.4 Processors . . . 85
5.4.1 Implicit Task Scheduling . . . 87
5.4.2 Idle Virtual Processors . . . 87
5.4.3 Blocking Virtual Processors . . . 87
5.5 Default Values . . . 88

6 Real-Time 89
6.1 Time-Defined Delays . . . 89
6.2 Duration and Time . . . 89
6.3 Timeout Operations . . . 91
6.3.1 Accept . . . 92
6.3.2 I/O . . . 92
6.4 Clock . . . 93
6.5 Periodic Task . . . 93
6.6 Sporadic Task . . . 95
6.7 Aperiodic Task . . . 95
6.8 Priority Inheritance Protocol . . . 96
6.9 Real-Time Scheduling . . . 97
6.10 User Supplied Scheduler . . . 98
6.11 Real-Time Cluster . . . 99
6.11.1 Deadline Monotonic Scheduler . . . 99

7 Miscellaneous 103
7.1 Symbolic Debugging . . . 103
7.2 Monitoring Multiprocessor Execution . . . 103
7.3 Installation Requirements . . . 104
7.4 Installation . . . 104
7.5 Reporting Problems . . . 104
7.6 Contributors . . . 105

A μC++ Grammar 107

B Data Structure Library (DSL) **109**
B.1 Stack . . . 109
B.1.1 Generator . . . 110
B.2 Queue . . . 110
B.2.1 Generator . . . 111
B.3 Sequence . . . 112
B.3.1 Generator . . . 113

C Example Programs **115**
C.1 Coroutine Binary Insertion Sort . . . 115
C.2 Readers And Writer . . . 117
C.3 Bounded Buffer . . . 118
C.3.1 Using Monitor Accept . . . 119
C.3.2 Using Monitor Condition . . . 120
C.3.3 Using Task . . . 121
C.3.4 Using P/V . . . 122
C.4 Disk Scheduler . . . 123
C.5 UNIX File I/O . . . 126
C.6 UNIX Socket I/O . . . 127
C.6.1 Client - UNIX/Datagram . . . 128
C.6.2 Server - UNIX/Datagram . . . 129
C.6.3 Client - INET/Stream . . . 130
C.6.4 Socket - INET/Stream . . . 131

Bibliography **135**

Preface

The goal of this work is to introduce concurrency into the object-oriented language C++ [Str97]. 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.

This manual does not discuss how to use the new constructs to build complex concurrent systems. An indepth discussion of these issues, with respect to μC++, is available in "Understanding Control Flow with Concurrent Programming using μC++". This 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 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.

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

Chapter 1

μC++ Extensions

μC++ [BD92] extends the C++ programming language [Str97] in somewhat the same way that C++ extends the C programming language. The extensions introduce new objects that augment the existing set of control flow facilities and provide for light-weight concurrency on uniprocessor and parallel execution on multiprocessor computers running the UNIX operating system. The following discussion is the rationale for the particular extensions that were chosen.

1.1 Design Requirements

The following requirements directed this work:

- Any linguistic feature that affects code generation *must* become part of the language. In other words, if the compiler can generate code that invalidates the correctness of a library package implementing a particular feature, either the library feature cannot be implemented safely or additional capabilities must be added to the programming language to support the feature. Concurrency is a language feature affected by code generation, and hence, must be added to the programming language [Buh95]. In the case of C++, the concurrency extensions are best added through new kinds of objects.

- All communication among the new kinds of objects must be statically type checkable because 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 requirement allows a programmer to choose the kind of object best suited to the particular problem without having to cope with communication restrictions.

  In contrast, some approaches have restrictions on interactions among concurrent 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 restriction 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 among 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. It is confusing to have multiple forms of communication in a language, such as message passing, message queues, or communication ports, as well as normal routine call.

- 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. Requiring users to build mutual exclusion out of locks often leads to incorrect programs. Also, reducing the scope in which synchronization can be used, by encapsulating it as part of language constructs, further reduces errors in concurrent programs.

- Both synchronous and asynchronous communication are needed in a concurrent system. However, 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 using synchronous mechanisms. Building synchronous communication out of asynchronous mechanisms requires a protocol for the caller to subsequently detect completion, which is error prone because the caller may not obey the protocol (e.g., never retrieve a result). Furthermore, 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 implementation requirement creates a storage management problem because different requests require different amounts of storage in the queue. Therefore, asynchronous communication is too complicated and expensive 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 requirement is essential for high priority requests, such as a time out or a termination request. (This priority is to be differentiated from execution priority.) In the latter case, selection control is very precise as the next request must only come from the specified requester. In general, the former case is usually sufficient and simpler to express.

- There must be flexibility in the order that requests are completed. That is, 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].

All of these requirements are satisfied in μC++ except the first, which requires compiler support. Even through μC++ lacks compiler support, its design assumes compiler support so the extensions are easily added to any C++ compiler.

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. Furthermore, 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. When control 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, mutual exclusion must also be provided as an elementary execution property, usually through special atomic hardware instructions.

1.3 High-level Execution Constructs

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.

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 caller are used.

Table 1.1 shows the different abstractions possible when an object possesses different execution properties:

object properties		object's member routine properties	
thread	execution-state	no mutual exclusion	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

Case 1 is an object, such as a **free routine** (a routine not a member of an object) or an object with member routines neither of which has the necessary execution properties, called a **class-object**. In this case, the caller's thread and execution-state are used to perform 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; the name **coroutine-monitor** has been adopted for this case. 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. Furthermore, there is no performance advantage over case 8. For these reasons, this case is rejected. 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 it can solve, and therefore, each has a place in a 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[1] 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 situation is to be distinguished from implicit schemes, such as those that attempt to *discover* concurrency in an otherwise sequential program, e.g., by parallelizing loops and access to data structures. While both schemes are complementary, and hence, can appear together in a single programming language, 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 the addition of the implicit approach.

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, a type defined by the **class** construct and the new constructs are called **class type**, **monitor type**, **coroutine type**, **coroutine-monitor type** and **task type**, 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, e.g., **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 all the μC++ extensions.

object properties		object's member routine properties	
thread	execution-state	no mutual exclusion	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

[1]The term "translator" is used rather than preprocessor because μC++ programs are partially parsed and symbol tables are constructed. A preprocessor, such as cpp, normally only manipulates strings.

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. Furthermore, μC++ uses a **shared-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 approach has its advantages as well as its disadvantages. Communicating objects do not have to 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.

□ Approaches taken by distributed shared-memory systems may provide the necessary implementation mechanisms to make the distributed memory case similar to the shared-memory case. □

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
    int &uRetCode;         // return value to the shell
    void main();           // user provides body for this routine
  public:
    uMain( int argc, char *argv[] ) : argc(argc), argv(argv) {}
};
```

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
    ...
    switch( argc ) {           // use argc from uMain
      case 2:
        no = atoi(argv[1]);    // use argv from uMain
    ...
    uRetcode = 0;              // use uRetcode from uMain
}
```

μC++ supplies 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 two arguments that are passed to the free routine main: argc, and argv. To return a value back to the shell, set the variable uRetCode and return from uMain::main; uRetCode is initialized to zero.

2.3 μC++ Runtime Structure

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 concurrent execution.

2.3.1 Cluster

A cluster is a collection of tasks and virtual processors (discussed next) that execute the 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. Parallelism 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.

Normally, a cluster uses a single-queue multi-server queueing model for scheduling its tasks on its processors (see Chapter 6 for other kinds of schedulers). This simple scheduling 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, both tasks and virtual processors, can be explicitly migrated from one cluster to another. No automatic load balancing among clusters is performed by μC++.

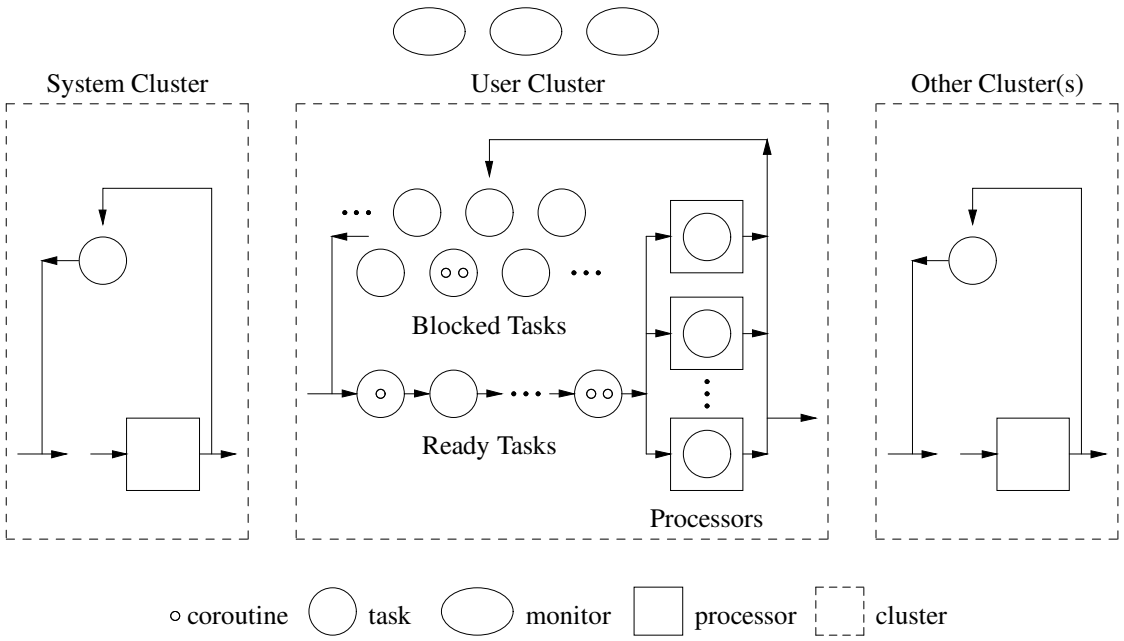


Figure 2.1: Runtime Structure of a μC++ Program

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 partitioning is essential, e.g., 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 by kernel thread (normally created through a UNIX process) that is subsequently scheduled for execution on a hardware processor by the underlying operating system. On a multiprocessor, kernel threads are usually distributed across the hardware processors and so some virtual processors are able to 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 interme-

diate resource, the 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 (in effect, there is only one virtual processor); and the multikernel, which is designed to use several processors. Thus, the unikernel is sensibly used on systems with a single hardware processor or non-shared 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 multiprocessor techniques unnecessarily.

	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

Each of the μC++ kernels has a debugging version, 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 version checks for stack overflow whenever context switches occur among coroutines and tasks, which catches many 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 version. After a program is debugged, the non-debugging version can be used to increase performance.

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. **It must appear before all other include files in each translation unit.**

2.5.1 Compiling a μC++ Program

The command u++ is used to compile μC++ program(s). This command works just like the GNU g++ [Tie90] command for compiling C++ programs, e.g.:

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

The following additional options are available for 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, which 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-debugging version of the unikernel or multikernel, which can substantially increase performance. **However, no runtime checks are performed so errors usually result in abnormal program termination.**

-yield When a 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 non-determinism in execution is in addition to random context switching due to pre-emptive scheduling, see Section 5.4.1). The extra yields of execution can help during the debugging phase of a μC++ program, but substantially slows the execution of the program.

-noyield Additional context switches are not inserted in member routines. **This is the default.**

-verify When a program is translated, a check to verify that the stack has not overflowed occurs at the beginning of each member routine. (This checking 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.**

-trace When a 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 "POET with μC++" for information about displaying event traces.) This option slows the execution of a program.

-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 steps are performed and the transformed program is written to standard output, which makes it possible to examine the code generated by the μC++ translator.

-compiler *name* 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++. (See Section 7.3 for supported compilers.)

-cpp *name* 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., -yield followed by -noyield, the last option takes precedence.

2.5.2 Preprocessor Variables

When programs are compiled using u++, the following preprocessor variables are available:

__U_CPLUSPLUS__ is always available during preprocessing and its value is the current major version number of μC++.

__U_CPLUSPLUS_MINOR__ is always available during preprocessing and its value is the current minor version number of μC++.[2]

[2]The C preprocessor allows only integer values in a preprocessor variable so a value like "4.7" is not allowed. Hence, the need to have two variables for the major and minor version number.

__U_DEBUG__ is available during preprocessing if the -debug compilation option is specified.

__U_YIELD__ is available during preprocessing if the -yield compilation option is specified.

__U_VERIFY__ is available during preprocessing if the -verify compilation option is specified.

__U_TRACE__ is available during preprocessing if the -trace compilation option is specified.

__U_MULTI__ is available during preprocessing if the -multi compilation option is specified.

These preprocessor variables 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
    // body of main routine
}
```

which 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 property means a coroutine is not restarted at the beginning on each activation and 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 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; this distinguished member is called the **coroutine main**. Instead of allowing direct interaction with main, its visibility is normally **private** or **protected**; therefore, a coroutine can only be activated indirectly by one of the coroutine's member routines. The decision to make the coroutine main **private** or **protected** depends solely on whether derived classes can reuse the coroutine main or must supply their own. Hence, a user interacts with a coroutine indirectly through its member routines. This approach 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 global variables, which can be referenced by main.

A coroutine can suspend its execution at any point by activating another coroutine, which is done in two ways. First, a coroutine can implicitly reactivate the coroutine that previously activated it via **uSuspend**. Second, a coroutine

can explicitly invoke a member of another coroutine, which causes activation of that coroutine via **uResume**. These two forms result in two different styles of coroutine control flow. A **full-coroutine** is part of a resume cycle, while a **semi-coroutine** [Mar80, p. 4, 37] is not part of a resume cycle. A full-coroutine can perform semi-coroutine operations because it subsumes the notion of the semi-coroutine; i.e., a full-coroutine can use suspend to activate the member routine that activated it or resume to itself, but it must always form a resume cycle with other coroutines.

□ 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. Also, without a separate execution-state, activation points must be managed explicitly, requiring the execution logic to be written as a series of cases, each ending by recording the next case to be executed on re-entry. However, explicit management of activation points is complex and error-prone, for more than a small number of activation points.

Simulating a coroutine with a class solves the problem of abstraction and does generalize to multiple instances, but does not handle the explicit management of activation points. Simulating a coroutine with a task, which also has an execution-state to handle activation points, is non-trivial because the organizational structure of a coroutine and task are different. Furthermore, simulating full-coroutines, which form a cyclic call-graph, may be impossible with tasks because of a task's mutual exclusion, which could cause deadlock (not a problem in μC++ because multiple entry is allowed by the same thread). 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() ...          // coroutine 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 appropriate coroutine constructor and any base-class constructors are executed in the normal order. The coroutine's execution-state is created and the starting point (activation point) is initialized to the coroutine's main routine visible by the inheritance scope rules from the coroutine type; 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 ensures that the starting 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. *Activating a terminated coroutine is an error.* A coroutine's destructor is invoked by the destroying thread when the block containing the coroutine declaration terminates or by an explicit **delete** statement for a dynamically allocated coroutine.

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. Therefore, it is suggested that these kinds of references be used with extreme caution.

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:

```
uCoroutine uBaseCoroutine {
  public:
    uBaseCoroutine();
    uBaseCoroutine( unsigned int stacksize );

    void *uStackPointer() const;            // stack information
    unsigned int uStackSize() const;
    ptrdiff_t uStackFree() const;
    ptrdiff_t uStackUsed() const;
    void uVerify();

    const char *uSetName( const char *name );
    const char *uGetName() const;
    enum uCoroutineState { uHalt, uActive, uInActive };
    uCoroutineState uGetState() const;

    uDualEvent uFailure;                    // exceptions
    uDualEvent uTerminated;
    uDualEvent uStackOverflow;
};
```

(The type **uDualEvent** is defined in Section 4.3.)

The overloaded constructor routine uBaseCoroutine has the following forms:

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

uBaseCoroutine(**unsigned int** stacksize) – creates a coroutine on the current cluster with the specified stack size (in bytes).

A coroutine type can be designed to allow declarations to specify the stack size 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 uStackPointer returns the address of the stack pointer saved at the last context switch of a coroutine; this is not the current stack pointer value.

The member routine uStackSize returns the maximum amount of stack space that is allocated for this coroutine. A coroutine cannot exceed this value during its execution.

The member routine uStackFree returns the amount of stack space free at the last context switch; this is not the current free stack space.

The member routine uStackUsed returns the amount of stack space used at the last context switch; this is not the current used stack space.

The member routine uVerify checks whether the current coroutine has overflowed its stack. If it has, the program terminates. To completely ensure the stack size is never exceeded, a call to uVerify must be included after each set of declarations, as in the following:

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

Thus, after a coroutine has allocated its local variables, a check is made that its stack has not overflowed. Clearly, this technique is not ideal and requires additional work for the programmer, but it does handle complex cases where the stack depth is difficult to determine and can be used to help debug possible stack overflow situations.

□ 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 and returns the previous name. The name is not copied so its storage must persist for the duration of the coroutine. The member routine uGetName returns the string name associated with a coroutine. If a coroutine has not been assigned a name, uGetName returns the type name of the coroutine. μC++ uses the name when printing any error message, which is helpful in debugging.

The member routine uGetState returns the current state of a coroutine's execution-state, which is one of the enumerated values uBaseCoroutine::uHalt, uBaseCoroutine::uActive or uBaseCoroutine::uInActive.

The free routine:

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

is used to determine the identity of the coroutine executing this routine. 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
}
```

As well, printing a coroutine's address for debugging purposes must be done like this:

```
uCout << "coroutine:" << &uThisCoroutine() << endl; // notice the ampersand (&)
```

2.6.3 Coroutine Control and Communication

Control flow among coroutines is specified by the **uResume** and **uSuspend** statements. The **uResume** statement may appear in any member of the coroutine, including the coroutine main, but normally it only appears in the public members. The **uSuspend** statement is used only within the coroutine main or non-public members called directly or indirectly from the coroutine main. Both **uResume** and **uSuspend** statements are composed of two parts. The first part inactivates the coroutine that invokes the statement and the second part reactivates another coroutine; the difference is which coroutine is reactivated. The **uResume** statement always activates the current coroutine object, i.e., the object specified by the implicit **this** variable. The **uSuspend** statement activates the coroutine that most recently executed a **uResume** statement on this coroutine. In effect, these statements cause control flow to transfer among execution states, which involves context switches.

It is important to understand that calling a coroutine's member from another coroutine does not cause a switch to the other coroutine. A switch only occurs when a **uResume** is executed in the other coroutine's member. Therefore,

printing &uThisCoroutine() in the other coroutine's member always prints the *calling* coroutine's address; printing **this** in the other coroutine's member always prints the *called* coroutine's address (which is the coroutine that **uResume** switches to). Hence, there is a difference between who is executing and where execution is occurring.

Figure 2.2 shows a semi-coroutine producer and consumer coroutine, and a driver routine. 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 C.1 shows a complex binary insertion sort implemented as a semi-coroutine.

Consumer

```
uCoroutine Cons {
    int p1, p2, status;     // communication
    bool done;

    void main() {
        // 1st resume starts here
        int money = 1;
        for ( ;; ) {
            uCout << "cons receives: " <<
                p1 << ", " << p2;
          if ( done ) break;
            status += 1;
            uCout << " and pays $" <<
                money << endl;
            uSuspend; // restart delivery & stop
            money += 1;
        } // for
        uCout << "cons stops" << endl;
    } // main
  public:
    Cons() : status(0), done(false) {}
    int delivery( int p1, int p2 ) {
        Cons::p1 = p1;
        Cons::p2 = p2;
        uResume;        // restart main
        return status;
    } // delivery
    void stop() {
        done = true;
        uResume;        // restart main
    } // stop
}; // Cons
```

Producer

```
uCoroutine Prod {
    Cons &cons;         // communication
    int N;

    void main() {
        // 1st resume starts here
        int i, p1, p2, status;
        for ( i = 1; i <= N; i += 1 ) {
            p1 = rand() % 100;
            p2 = rand() % 100;
            uCout << "prod delivers: " <<
                p1 << ", " << p2 << endl;
            status = cons.delivery( p1, p2 );
            uCout << "prod status: " <<
                status << endl;
        } // for
        uCout << "prod stops" << endl;
        cons.stop();
    } // main
  public:
    Prod( Cons &c ) : cons(c) {}
    void start( int N ) {
        Prod::N = N;
        uResume;        // restart main
    } // start
}; // Prod

void uMain::main() {
    Cons cons;          // create consumer
    Prod prod(cons);    // create producer
    prod.start(5);      // start producer
} // uMain::main
```

Figure 2.2: Semi-Coroutine Producer-Consumer

Figure 2.2 shows a full-coroutine producer and consumer coroutine, and a driver routine. 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 that mutex members are executed with mutual exclusion; that is, only one task at a time can be executing in the mutex members. Similar to an

Consumer

```
uCoroutine Cons {
    Prod &prod;             // communication
    int p1, p2, status;
    bool done;

    void main() {
        // 1st resume starts here
        int money = 1, receipt;
        for ( ;; ) {
            uCout << "cons receives: " <<
                p1 << ", " << p2;
          if ( done ) break;
            status += 1;
            uCout << " and pays $" <<
                money << endl;
            receipt = prod.payment( money );
            uCout << "cons receipt #" <<
                receipt << endl;
            money += 1;
        } // for
        uCout << "cons stops" << endl;
    } // Cons::main
  public:
    Cons( Prod &p ) : prod(p) {
        done = false;
        status = 0;
    } // Cons::Cons

    int delivery( int p1, int p2 ) {
        Cons::p1 = p1;    // restart cons in
        Cons::p2 = p2;    // Cons::main 1st time
        uResume;          // and afterwards cons
        return status;    // in Prod::payment
    } // Cons::delivery

    void stop() {
        done = true;
        uResume;
    } // Cons::stop
}; // Cons
```

Producer

```
uCoroutine Prod {
    Cons *cons;             // communication
    int N, money, receipt;


    void main() {
        // 1st resume starts here
        int i, p1, p2, status;
        for ( i = 1; i <= N; i += 1 ) {
            p1 = rand() % 100;
            p2 = rand() % 100;
            uCout << "prod delivers: " <<
                p1 << ", " << p2 << endl;
            status = cons->delivery( p1, p2 );
            uCout << "prod status: " <<
                status << endl;
        } // for
        uCout << "prod stops" << endl;
        cons->stop();
    } // main
  public:
    Prod() : receipt(0) {}

    int payment( int money ) {
        Prod::money = money;
        uCout << "prod payment of $" <<
            money << endl;
        uResume;          // restart prod
        receipt += 1;     // in Cons::delivery
        return receipt;
    } // payment

    void start( int N, Cons &c ) {
        Prod::N = N;
        cons = &c;
        uResume;
    } // start
}; // Prod

void uMain::main() {
    Prod prod;
    Cons cons( prod );
    prod.start( 5, cons );
} // main
```

Figure 2.3: Full-Coroutine Producer-Consumer

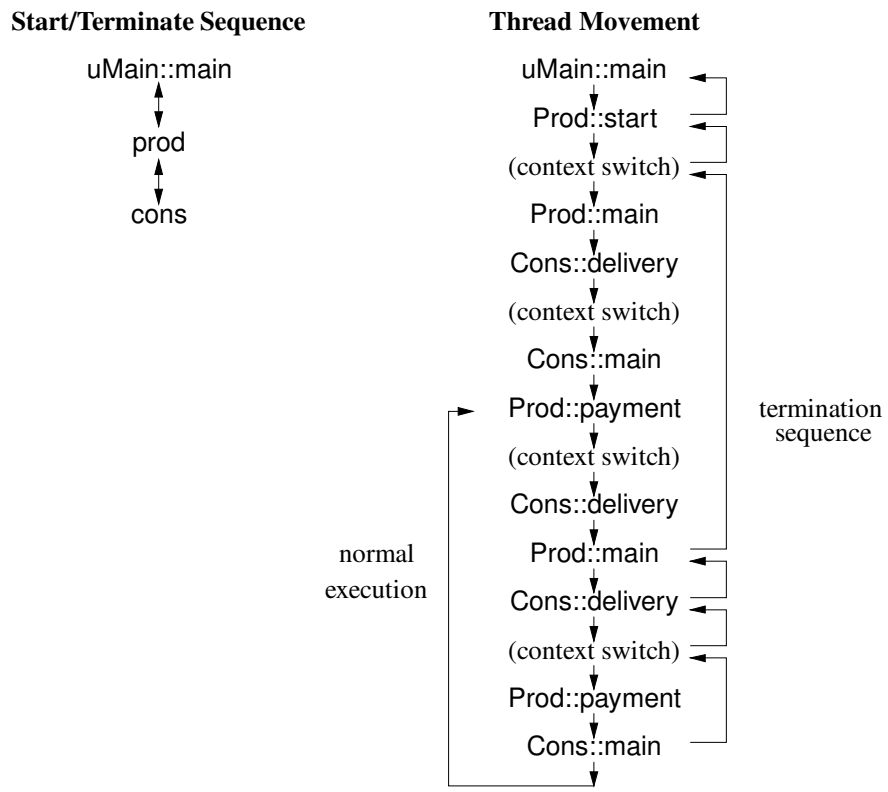


Figure 2.4: Cyclic Control Flow in Full-Coroutine

execution-state, a mutex object is either **active** or **inactive**, depending on whether or not a task is executing a mutex member (versus a task executing the coroutine main). 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 object becomes unlocked. *An active task may call other mutex members either directly from within the mutex type or indirectly by calling another object which subsequently calls back into the mutex object.* If an active task enters multiple mutex objects, it owns those objects and can enter anyone of them at any time. *However, if the active task releases control of one of these mutex objects (i.e., unlocks the object), it does not unlock any other objects it currently controls.*

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. Hence, public member routines M, ~M, x, y, and z of mutex type M are mutex members executing mutually exclusively of one another. Member routines that are **protected** and **private** are *always* implicitly **uNoMutex**, except for the destructor of a mutex type, which is always **uMutex** regardless of its visibility. *Because the destructor of a mutex type is always executed mutually exclusively, the call to the destructor may block, either at termination of a block containing a mutex object or when deleting a dynamically allocated mutex object.*

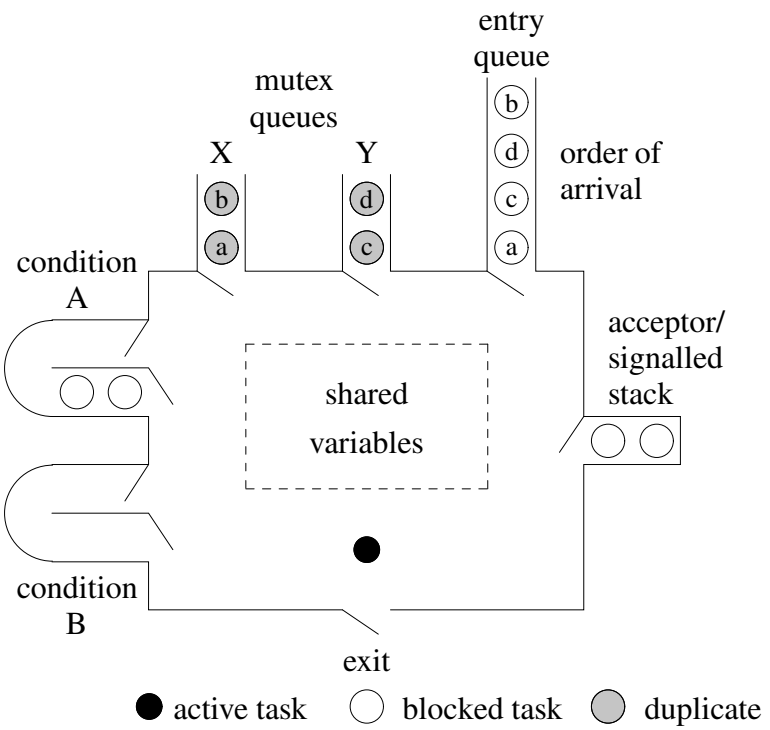


Figure 2.5: μC++ Mutex Object

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 task 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 type'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 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.

2.8 Scheduling

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

```
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 forming different scheduling patterns. 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 either implicitly by the implicit scheduler (discussed next) or explicitly by another (active) task.

2.8.1 Implicit Scheduling

Implicit scheduling occurs when a mutex object becomes unlocked because the active task blocks in or exits from a mutex member. The next task to use the mutex object is then chosen from one of a number of lists associated with 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 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 of arrival, which is important for selecting a task when there is no active task in a mutex object. When a task in the mutex object is blocked implicitly (see Section 2.8.2) or is reactivated by another (active) task (see Section 2.8.3), 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 implicit scheduler. For some of the following scheduling statements, the implicit scheduler is directed to select from a specific set of queues; hence, there is no choice with regard to which queues are examined. For other scheduling statements, the implicit scheduler may make a choice among the queues. When a choice is possible, the implicit scheduler for μC++ makes selections based on the results presented in [BFC95] to give the user the greatest possible control and produce efficient performance. These selection rules are:

1. Select tasks that have entered the mutex object, blocked, and now need to continue execution over tasks that have called and are waiting to enter.

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. (The **uSignalBlock** statement is an exception to this rule, see page 25.)

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, but is selected by the implicit scheduler.

2.8.2 External Scheduling

External scheduling controls state changes to a mutex object by scheduling calls to specified mutex members, which indirectly schedules tasks calling from *outside* the mutex object. This technique takes advantage of the entry queue to block tasks unconditionally when the mutex object is active (i.e., block outside) and the acceptor stack to block tasks conditionally that have entered the monitor (i.e., block inside). Much of the scheduling that occurs and the programmer thinks about is the outside scheduling from the entry queue rather than the internal scheduling on the acceptor stack, which occurs implicitly. External scheduling is accomplished with the accept statement.

2.8.2.1 Accept Statement

A **uAccept** statement dynamically chooses the mutex member(s) that executes next, which indirectly controls the next accepted caller, that is, the next caller to the accepted mutex member. The simple form of the **uAccept** statement is:

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

with the restriction that constructors, **new**, **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. That is, the object does not exist until after the **new** routine is completed and a constructor starts; similarly, the object does not exist when **delete** is called. In all these cases, member routines cannot be called, and hence accepted, because the object does not exist or is not initialized. **uNoMutex** members are excluded because they contain no code affecting the caller or acceptor with respect to mutual exclusion.

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

```
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 if 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. Therefore, a guard is not the same as an **if** statement, e.g.:

```
if ( count == 0 ) uAccept( mem );          uWhen ( count == 0 ) uAccept( mem );
```

In the right example, the program aborts if the conditional is false. (It is possible to simulate the **if** version using the **uElse** clause discussed below.) The *conditional-expression* of a **uWhen** may call a routine, **but the routine must not block or context switch.**

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 is 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                                   // optional statement
uOr uWhen ( conditional-expression )                // optional guard
    uAccept( mutex-member-name )
        statement                                   // optional 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 **uAccept**s indicates their relative priority for selection if there are several outstanding calls. Once the accepted call has completed *or the caller waits*, the statement after the accepting **uAccept** clause is executed and the accept statement is complete. *Notice, an accept statement accepts only one call, regardless of the number of mutex members listed in the statement.* 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. Again, a group of **uAccept** statements is not the same as a group of **if** statements, e.g.:

```
if ( c1 ) uAccept( mem1 );          uWhen ( c1 ) uAccept( mem1 );
else if ( c2 ) uAccept( mem2 );     uOr uWhen ( c2 ) uAccept( mem2 );
```

The left example, accepts only mem1 if c1 is true or only mem2 if c2 is true. The right example, accepts either mem1 or mem2 if c1 and c2 are true.

□ Note, 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 2.8.3.2). □

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

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

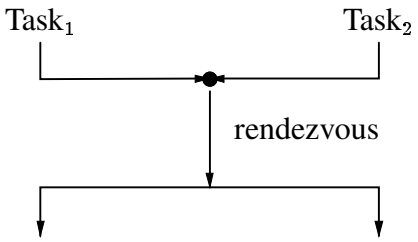
The left example accepts a call to either member fred or mary. The right example accepts a call to member fred if one is currently available, otherwise it accepts a call to member mary. The syntactic difference is subtle, and yet, their execution is significantly different. A less subtle syntactic difference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

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

□

2.8.2.2 Breaking a Rendezvous

The accept statement forms a **rendezvous** between the acceptor and the accepted tasks, where a rendezvous is a point in time at which both tasks wait for a section of code to execute before continuing.



The start of the rendezvous begins when the accepted mutex member begins execution and ends when the acceptor task restarts execution, either because the accepted task finishes executing of the mutex member *or the accepted task waits*. In the latter case, correctness implies sufficient code has been executed in the mutex member before the wait occurs for the acceptor to continue successfully. Finally, for the definition of rendezvous, it does not matter which task executes the rendezvous, but in μC++, it is the accepted task that executes it. It can be crucial to correctness that the acceptor know if the accepted task does not complete the rendezvous code, otherwise the acceptor task continues under the incorrect assumption that the rendezvous action has occurred. To this end, an asynchronous event is propagated to the acceptor task if the accepted member terminates abnormally (see Section 4.8.2);

2.8.2.3 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, which allows a mutex object to clean up before it terminates. (This semantics is the same as the **uSignal** statement, see page 25.) Only when the caller to the destructor is popped off the acceptor/signalled stack by the internal scheduler can the destructor execute. The destructor can reactivate any blocked tasks on the acceptor/signalled stack; at this point, the task behaves like a monitor because its thread is halted.

□ 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. Furthermore, 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
```

2.8.2.4 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 impossible 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 type's 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 arise only rarely.

2.8.3 Internal Scheduling

A complementary approach to external scheduling is internal scheduling. Instead of scheduling tasks from outside the mutex object from the entry queue (the entry queue is still necessary), most of the scheduling occurs inside the monitor. To do scheduling *inside* the monitor requires additional queues *inside* the monitor on which tasks can block and subsequently be unblocked by other tasks. For that purpose, condition variables are provided, with an associated wait and signal statement.

2.8.3.1 Condition Variables and Wait/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:
    bool uEmpty() const;
    long int uFront() const;

    uDualEvent uSuspendedFailure;
};
uCondition DiskNotIdle;
```

A condition variable is owned by the mutex object that performs the first wait on it; subsequently, only the owner can wait and signal that condition variable. The member routine uEmpty() returns **false** if there are tasks blocked on the queue and **true** otherwise. The member routine uFront returns an integer value stored with the waiting task at the front of the condition queue. It is an error to examine the front of an empty condition queue; therefore, a condition must be checked to verify that there is a blocked task, as in:

```
if ( ! DiskNotIdle.uEmpty() && DiskNotIdle.uFront() == 1 ) ...
```

(This capability is discussed in detail shortly.)

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 if not its owner.

□ 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 block on a condition queue, the active task in a mutex object executes a **uWait** statement, e.g.,

```
uWait DiskNotIdle;
```

This statement causes the active task to block on condition DiskNotIdle, which unlocks the mutex object and invokes the internal scheduler. Internal scheduling 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 scheduling did not accept a call at this point, deadlock would occur.

The **uWait** statement allows an optional integer value to be stored with each task on a condition queue by including a value after the condition, as in:

uWait *condition* **uWith** *integer-expression*

If no value is specified, the value for the blocked task is undefined. The integer value can be accessed by other tasks through the uCondition member routine uFront. This value can be used to provide more precise information about a waiting task than can be inferred from its presence on a particular condition variable. For example, the value of the front blocked task on a condition can be examined by a signaller to help make a decision about which condition variable it should signal next. This capability is useful, for example, in a problem like the readers and writer. (See Appendix C.2 for an example program using this feature, but only after reading Section 2.9 on monitors.) In that case, reader and writer tasks wait on the same condition queue to preserve FIFO (first-in first-out) order and each waiting task is marked with a value for reader or writer, respectively. A task that is signalling can first check if the awaiting task at the head of a condition queue is a reader or writer task by examining the stored value before signalling.

□ The value stored with a waiting task and examined by a signaller should not be construed as a message between tasks. The information stored with the waiting task is not meant for a particular task nor is it received by a particular task. Any task in the monitor can examine it. Also, the value stored with each task is *not* a priority for use in the subsequently selection of a task when the monitor is unlocked.

If this capability did not exist, it can be mimicked by creating and managing an explicit queue in the monitor that contains the values. Nodes would have to be added and removed from the explicit queue as tasks are blocked and restarted. Since there is already a condition queue and its nodes are added and removed at the correct times, it seemed reasonable to allow users to store some additional data with the blocked tasks. □

A task is reactivated from a condition variable when another (active) task executes a signal statement. There are two forms of signal statement. The first form is the **uSignal** statement, e.g.:

```
uSignal DiskNotIdle;
```

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

```
uSignalBlock DiskNotIdle;
```

The effect of a **uSignalBlock** statement is to remove one task from the specified condition variable and make it the active task, and push the signaller onto the acceptor/signalled stack. The signalled task continues execution and the signaller is scheduled by the internal scheduler when the mutex object is next unlocked. This semantics is *like* the **uAccept** statement, which always blocks the acceptor.

□ The **uAccept**, **uWait**, **uSignal** and **uSignalBlock** statements can be executed by any routine of a mutex type. Even though these statements block the current task, they can be allowed in any member routine because member routines are executed by the caller, not the task the member is defined in. This capability 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. □

2.8.3.2 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 a mutex 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 approach avoids complex selection expressions and possibly their repeated evaluation. In addition, all the normal programming language constructs and data structures can be used in the process of making a 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, it is essential that a request be postponable even after it is accepted because of any number of reasons during the servicing of the request. 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 95'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 may not be able to be sent directly from a mutex member to another mutex member because deadlock occurs due to 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 servicing a 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 include file uC++.h.

2.9.1 Monitor Creation and Destruction

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

```
uMutex class M {
  public:
    void r( ... ) ...       // mutex member
};
M *mp;                      // pointer to a M
{ // start a new block
    M m, ma[3];             // local creation
    mp = new M;             // dynamic creation
    ...
} // 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 mutex members of a 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++, both internal and external scheduling are provided, where most traditional monitors provide only internal scheduling. Figure 2.6 compares the traditional internal scheduling style using explicit condition variables to the external scheduling style uses accept statements. The problem is the exchange of values (telephone numbers) between two kinds of tasks (girls and boys). (While **uAccept** allows the removal of all condition variables in this case, this is not always possible.)

2.10 Coroutine-Monitor

The coroutine-monitor is a coroutine with mutual exclusion, making it safely accessible 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:

Internal Scheduling	External Scheduling

uMonitor DatingService {
  int GirlPhoneNo, BoyPhoneNo;
  uCondition GirlWaiting, BoyWaiting;
public:
  int Girl(**int** PhoneNo) {
  if (BoyWaiting.uEmpty()) {
  uWait GirlWaiting;
  GirlPhoneNo = PhoneNo;
  } **else** {
  GirlPhoneNo = PhoneNo;
  uSignalBlock BoyWaiting;
  } *// if*
  return BoyPhoneNo;
  } *// Girl*
  int Boy(**int** PhoneNo) {
  if (GirlWaiting.uEmpty()) {
  uWait BoyWaiting;
  BoyPhoneNo = PhoneNo;
  } **else** {
  BoyPhoneNo = PhoneNo;
  uSignalBlock GirlWaiting;
  } *// if*
  return GirlPhoneNo;
  } *// Boy*
}; *// DatingService*
	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*

Figure 2.6: Internal versus External Scheduling

```
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
};
```

The macro name **uCoMonitor** is defined to be "**uMutex uCoroutine**" in include file uC++.h.

2.10.1 Coroutine-Monitor Creation and Destruction

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

2.10.2 Coroutine-Monitor Control and Communication

A coroutine-monitor can make use of **uSuspend**, **uResume**, **uAccept** and uCondition variables, **uWait**, **uSignal** and **uSignalBlock** 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; this distinguished member is called the **task main**. Instead of allowing direct interaction with main, its visibility is normally **private** or **protected**. The decision to make the task main **private** or **protected** depends solely on whether derived classes can reuse the task main or must supply their own. Hence, a user interacts with a task indirectly through its member routines. This approach 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 global 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() ...         // task main
  public:
    void r( ... ) ...
};
T *tp;                      // pointer to a T task
{ // 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. The task's execution-state and thread are created and the starting point for the new thread (activation point) is initialized to the task's main routine visible by the inheritance scope rules from the task type. After this point, the creating task executes concurrently with the new task. The location of a task's variables—in the task's data area or in member routine main—depends on whether the variables must be accessed by member routines other than main. 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. At this point, the task becomes a monitor and can still be used in that form. 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 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.

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. Also, any **static** member variables declared within a task are shared among all instances of that task type. If a task makes global references or has **static** variables, there is the general problem of concurrent access to these shared variables.

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" 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, which is the same sort of protocol 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:

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

    void uYield( unsigned int times = 1 );
    uCluster &uMigrate( uCluster &uClus );
    uCluster &uGetCluster() const;

    enum uTaskState { uReady, uRunning, uBlocked };
    uTaskState uGetState() const;

    int uGetActivePriority();
    int uGetBasePriority();
};
```

The public member routines of uBaseCoroutine are inherited and have the same functionality (see Section 2.6.2).

The overloaded constructor routine uBaseTask has the following forms:

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

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

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

uBaseTask(uCluster &uClus, **unsigned int** stacksize) – creates a 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 stack size by doing the following:

```
uTask T {
  public:
    T() : uBaseTask( 8192 ) {};                    // current cluster, 8K stack
    T( unsigned int s ) : uBaseTask( s ) {};       // current cluster and user stack size
    T( uCluster &c ) : uBaseTask( c ) {};          // user cluster
    T( uCluster &c, unsigned 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 member routine routine uYield gives up control of the virtual processor to another ready task the specified number of times. For example, the call uYield(5) immediately returns control to the μC++ kernel and the next 4 times the task is scheduled for execution. If there are no other ready tasks, the yielding task is simply stopped and restarted 5 times (i.e., 5 context switches from itself to itself). uYield 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., uYield(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 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.

□ When the -yield option is used, calls to uYield(rand() % 3) are automatically inserted at the beginning of each member routine. □

Although uYield is a public member routine of every task type, one task cannot yield another task; a task can only yield itself because a task can only be yielded when it is running, which is true when a task yields itself. If one task could yield another, the yielded task may be ready or blocked, but in either of these states there is no virtual processor to yield. If the yielded task is running, it would have to be interrupted and blocked, but it may be performing a critical operation that cannot be interrupted. Attempting to make all cases work correctly and consistently is problematic and not particularly useful. Finally, the ability to perform such a powerful operation on a task without its permission seems unreasonable.

The member routine uMigrate allows a task to move itself from one cluster to another so that it can access resources dedicated to that cluster's processor(s), as in:

```
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 for the same reason as for uYield.

The member routine uGetCluster returns the current cluster a task is executing on.

The member routine uGetState returns the current state of a task, which is one of the enumerated values uBaseTask::uReady, uBaseTask::uRunning or uBaseTask::uBlocked.

The following member routines are used in real-time programming (see Chapter 6). The member routine uGetActivePriority returns the current active priority of a task, which is an integer value between 0 and 31. The member routine uGetBasePriority returns the current base priority of a task, which is an integer value between 0 and 31.

The free routine:

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

is used to determine the identity of the task executing this routine. Because it returns a reference to the base task type, uBaseTask, for 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 verify the stack or yield execution of the calling task by performing the following:

```
uThisTask().uVerify();
uThisTask().uYield();
```

As well, printing a task's address for debugging purposes must done like this:

```
uCout << "task:" << &uThisTask() << endl;   // notice the ampersand (&)
```

2.11.3 Task Control and Communication

A task can make use of **uAccept** and uCondition variables, **uWait**, **uSignal** and **uSignalBlock** to block and unblock tasks that enter it. Appendix C.4 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.

2.12 Commentary

Initially, every attempt was made to add the new μC++ types and statements by creating a library of **class** definitions that were used through inheritance and preprocessor macros. This approach has been used by others to provide coroutine facilities [Sho87, Lab90] and simple parallel facilities [DG87, BLL88]. However, after discovering many limitations with all library approaches, it was abandoned in favour of language extensions.

The most significant problem with all library approaches to concurrency is the lack of soundness and/or efficiency [Buh95]. A compiler many perform valid sequential optimizations that invalidate a correct concurrent program. Code movement, dead code removal, and copying values into registers are just some examples of optimizations that can invalidate a concurrent program, e.g., moving code into or out of a critical section, removing a timing loop, or copying a shared variable into a register making it invisible to other processors. To preserve soundness, it is necessary to identify and selectively turn off optimizations for those concurrent section of code that might cause problems. However, a programmer may not be aware of when or where a compiler is using an optimization that affects concurrency; only the compiler writer has that knowledge. Therefore, when using a concurrent library, a programmer can at best turn off all optimizations in an attempt to ensure soundness, which can have a significant performance impact on the remaining execution of the program, which is composed of large sections of sequential code that can benefit from the optimizations.

Even if a programmer can deal with the soundness/efficiency problem, there are other significant problems with attempting to implement concurrency via the library approach. In general, a 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.

On this approach, thread 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 a μC++ task's main) 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 forking of control 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 unknown. Another approach that does not rely on stack diddling while still allowing inheritance is to determine when all initialization is completed so that the new thread can be started. However, it is impossible in C++ (and most other object-oriented programming languages) for a constructor to determine if it is the last constructor executed in an inheritance chain. 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 (and now Java [GJS96]) 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 a destructor is 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. Implementing this operation is expensive since the addition or removal of a message to/from a queue must 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 [MMPN93], or wrapper routines, as in GNU C++ [Tie88], might be able to provide automatically any special member code. Furthermore, 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 completely unsatisfactory. We decided that language extensions would better suit our goals by providing soundness and efficiency, greater flexible and consistency with existing language features, and static checking.

2.13 Inheritance

C++ provides two forms of inheritance: "private" and "protected" inheritance, which provide 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.)

In C++, class definitions can inherit from one another using both single and multiple inheritance. In μC++, there are multiple kinds of types, class, mutex, coroutine, and task, so the situation is more complex. The problem is that mutex, coroutine and task types provide implicit functionality that cannot be arbitrarily mixed. 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, mutex, coroutine or task, and the coding styles used in each cannot, in general, be arbitrarily mixed. For example, an object produced by a task type 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, there is a significantly greater chance if users casually combine types of different kinds. Therefore, only the following forms of inheritance are allowed in μC++.

First, the trivial case of *single* inheritance among homogeneous kinds of type, i.e., a class or coroutine or task type inherits from another class or coroutine or task, respectively, is supported in μC++. In this situation, all implicit functionality matches between base and derived types, and therefore cannot cause problems.

Next, private and protected inheritance of a **struct/union/class** with any of the other types is allowed. Since private and protected inheritance is for code reuse only, and the basic types contain no implicit functionality, this form of inheritance does not affect the public interface of the derived type. Because the base type cannot be directly accessed from the derived type, any implicit functionality of the derived type is maintained.

In general, public inheritance among heterogeneous kinds of type is not supported, for reasons already mentioned. The exception is allowing a mutex class to inherit from a **struct/union/class**. The reason is that mutex objects already have both public variable and routine members without mutual exclusion (e.g., **uNoMutex** routine members). Therefore, allowing mutex objects to publicly inherit from the basic types introduces only situations than already exist.

As well, this form of inheritance is very useful to generate concurrent types from existing non-concurrent types, e.g., to define a mutex queue by deriving from a simple queue. To change a simple queue to a mutex queue requires a monitor to inherit from the class Queue and redefine all of the class's member routines so mutual exclusion occurs when they are invoked, as in:

```
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
```

However, there is a fundamental problem with non-virtual members in C++, which can cause significant confusion because non-virtual routine calls are statically bound. For example, routines Queue::insert and Queue::remove do not provide mutual exclusion because they are members of the class, while routines MutexQueue::insert and MutexQueue::remove do provide mutual exclusion because they are members of a mutex type. Because the pointer variable qp is of type Queue, the call qp->insert calls Queue::insert even though insert is 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 errors. In object-oriented programming languages that have only virtual member routines, this is not a problem. The problem does not occur with private or protected inheritance because no subtype relationship is created, and hence, the assignment to qp would be invalid.

Multiple inheritance is allowed for private/protected/public inheritance of **struct/union/class** with a mutex class, *with the restriction that at most one of the base classes in the hierarchy is a mutex class*. Multiple inheritance is not allowed among coroutines or tasks. Again, the main reason is that the coding styles of each cannot, in general, be arbitrarily mixed. For example, 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. It is doubtful that the coroutine's main routine has appropriate accept statements.

2.14 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**, **uSignal** and **uSignalBlock** are sufficient. Nevertheless, several low-level lock mechanisms are provided for teaching purposes and for special situations.

2.14.1 Counting Semaphore

A semaphore in μC++ is implemented as a counting semaphores as described by Dijkstra [Dij65]. 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:

```
uMonitor uSemaphore {
  public:
    uSemaphore( unsigned int value = 1 );
    void uV( unsigned int times = 1 );
    void uP();
    void uP( uSemaphore s );
    uNoMutex bool uEmpty();
};
uSemaphore x, y(1), *z;
z = new uSemaphore(4);
```

The declarations create three semaphores variables and initializes them to the value 1, 0, 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 values are ≥ 0. The default value is 1.

The member routines uP and uV are used to perform the classical counting semaphore operations. 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. 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 semaphore, that semaphore is Ved before Ping on the semaphore object; the two operations occur atomically.. The member routine uEmpty returns **false** if there are threads blocked on the semaphore and **true** 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).

To use counting semaphores in a μC++ program, include the file:

```
#include <uSemaphore.h>
```

2.14.1.1 Commentary

The wait and signal operations on conditions are very similar to the 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 (unblock) 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 unblock multiple tasks, but only one of the these tasks is able to execute because of the mutual exclusion property of the mutex type.

2.14.2 Owner Lock

An owner lock is owned by the task that acquires it; all other tasks wanting the lock block 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. (Owner locks are used in the implementation of the non-blocking I/O stream library, see Section 3.2.1). The type uOwnerLock defines an owner lock:

```
uMonitor uOwnerLock {
  public:
    uOwnerLock();
    void uAcquire();
    bool uTryAcquire();
    void uRelease();
};
uOwnerLock x, y, *z;
z = new uOwnerLock;
```

The declarations create three owner locks and initializes them to open.

The 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 blocks until the lock is released by the current owner. The member routine uTryAcquire makes one attempt to try to acquire the lock, i.e., it does not block. uTryAcquire returns **true** if the lock is acquired and **false** otherwise. 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).

To use owner locks in a μC++ program, include the file:

```
#include <uOwnerLock.h>
```

2.14.3 Lock

A lock is either closed (0) or opened (1), 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 (spin) 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 usually not a problem.

The type uLock defines a lock:

```
class uLock {
  public:
    uLock( unsigned int value = 1 );
    void uAcquire();
    bool uTryAcquire();
    void uRelease();
};
uLock x, y, *z;
z = new uLock( 0 );
```

The declarations create three locks variables and initializes the first two to open and the last to closed.

The constructor routine uLock has the following form:

uLock(**int** value) – this form specifies an initialization value for the lock. Appropriate values are 0 and 1. The default value is 1.

The 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 until it can acquire the lock. The member routine uTryAcquire makes one attempt to try to acquire the lock, i.e., it does not spin waiting. uTryAcquire returns **true** if the lock is acquired and **false** otherwise. 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).

2.14.4 Barrier

A barrier allows N tasks to synchronize, possible multiple times, during their life time. Barriers are used to repeatedly coordinate a group of tasks performing a concurrent operation followed by a sequential operation. In μC++, a barrier is a mutex coroutine to provide the necessary mutual exclusion and to allow code to be easily executed both before and after the N tasks synchronize on the barrier. The type uBarrier defines a barrier:

```
uMutex uCoroutine uBarrier {
  protected:
    virtual void main() {
        for ( ;; ) {
            uSuspend;
        }
    }
  public:
    uBarrier( unsigned int total );
    void uReset( unsigned int total );
    uNoMutex unsigned int uTotal() const;
    uNoMutex unsigned int uWaiters() const;
    void uBlock();
    virtual void uLast() {
        uResume;
    }
};
uBarrier x(10), *y;
y = new uBarrier( 20 );
```

The declarations create two barrier variables and initializes the first to work with 10 tasks and second to work with 20 tasks.

The constructor routine uBarrier has the following form:

uBarrier(**unsigned int** total) – this form specifies the total number of tasks participating in the synchronization. Appropriate values are ≥ 0.

The member routine uReset changes the total number of tasks participating in the synchronization; no tasks may be waiting in the barrier when the total is changed. uBlock is called to synchronize with the other N tasks; tasks block until all N tasks have called uBlock. The member routines uTotal and uWaiters return the total number of tasks participating in the synchronization and the total number of tasks currently waiting at the barrier, respectively. The virtual member routine uLast is called by the last task to synchronize at the barrier. It can be replaced by subclasses of uBarrier to provide a specific action to be executed when synchronization is complete. This capability is often used to reset a computation before releasing the tasks from the barrier to start the next computation. The default code for uLast is to resume the coroutine monitor.

The protected member routine main is usually replaced by subclassing to supply the code to be executed before and after tasks synchronize. The general form for a barrier main routine is:

```
void main() {
    // code executed before synchronization (initialization)
    uSuspend;
    // code executed after synchronization (termination)
}
```

Often, the constructor for the subclass would **uResume** its coroutine main at the very end to prime the barrier's initialization. When main suspends back to the constructor, the barrier is initialized and ready to synchronize the first set of tasks.

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

To use barriers in a μC++ program, include the file:

```
#include <uBarrier.h>
```

2.15 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 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 the 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 active when the declaration of the context save-area occurs.

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

```
class uContext {
  public:
    uContext();
    uContext( void *key );
    virtual void uSave() = 0;
    virtual void uRestore() = 0;
}; // uContext
```

The overloaded constructor routine uContext has the following forms:

uContext() – creates the context with a unique search key (discussed shortly).

uContext(**void** *key) – creates the context with the user supplied search key.

Multiple context areas can be declared, and hence, associated with a coroutine or task. However, a context is only associated with an execution-state if its search key is unique. This requirement prevents the same context from being associated multiple times with a particular coroutine or task.

Figure 2.7 shows how the context of a hardware co-processor can be saved and restored as part of the context of task worker. A unique search-key for all instances of CoProcessorCxt is created via the address of the static variable, uUniqueKey, because the address of a static variable is unique within a program. Therefore, the value assigned to uUniqueKey is irrelevant, but a value must be assigned in one translation unit for linking purposes. This address is implicitly stored in each instance of CoProcessorCxt. When a context is added to a task, a search is performed for any context with the same key. If a context with the same key is found, the new context is not added; otherwise it is added to the list of user contexts for the task.

□ **WARNING:** Put no code into routines uSave and uRestore that results in a context switch, e.g., printing using uCout or uCerr (use printf if necessary). These routines are called during a context switch, and a context switch cannot be recursively invoked. □

2.15.1 Predefined Floating Point Context

In most operating systems, 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 programs 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.

To save and restore the float-point registers on a context switch, declare a single instance of the predefined 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:

```
class CoProcessorCxt : public uContext {
    static int uUniqueKey;              // unique address across all instances
    int reg[3];                         // co-processor has 3 integer registers
  public:
    CoProcessorCxt() : uContext( &uUniqueKey ) {};
    void uSave();
    void uRestore();
};

int CoProcessorCxt::uUniqueKey = 0;   // must initialize in one translation unit

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

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

Figure 2.7: Saving Co-processor Context

```
uCoroutine C {
    void main() {
        uFloatingPointContext fpcxt;          // the name of the variable is insignificant
        ... // 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. □

□ **WARNING:** Some processors, like the SPARC, implicitly save both fixed and floating-point registers, which means it is unnecessary to create instances of uFloatingPointContext in tasks performing floating-point operations. However, leaving out uFloatingPointContext is dangerous because the program is not portable to other processors. Therefore, it is important to always include an instance of uFloatingPointContext in tasks performing floating-point operations. For processors like the SPARC, uFloatingPointContext does nothing, so there is no cost. □

Additional context can be associated with a coroutine or task in a free routine, member routine, or as part of a class-object 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 capability 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. Remember, if the floating-point registers are already being saved, the additional association is ignored because of the unique search key.

2.16 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 exist simply because time limitations on this project have prevent it from being implemented.

- While μC++ has extended C++ with concurrency constructs, it is not a compiler. Therefore, it suffers from the soundness/efficiency problem related to all concurrency library approaches (see Section 2.12). To mitigate soundness problems, μC++ implicitly turns on or off compiler optimizations know to cause soundness problems. For example, with g++ compiler, the flags -fvolatile and -fvolatile-global are implicitly inserted by the u++ command to ensure that shared variables are not stored in registers. Unfortunately, turning on these flags affects all variables, and hence, prevents many valid optimizations. Since, it is virtually impossible to determine whether a variable is or is not shared by multiple tasks, it is necessary to take such Draconian measures to ensure that correct concurrent programs are sound.

- 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 problem is an artifact of μC++ being a translator.

- μC++ allows at most 128 mutex members because a 128 bit mask is used to test for accepted member routines.

  Unfortunately, bit masks, in general, do not extend to support multiple inheritance. However, 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, e.g.:

  ```
  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 problem is an artifact of μC++ being a translator.

- Anonymous coroutine and task types are not supported, e.g.:

```
uTask /* no name */ {          // must have a name
    ...
} t1, t2, t3;
```

  Both a coroutine and a task must have a constructor and destructor, which can only be created using the name of the type constructor. Having the translator generate a hidden unique name is problematic because the order of include files may cause the generation of a different name for different compilations, which plays havoc with linking because of name mangling.

- It is impossible to inherit from a **typedef** name. Currently, the translator cannot locate the synonym for the type name to perform necessary checking.

- There is no discrimination mechanism in the **uAccept** statement to differentiate among overloaded mutex member routines. When time permits, a scheme using a formal declarer in the **uAccept** statement to disambiguate overloaded member routines will be implemented, e.g.:

```
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.

- A **try** block surrounding a constructor body is not supported, as in:

```
class fred : public mary {
    const int i;
  public:
    fred();          // constructor
};
fred::fred() try : mary(3), i(27) {
    // body of constructor
} catch {
    // handle exceptions from initialization constructors (e.g., mary)
}
```

  This problem is an artifact of μC++ being a translator.

Chapter 3

Input/Output

3.1 Interaction with the File System

A major problem with concurrency and the file system is that, like the compiler, the file system does not know a program is concurrent (see Section 2.12). For a sequential program performing an I/O operation that cannot proceed immediately, the normal action for the file system is to block the program until the operation can continue. For example, when a program needs input from the keyboard, the file system blocks the program until data is entered. This action is correct for a sequential program because there is no other work for it to do until the new data is supplied by the user. However, this action may be incorrect for a concurrent program because there may be other work to do even without the user data. Therefore, the normal action by the file system, called heavy blocking (see Section 5.4.3), is usually inappropriate for a concurrent program because it inhibits concurrency. Therefore, I/O operations must be transformed from heavy blocking to light blocking so that execution of other tasks can continue. This transformation is achieved by nonblocking I/O. To simplify the complexity of nonblocking I/O, μC++ supplies a nonblocking I/O library.

3.1.1 Unikernel File Operations

While I/O operations can be made nonblocking, this requires special action as the nonblocking I/O operations may not restart automatically when the operation completes. Instead, it may be necessary to poll for I/O completions, which is done through the select operation in UNIX, while other tasks execute. Only when all tasks on a cluster are directly or indirectly (light) blocked, waiting for I/O operations to complete, can the virtual processor be heavy blocked.

This scenario is implemented automatically by μC++ choosing a task performing I/O to poll for completion of *any* I/O operation; all other tasks performing I/O are light blocked. When an I/O operation completes (e.g., a read or write), the task waiting for that operation is unblocked by the poller. If the poller's I/O completes, it unblocks one of the I/O blocked tasks and that task becomes the I/O poller. Only when the poller detects that no I/O operations have completed and there are no tasks on the cluster to execute (i.e., the cluster's ready queue is empty) does the poller perform a heavy block. This scheme allows other tasks to progress with only a slight degradation in performance due to the polling task.

3.1.2 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 lack of sharing means that if a task opens a file on one virtual processor, it cannot 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, called an I/O cluster. 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 the simple case, an I/O cluster is implicitly created when a file is opened ("file" is used generically for a stream, file or socket object); hence, each open file has a corresponding UNIX process. A user task performs an I/O operation

by calling the appropriate member routine of the file, which implicitly migrates the task to the I/O cluster containing the file descriptor and performs the actual operation. When the user task closes the file, the I/O cluster and all its resources are released. To ensure that multiple tasks are not performing I/O operations simultaneously on the same file descriptor, each file is implemented as a monitor that provides mutual exclusion of I/O operations. Figure 3.1 illustrates the runtime structures created for the simple case of accessing a file on an I/O cluster.

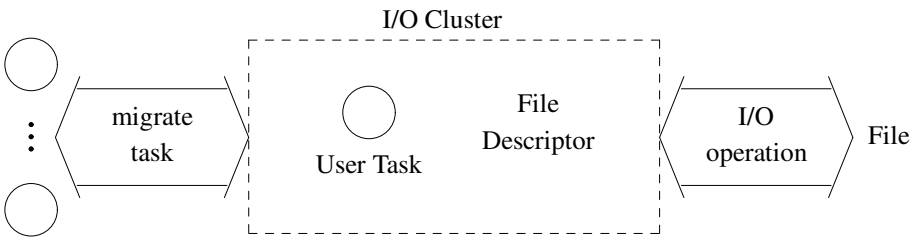


Figure 3.1: Simple I/O Cluster

In the simple case, 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 situation is not a problem if all tasks are reading the file, but does 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.

In the complex case, an explicit I/O cluster is created and multiple files are opened on that cluster. This case is useful to reduce the number of UNIX processes created to perform I/O. For example, the standard I/O files, called uCin, uCout and uCerr in μC++, are all opened on the system cluster, which is an I/O cluster. Figure 3.2 illustrates the runtime structures created for the complex case of accessing multiple files on an I/O cluster.

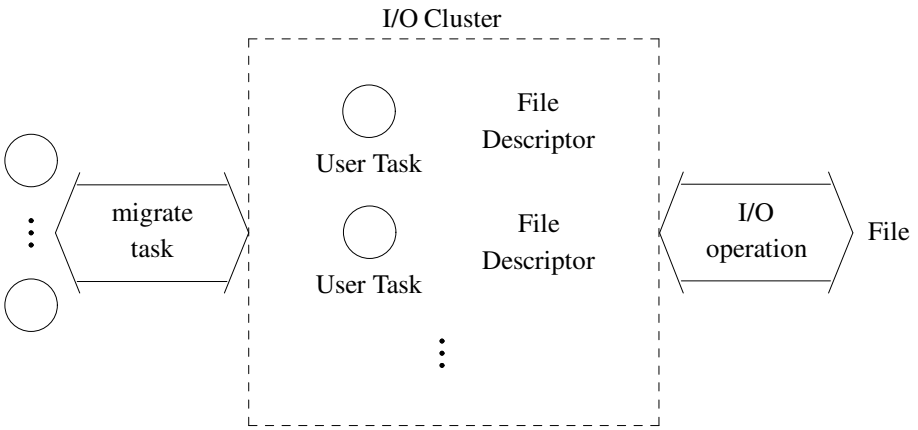


Figure 3.2: Complex I/O Cluster

The I/O cluster interface is the following:

```
class uIOCluster : public uCluster {
  public:
    uIOCluster( unsigned int stacksize = uDefaultStackSize(), const char *name = "uIOCluster" );
    uIOCluster( const char *name );

    static const int uReadSelect;
    static const int uWriteSelect;
    static const int uExceptSelect;

    int uSelect( int fd, int rwe, timeval *timeout = NULL );
    int uSelect( fd_set *rfd, fd_set *wfd, fd_set *efd, timeval *timeout = NULL );
    int uSelect( int nfds, fd_set *rfd, fd_set *wfd, fd_set *efd, timeval *timeout = NULL );
};
```

The overloaded constructor routine uIOCluster has the following forms:

uIOCluster(**unsigned int** stacksize = uDefaultStackSize(), **const char** *name = "uIOCluster") – this form uses the user specified stack size and cluster name (see Section 5.5 for the first default value).

uIOCluster(**const char** *name) – this form uses the user specified name for the cluster and the current cluster's default stack size.

When an I/O cluster terminates, it must have no tasks executing on it, which is the same requirement for any cluster (see Section 5.3). It is an error to add processors to an I/O cluster; only one processor is allowed, and it is created and destroyed implicitly when the I/O cluster is created and destroyed. The reason for this restriction is discussed in Section 3.1.2, i.e., some UNIX systems have non-shared file descriptors among virtual processors (i.e., UNIX processes).

The overloaded member routine uSelect works like the UNIX select routine, but on a per task basis. In general, uSelect is used only in esoteric situations, e.g., when μC++ file objects are mixed with standard UNIX file objects on the same I/O cluster. These members return the total number of file descriptors set in all file descriptor masks, and each routine has the following form:

uSelect(**int** fd, **int** rwe, timeval *timeout = NULL) – this form is a shorthand select for a single file descriptor. The mask, rwe, is composed of logically "or"ing flags uReadSelect, uWriteSelect, and uExceptSelect. The timeout value points to a maximum delay value, specified as a timeval, to wait for the I/O to complete. If the timeout pointer is null, the select blocks until the I/O operation completes or fails. This form is more efficient than the next forms with complete file descriptor sets, but handles only a single file.

uSelect(fd_set *rfd, fd_set *wfd, fd_set *efd, timeval *timeout = NULL) – this form examines *all* I/O file descriptors in the sets pointed to by rfd, wfd, and efd, respectively, to see if any of their file descriptors are ready for reading, or writing, or have exceptional conditions pending. The timeout value points to a maximum delay value, specified as a timeval, to wait for an I/O to complete. If the timeout pointer is null, the select blocks until one of the I/O operations completes or fails.

uSelect(**int** nfds, fd_set *rfd, fd_set *wfd, fd_set *efd, timeval *timeout = NULL) – same as above, except only the first nfds I/O file descriptors in the sets are examined.

There does not seem to be any standard semantic action when multiple kernel threads access the same file descriptor in select. Some systems wake all kernel threads waiting for the same file descriptor; others wake the kernel threads in FIFO order of their request for the common file descriptor. μC++ adopts the former semantic and wakes all tasks waiting for the same file descriptor. In general, this is not a problem because *all* μC++ file routines retry their I/O operation, and only one succeeds in obtaining data (which one is non-deterministic).

Finally, it is impossible to precisely deliver uSelect errors to the task that caused it. For example, if one task in waiting for I/O on a file descriptor and another task closes the file descriptor, the UNIX select fails but with no information about which file descriptor caused the error. Therefore, μC++ wakes up all tasks waiting on the uSelect at the time of the error and the tasks must retry their I/O operation. Again, *all* μC++ file routines retry their I/O operations after waiting on uSelect.

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

□ 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, the poller task has to wake up once a second to deal with the rare event that a signal sent to wake it up is missed. This problem only occurs when a task is migrating from one cluster to another cluster on which I/O is being performed. □

3.2 μC++ Stream Library

The following interface has been provided to use C++ streams. These μC++ objects have almost identical behaviour to their C++ stream equivalents, but in addition, each I/O operation is performed mutually exclusively and in a nonblocking manner.

3.2.1 uIStream and uOStream I/O

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

- The standard C++ stream objects cin, cout and cerr are now called uCin, uCout and uCerr, respectively.
- All of the member routines for stream objects have mutual exclusion through an owner lock (see Section 2.14.2) to control stream locking.

  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:

  ```
  task1
  uCout << "abc " << "def ";
  task2
  uCout << "uvw " << "xyz ";
  ```

  some of the different outputs that can appear are:

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

- Additional manipulator routines are defined for controlling mutual exclusion of a stream across a sequence of operations:

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

  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:

  ```
  task1
  uCout << uAcquire << "abc " << "def " << uRelease;
  task2
  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 (because of the owner lock) 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 blocks until the task holding the lock is unblocked and releases it. This scenario 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;
```

- 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).

To use the stream interface, include one of following files:

```
#include <uOStream.h>      // uOStream only
#include <uIStream.h>      // uIStream only
#include <uIOStream.h>     // both uOStream and uIStream
```

at the beginning of each source file.

3.2.1.1 Commentary

Existing types with I/O operators << and >> for streams istream and ostream do not work with streams uIStream and uOStream, respectively. The problem is that uIStream and uOStream are not subtypes of istream and ostream because stream mutual exclusion cannot be ensured. For example, assuming the subtyping relation did exist and a library type string has I/O routines:

```
istream &operator>>( istream &, string & );
ostream &operator<<( ostream &, const string & );
```

The following example:

```
string s;
uCin  >> s;      // rewrite rules: >>( uCin, s );
uCout << s;      // rewrite rules: <<( uCout, s );
```

would be allowed because of the subtyping relationship, but neither of the I/O free routines for string acquire and release mutual exclusion on the streams, and hence, may cause the program to fail. There is also a contra-variance problem with cascading the I/O operators << and >> because the return type is not a subtype of subsequent stream parameters, but this is a secondary issue compared to lack of mutual exclusion. Hence, uIStream and uOStream cannot be subtypes of istream and ostream.

The only solution is to create a pair of I/O free routines for each library type used in a μC++ application, e.g.:

```
class uIFStream : public uIStream {
  public:
    uIFStream();
    uIFStream( const char *name, int mode = ios::in, int prot = 0644 );
    uIFStream( const char *name, uIOCluster &cluster, int mode = ios::in, int prot = 0644 );
    uIFStream( int fd, uIOCluster &cluster );
    uIFStream( int fd, uIOCluster &cluster, char *buf, int bufsize );
    void attach( int fd, uIOCluster &cluster );
    void open( const char *name, int mode = ios::in, int prot = 0644 );
    void open( const char *name, uIOCluster &cluster, int mode = ios::in, int prot = 0644 );
    void close();
    uFilebuf *rdbuf();
}; // uIFStream

class uOFStream : public uOStream {
  public:
    uOFStream();
    uOFStream( const char *name, int mode = ios::out, int prot = 0644 );
    uOFStream( const char *name, uIOCluster &cluster, int mode = ios::out, int prot = 0644 );
    uOFStream( int fd, uIOCluster &cluster );
    uOFStream( int fd, uIOCluster &cluster, char *buf, int bufsize );
    void attach( int fd, uIOCluster &cluster );
    void open( const char *name, int mode = ios::out, int prot = 0644 );
    void open( const char *name, uIOCluster &cluster, int mode = ios::out, int prot = 0644 );
    void close();
    uFilebuf *rdbuf();
}; // uOFStream
```

Figure 3.3: uIFStream and uOFStream Interface

```
uIStream &operator>>( uIStream &is, string &s ) {
    uIosWrapper wrapper( is );              // establish mutual exclusion

    is.stream() >> s;                       // return underlying istream
    return is;
}
uOStream &operator<<( uOStream &os, const string s ) {
    uIosWrapper wrapper( os );              // establish mutual exclusion

    os.stream() << s;                       // return underlying ostream
    return os;
}
```

The new I/O free routines use the uIosWrapper class to establish mutual exclusion for the stream. (The acquiring and releasing of the mutual exclusion occurs in the constructor and destructor of the uIosWrapper instance wrapper.) Then, the underlying C++ stream is returned from its μC++ form, and overloading selects the I/O free routine for the particular type.

3.2.2 uIFStream and uOFStream I/O

The types uIFStream and uOFStream (see Figure 3.3) are similar to the C++ classes ifstream and ofstream, with the same exceptions as for types uIStream and uOStream. The only additions are the ability to specify the I/O cluster on which the stream's file descriptor resides.

To use the uFStream I/O interface, include the file:

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

at the beginning of each source file.

3.2.3 uIOManip Manipulators

The I/O manipulators for stream files, e.g., setbase, setprecision, etc., are available in the include file:

```
#include <uIOManip.h>
```

3.3 UNIX File I/O

The following interface has been provided to use UNIX files. 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 );
    uFile( const char *name, uIOCluster &cluster );

    const char *uGetName() const;
    void status( struct stat &buf );

    uDualEvent uFailure;
    uDualEvent uTerminateFailure;
    uDualEvent uStatusFailure;
}
```

The parameters for the first and second constructors of uFile are as follows. The name parameter is the UNIX name of the file, which is connected to the program. The optional I/O cluster is where the file descriptor is created. 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 member routine uGetName returns the string name associated with a file.

The parameter for member routine 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 a file object is still inaccessible after a connection is made, there are no member routines to access its contents.

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 file: <fcntl.h>

3.3.1 File Access

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

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

```
class uFileAccess {
  public:
    uFileAccess( uFile &f, int flags, int mode = 0644 );

    int read( char *buf, int len, uDuration *timeout = NULL );
    int readv( const struct iovec *iov, int iovcnt, uDuration *timeout = NULL );
    int write( char *buf, int len, uDuration *timeout = NULL );
    int writev( const struct iovec *iov, int iovcnt, uDuration *timeout = NULL );
    off_t lseek( off_t offset, int whence );
    int fsync();

    uDualEvent uFailure;
    uDualEvent uOpenFailure;
    uDualEvent uCloseFailure;
    uDualEvent uSeekFailure;
    uDualEvent uSyncFailure;
    uDualEvent uReadFailure;
    uDualEvent uReadTimeout;
    uDualEvent uWriteFailure;
    uDualEvent uWriteTimeout;
};
```

Figure 3.4: uFileAccess Interface

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 listed in Figure 3.4:

The parameters for the constructor uFileAccess are as follows. The f parameter is a uFile object 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 member routines read, readv, write, writev, 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.) The only exception is the optional parameter timeout, which points to a maximum waiting time for completion of the I/O operation before aborting the operation by raising an abnormal event (see Section 6.3.2). (The type uDuration is defined in Section 6.2.) Appendix C.5 shows reading and writing to UNIX files.

3.4 BSD Sockets

The following interface has been provided to use BSD sockets. A socket is an end point in connecting and communicating among tasks in different UNIX processes, possibly on different computers. A socket endpoint is accessed as a *client*, which is one to many for connectionless communication with multiple servers, or one to one for peer connection communication with a server's acceptor. Alternatively, a socket endpoint is accessed as a *server*, which is one to many for connectionless communication with multiple clients, or one to one for peer connection communication with a server's *acceptor*. The relationship between connectionless and peer connected communication is shown in Figures 3.6 and 3.5. The dashed lines show the connect/accept of the client and server for peer communication. The dotted lines show the creation of an acceptor to service the connection for peer communication. The solid lines show the bidirectional communication between client and server/acceptor. A server socket has a name, either a character string for UNIX pipes and/or port-number/machine for an INET address, that clients must know to communicate. For connectionless communication, the server usually has a reader task that receives messages containing the client's address. The message can be processes by the reader task or given to a worker task to process, which subsequently returns a reply using the client's address. For peer connection communication, the server usually has one task in a loop accepting connections from clients, and each acceptance creates an acceptor task. The acceptor task receives messages from only one client, processes the message and subsequently returns a reply, in parallel with accepting clients. Since the acceptor and client are connected, communicated messages do not contain client addresses. These relationships are represented in a μC++ program by declarations of client, server and acceptor objects, respectively.

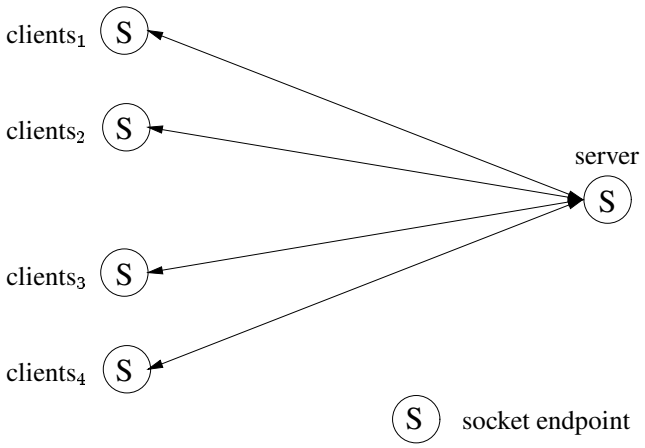


Figure 3.5: Client/Server Connectionless

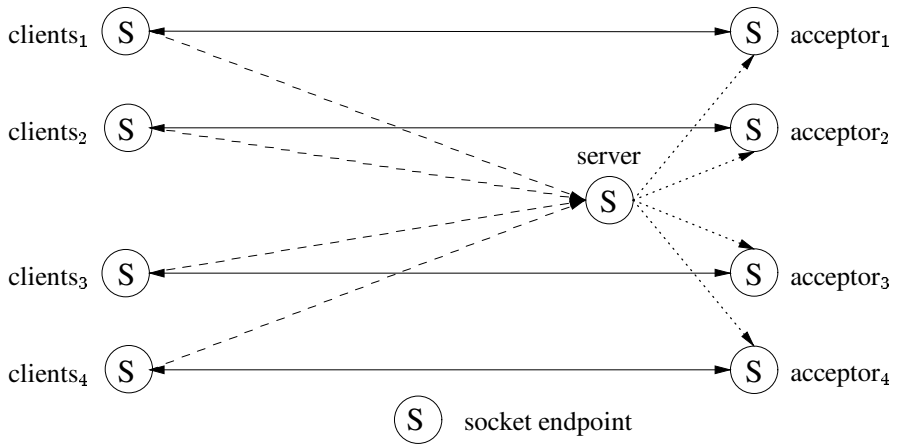


Figure 3.6: Client/Server Peer Connected

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/fcntl.h>, <sys/types.h>, <sys/socket.h>,<sys/un.h>, <netdb.h>.

3.4.1 Client

In μC++, a client, its socket endpoint and a connection to a server are created by declaration of a uSocketClient object, e.g.:

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

which creates a client variable, client, connected to the UNIX server socket, abc. The operations provided by uSocketClient are listed in Figure 3.7:

The first 4 constructors of uSocketClient are for use with the UNIX address family. The parameters for the constructors are as follows. The name parameter is the name of an existing UNIX stream that the client is connecting to. The type and protocol parameters are explained in the UNIX manual entry for socket. Only types SOCK_STREAM and SOCK_DGRAM communication can be specified, and any protocol appropriate for the specified communication type (usually 0). The optional parameter timeout is a pointer to a maximum waiting time for completion of a connection for type SOCK_STREAM before aborting the operation by raising an abnormal event (see Section 6.3.2); this parameter

```
class uSocketClient {
  public:
    // AF_ UNIX
    uSocketClient( const char *name, int type = SOCK_STREAM, int protocol = 0 );
    uSocketClient( const char *name, uDuration *timeout, int type = SOCK_STREAM, int protocol = 0 );
    uSocketClient( const char *name, uIOCluster &cluster, int type = SOCK_STREAM, int protocol = 0 );
    uSocketClient( const char *name, uIOCluster &cluster, uDuration *timeout, int type = SOCK_STREAM,
        int protocol = 0 );
    // AF_ INET, local host
    uSocketClient( unsigned short port, int type = SOCK_STREAM, int protocol = 0 );
    uSocketClient( unsigned short port, uDuration *timeout, int type = SOCK_STREAM, int protocol = 0 );
    uSocketClient( unsigned short port, uIOCluster &cluster, int type = SOCK_STREAM, int protocol = 0 );
    uSocketClient( unsigned short port, uIOCluster &cluster, uDuration *timeout, int type = SOCK_STREAM,
        int protocol = 0 );
    // AF_ INET, other host
    uSocketClient( unsigned short port, const char *name, int type = SOCK_STREAM, int protocol = 0 );
    uSocketClient( unsigned short port, const char *name, uDuration *timeout, int type = SOCK_STREAM,
        int protocol = 0 );
    uSocketClient( unsigned short port, const char *name, uIOCluster &cluster, int type = SOCK_STREAM,
        int protocol = 0 );
    uSocketClient( unsigned short port, const char *name, uIOCluster &cluster, uDuration *timeout,
        int type = SOCK_STREAM, int protocol = 0 );

    int getsockname( sockaddr *name, int *len );
    int getpeername( sockaddr *name, int *len );

    int read( char *buf, int len, uDuration *timeout = NULL );
    int readv( const struct iovec *iov, int iovcnt, uDuration *timeout = NULL );
    int write( char *buf, int len, uDuration *timeout = NULL );
    int writev( const struct iovec *iov, int iovcnt, uDuration *timeout = NULL );
    int send( char *buf, int len, int flags = 0, uDuration *timeout = NULL );
    int sendto( char *buf, int len, int flags = 0, uDuration *timeout = NULL );
    int sendto( char *buf, int len, void *to, int tolen, int flags = 0, uDuration *timeout = NULL );
    int sendmsg( const msghdr *msg, int flags = 0, uDuration *timeout = NULL );
    int recv( char *buf, int len, int flags = 0, uDuration *timeout = NULL );
    int recvfrom( char *buf, int len, int flags = 0, uDuration *timeout = NULL );
    int recvfrom( char *buf, int len, void *from, int *fromlen, int flags = 0, uDuration *timeout = NULL );
    int recvmsg( msghdr *msg, int flags = 0, uDuration *timeout = NULL );

    uDualEvent uFailure;
    uDualEvent uOpenFailure;
    uDualEvent uOpenTimeout;
    uDualEvent uCloseFailure;
    uDualEvent uReadFailure;
    uDualEvent uReadTimeout;
    uDualEvent uWriteFailure;
    uDualEvent uWriteTimeout;
};
```

Figure 3.7: uSocketClient Interface

is only applicable for peer connection, SOCK_STREAM, communication. The optional parameter cluster is where the socket for the client is created.

The next 4 constructors of uSocketClient are for use with the INET address family on a local host. The parameters for the constructors are as follows. The port parameter is the port number of an INET port on the local host machine. The type and protocol parameters are explained in the UNIX manual entry for socket. Only types SOCK_STREAM and SOCK_DGRAM communication can be specified, and any protocol appropriate for the specified communication type (usually 0). The optional parameter timeout is a pointer to a maximum waiting time for completion of a connection for type SOCK_STREAM before aborting the operation by raising an abnormal event; this parameter is only applicable for peer connection, SOCK_STREAM, communication. The optional parameter I/O cluster is where the socket for the client is created.

The last 4 constructors of uSocketClient are for use with the INET address family on a non-local host. All parameters are the same as for the local host case, except the non-local host machine-name is specified by the name parameter.

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 I/O members are explained in their corresponding UNIX manual entries, with the following exceptions:

- getpeername is only applicable for peer connection, SOCK_STREAM, communication.
- the first parameter to these UNIX routines is unnecessary, as it is provided implicitly by the uSocketClient object
- the lack of address for the overloaded member routines sendto and recvfrom.

  The client implicitly remembers the address of the initial connection and each recvfrom call. Therefore, no address needs to be specified in the sendto, as the data is sent directly back to the last address received. If a client needs to communicate with multiple servers, explicit addresses can be specified in both sendto and recvfrom.
- the optional parameter timeout, which points to a maximum waiting time for completion of the I/O operation before aborting the operation by raising an abnormal event

Appendix C.6.1 shows a client communicating with a server using a UNIX socket and datagram messages. Appendix C.6.3 shows a client connecting to a server using an INET socket and stream communication with an acceptor.

3.4.2 Server

In μC++, a server, its socket endpoint and a connection to a client are created by declaration of a uSocketServer object, e.g.:

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

which creates a server variable, server, and a UNIX server socket endpoint, abc. The operations provided by uSocketServer are listed in Figure 3.8:

The first 2 constructors of uSocketServer are for use with the UNIX address family. The parameters for the constructors are as follows. The name parameter is the name of a new UNIX server socket that the server is creating. The type and protocol parameters are explained in the UNIX manual entry for socket. Only types SOCK_STREAM and SOCK_DGRAM communication can be specified, and any protocol appropriate for the specified communication type (usually 0). The backlog parameters is explained in the UNIX manual entry for listen; it specifies a limit on the number of incoming connections from clients and is only applicable for peer connection, SOCK_STREAM, communication.

The next 4 constructors of uSocketServer are for use with the INET address family on a local host. The parameters for the constructors are as follows. The port parameter is the port number of an INET port on the local host machine, or a pointer to a location where a free port number, selected by the UNIX system, is placed. The type and protocol parameters are explained in the UNIX manual entry for socket. Only types SOCK_STREAM and SOCK_DGRAM communication can be specified, and any protocol appropriate for the specified communication type (usually 0). The backlog parameters is explained in the UNIX manual entry for listen; it specifies a limit on the number of incoming connections from clients and is only applicable for peer connection, SOCK_STREAM, communication. The optional parameter cluster is where the socket for the server is created.

```
class uSocketServer {
  public:
    // AF_ UNIX
    uSocketServer( const char *name, int type = SOCK_STREAM, int protocol = 0, int backlog = 10 );
    uSocketServer( const char *name, uIOCluster &cluster, int type = SOCK_STREAM, int protocol = 0,
        int backlog = 10 );
    // AF_ INET, local host
    uSocketServer( unsigned short port, int type = SOCK_STREAM, int protocol = 0, int backlog = 10 );
    uSocketServer( unsigned short port, uIOCluster &cluster, int type = SOCK_STREAM, int protocol = 0,
        int backlog = 10 );
    uSocketServer( unsigned short *port, int type = SOCK_STREAM, int protocol = 0, int backlog = 10 );
    uSocketServer( unsigned short *port, uIOCluster &cluster, int type = SOCK_STREAM, int protocol = 0,
        int backlog = 10 );

    int getsockname( sockaddr *name, int *len );
    int getpeername( sockaddr *name, int *len );

    int read( char *buf, int len, uDuration *timeout = NULL );
    int readv( const struct iovec *iov, int iovcnt, uDuration *timeout = NULL );
    int write( char *buf, int len, uDuration *timeout = NULL );
    int writev( const struct iovec *iov, int iovcnt, uDuration *timeout = NULL );
    int send( char *buf, int len, int flags = 0, uDuration *timeout = NULL );
    int sendto( char *buf, int len, int flags = 0, uDuration *timeout = NULL );
    int sendto( char *buf, int len, void *to, int tolen, int flags = 0, uDuration *timeout = NULL );
    int sendmsg( const msghdr *msg, int flags = 0, uDuration *timeout = NULL );
    int recv( char *buf, int len, int flags = 0, uDuration *timeout = NULL );
    int recvfrom( char *buf, int len, int flags = 0, uDuration *timeout = NULL );
    int recvfrom( char *buf, int len, void *from, int *fromlen, int flags = 0, uDuration *timeout = NULL );
    int recvmsg( msghdr *msg, int flags = 0, uDuration *timeout = NULL );

    uDualEvent uFailure;
    uDualEvent uOpenFailure;
    uDualEvent uCloseFailure;
    uDualEvent uReadFailure;
    uDualEvent uReadTimeout;
    uDualEvent uWriteFailure;
    uDualEvent uWriteTimeout;
};
```

Figure 3.8: uSocketServer Interface

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 the I/O members are explained in their corresponding UNIX manual entries, with the following exceptions:

- getpeername is only applicable for peer connection, SOCK_STREAM, communication.
- the first parameter to these UNIX routines is unnecessary, as it is provided implicitly by the uSocketClient object
- the lack of address for the overloaded member routines sendto and recvfrom.

  The server implicitly remembers the address of the initial connection and each recvfrom call. Therefore, no address needs to be specified in the sendto, as the data is sent directly back to the last address received. If a server needs to communicate with multiple clients without responding back immediately to each request, explicit addresses can be specified in both sendto and recvfrom.
- the optional parameter timeout, which points to a maximum waiting time for completion of the I/O operation before aborting the operation by raising an abnormal event (see Section 6.3.2)

```
class uSocketAccept {
  public:
    uSocketAccept( uSocketServer &s, void *adr = NULL, int *len = NULL );
    uSocketAccept( uSocketServer &s, uDuration *timeout, void *adr = NULL, int *len = NULL );

    int getsockname( sockaddr *name, int *len );
    int getpeername( sockaddr *name, int *len );

    int read( char *buf, int len, uDuration *timeout = NULL );
    int readv( const struct iovec *iov, int iovcnt, uDuration *timeout = NULL );
    int write( char *buf, int len, uDuration *timeout = NULL );
    int writev( const struct iovec *iov, int iovcnt, uDuration *timeout = NULL );
    int send( char *buf, int len, int flags = 0, uDuration *timeout = NULL );
    int sendto( char *buf, int len, int flags = 0, uDuration *timeout = NULL );
    int sendto( char *buf, int len, void *to, int tolen, int flags = 0, uDuration *timeout = NULL );
    int sendmsg( const msghdr *msg, int flags = 0, uDuration *timeout = NULL );
    int recv( char *buf, int len, int flags = 0, uDuration *timeout = NULL );
    int recvfrom( char *buf, int len, int flags = 0, uDuration *timeout = NULL );
    int recvfrom( char *buf, int len, void *from, int *fromlen, int flags = 0, uDuration *timeout = NULL );
    int recvmsg( msghdr *msg, int flags = 0, uDuration *timeout = NULL );

    uDualEvent uFailure;
    uDualEvent uOpenFailure;
    uDualEvent uOpenTimeout;
    uDualEvent uCloseFailure;
    uDualEvent uReadFailure;
    uDualEvent uReadTimeout;
    uDualEvent uWriteFailure;
    uDualEvent uWriteTimeout;
};
```

Figure 3.9: uSocketAccept Interface

Appendix C.6.2 shows a server communicating with multiple clients using a UNIX socket and datagram messages. Appendix C.6.4 shows a server communicating with multiple clients using an INET socket and stream communication with an acceptor.

3.4.3 Server Acceptor

After a server socket is created for peer connection communication, it is possible to accept connections from clients by declaration of a uSocketAccept object, e.g.:

```
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 provided by uSocketAccept are listed in Figure 3.9:

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, and are used to determine information about the client the acceptor is connected to. The optional parameter timeout is a pointer to a maximum waiting time for completion of the connection before aborting the operation by raising an abnormal event (see Section 6.3.2).

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 I/O members are explained in their corresponding UNIX manual entries, with the following exceptions:

- the first parameter to these UNIX routines is unnecessary, as it is provided implicitly by the uSocketClient object

- the optional parameter timeout, which points to a maximum waiting time for completion of the I/O operation before aborting the operation by raising an abnormal event (see Section 6.3.2)

□ 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 3.1). Currently, there is no facility to do this. □

3.4.4 Ancillary Routines

Several systems do not have thread safe versions of routines gethostname and gethostbyname. As well, time-slicing during the execution of these routines causes internal system calls (such as opens and reads of network files) to fail. The following routines are thread-safe with time-slicing turned off:

```
extern int uGethostname( char *name, int len );
extern int uGethostbyname( char *name, struct hostent *h );
```

The latter routine copies the structure hostent into argument h, and returns 0 on successful completion and -1 otherwise.

Chapter 4

Abnormal Events and Errors

While C++ has a simple abnormal event handling mechanism (AEHM) based on throwing and catching types, the mechanism does not extend to a concurrent environment. The reason is that C++ exceptions transfer out of blocks (including blocks created by routine calls); these blocks are built on an execution-state's stack. When there is a single execution-state, as for normal C++ execution, transfer of control is obvious: traverse up the blocks on the stack until an appropriate handler is found or the top of the stack is reached. However, in a concurrent system, there are multiple stacks, and hence, transfer of control for propagating an exception becomes more complex, because it may be necessary to transfer or propagate an abnormal event to another execution-state. Such a capability requires additional language semantics and constructs; therefore, the abnormal event mechanism in μC++ is a superset of that in C++. As well, with hindsight, some of the poorer features of C++'s exception mechanism are replaced by better mechanisms.

4.1 AEHM

An **abnormal event** occurs when an operation cannot perform its desired computation (this is similar to Eiffel's notion of failure of a contract [Mey88, p. 395]). Two actions can sensibly be taken when an abnormal event occurs:

1. The operation can fail, thereby causing termination of the expression, statement or block from which the operation was invoked. In this case, control transfers, called throwing, to a location other than after the operation invocation. This results in an **exceptional** change in control flow. To be useful, the location that control transfers to must be dynamically determined, as opposed to statically determined; otherwise, the same action and context for that action is executed for every exceptional change in control flow.

2. The operation can call a corrective routine, called resuming, which either takes some corrective action so that the operation can succeed or determines that a correction is impossible and fails as in case 1. The corrective action is an **intervention** in the normal computation of an operation. Like the previous case, the correction routine has to be dynamically determined for the same reason.

Both kinds of abnormal events are supported in μC++. Thus, there are two possible outcomes of an operation: normal completion possibly returning a value, or failure with a change in control flow.

> □ The case where an operation returns a special value to indicate an abnormal event, i.e., a **return code**, is not an actual failure. For example, the UNIX open operation returns a negative value instead of a file descriptor when a file cannot be opened. A return code does not indicate a failure because execution continues normally. In this case, both the user and the operation definer consider the special value to be a legitimate result that must be treated in an appropriate manner. Unlike an exception, which causes a change in control flow, there is no requirement for a user to deal with the indicated failure after the operation invocation; the program continues as if the operation succeeded. Thus, it is the user's responsibility to test the return code and perform some appropriate action before any further operations that rely on the original operation are performed. Unfortunately, relying on users to check return codes is extremely naive. □

4.2 μC++ AEHM

The following features characterize the μC++ AEHM:

- μC++ supports three different kinds of events: throw, resume and dual events; C++ only supports throw events. Each kind of event has its own event hierarchy. These hierarchies are built by publicly inheriting from another event of the same kind. Abnormal event parameters are encapsulated inside a class object, just like C++ exceptions. μC++ provides a set of predefined events covering abnormal kernel and I/O operations.
- μC++ supports two forms of event propagation: throwing and resuming; C++ only supports throwing propagation. Throw events can only be thrown, resume events can only be resumed, and dual events can be either thrown or resumed. μC++ adopts a propagation mechanism so recursive resuming is eliminated.
- μC++ restricts propagation to event objects; C++ allows any class type to be propagated.
- μC++ supports two kinds of handlers: terminating and resuming; C++ only supports terminating handlers. Unfortunately, resuming handlers are only C++ routines of a particular signature due to the lack of nested routines in C++.
- μC++ supports propagating events from one execution-state to another, called **asynchronous propagation**. The μC++ kernel implicitly polls for asynchronous events when restarting after blocking or after a context switch. Such asynchronous events may interrupt a task's normal control flow. It is also possible to (hierarchically) block asynchronous events when their delivery would be inappropriate or erroneous.

4.3 Abnormal Events

While C++ allows any type to be used as an event, μC++ restricts events to those types defined by three additional kinds of classes: **uDualEvent**, **uThrowEvent** and **uRaiseEvent**. An event type has all the properties of a **class**. The general form of an **event type** is the following:

```
uDualEvent event-name {  // or uThrowEvent or uRaiseEvent
  private:
    ...                  // these members are not visible externally
  protected:
    ...                  // these members are visible to descendants
  public:
    ...                  // these members are visible externally
};
```

As well, every abnormal event must have a public default and copy constructor.

Peculiar to the μC++ implementation of abnormal events is the need to initialize each event, as in:

```
uDualEvent ev { ... }    // define dual event
uInitEvent( ev );        // initialize event
```

The initialization can occur only once in a program, and therefore, it usually appears in the implementation file, i.e., .C file, not in the interface file, e.g., .h file.

4.3.1 Abnormal Event Creation and Destruction

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

```
uDualEvent D {
    ...
};
D d;                     // local creation
uRaise d;
D *dp = new D;           // dynamic creation
uRaise *dp;
delete dp;
uThrow D();              // temporary local creation
```

4.3.2 Inherited Members

Each event type, if not derived from another event type, is implicitly derived from a matching type uAEHM::uDualClass, uAEHM::uThrowClass or uAEHM::uRaiseClass, respectively, as in:

```
uDualEvent event-name : public uAEHM::uDualClass ...
uThrowEvent event-name : public uAEHM::uThrowClass ...
uRaiseEvent event-name : public uAEHM::uRaiseClass ...
```

where the interface for the base classes are as follows:

```
class uAEHM::uDualClass {
    uDualClass( const char *const msg = "" );
    const char *const uMsg() const;
    const uBaseCoroutine &uSrc() const;
    const char *uSrcName() const;
    virtual void uDefaultResume() const;
    virtual void uDefaultTerminate() const;
};
class uAEHM::uThrowClass : private uAEHM::uDualClass {
    uThrowClass( const char *const msg = "" );
    const char *const uMsg() const;
    const uBaseCoroutine &uSrc() const;
    virtual void uDefaultTerminate() const;
};
class uAEHM::uRaiseClass : private uAEHM::uDualClass {
    uRaiseClass( const char *const msg = "" );
    const char *const uMsg() const;
    const uBaseCoroutine &uSrc() const;
    virtual void uDefaultResume() const;
};
```

Only the base class uAEHM::uDualClass is discussed as the other two event types are subsets of it.

The constructor routine uDualClass has the following form:

uDualClass(**const char** *const msg = "") – creates the dual event with specified message, which is printed in an error message if the event is not caught. The message is not copied so its storage must persist for the duration of the event.

The member routine uMsg returns the string message associated with an event. The member routine uSrc returns the coroutine/task that propagated the event. In some cases, the coroutine or task may be deleted when the event is caught so this reference may be undefined. The member routine uSrcName returns the name of coroutine/task that propagated the event. This name is copied from the propagating coroutine/task when the event is created so it is safe to use even if the coroutine/task is deleted.

The member routine uDefaultResume is implicitly called if the event is resumed but not handled; the default action is to throw the event, which begins the search for a terminating handler from the point of the initial resume for dual events, or call uAbort to terminate the program with the supplied message for resume events. The member routine uDefaultTerminate is implicitly called if the event is thrown but not handled; the default action is to call uAbort to terminate the program with the supplied message. In both cases, a user-defined default action is implemented by overriding the appropriate virtual member.

4.4 Event Propagation

There are two propagation mechanisms: throwing and resuming; furthermore, each kind of propagation can be done synchronously or asynchronously. The form of propagation for an event is specified by the propagation statement:

```
uThrow throwable-event;
uThrow throwable-event uAt uBaseCoroutine-id;
uRaise resumable-event;
uRaise resumable-event uAt uBaseCoroutine-id;
```

The kind of event (throw, resume, dual) must match with the kind of propagation statement, i.e., **uThrow** or **uRaise**. (The keyword **uRaise** is used because **uResume** is already associated with coroutine control flow.) The optional **uAt** clause provides for asynchronous propagation of an event to another coroutine or task.

Events in μC++ are propagated differently from C++. In C++, the **throw** statement initializes a temporary object, the type of which is determined from the static type of the operand, and propagates the temporary object. In μC++, the **uThrow** and **uRaise** statements throw the actual operand. For example:

C++	μC++

class fred {};
 	
class mary : **public** fred {};
 	
void x(fred &f) {
  throw f;
}
mary m;
x(m);
	uThrowEvent fred {};
	uInitEvent(fred);
	uThrowEvent mary : **public** fred {};
	uInitEvent(mary);
	void x(fred &f) {
	  uThrow f;
	}
	mary m;
	x(m);

in the C++ program, routine x is passed an object of type mary but throws an object of type fred, because the static type of the operand for throw, f, is of type fred. However, in the μC++ program, routine x is passed an object of type mary and throws the original object of type mary. This change makes a significant difference in the organization of handlers for dealing with abnormal events.

Finally, to rethrow an event inside a terminating handler use:

```
uThrow;
```

which is only valid in the context of a terminating handler. No similar form exists for resuming for implementation reasons (see Section 4.5.2).

4.4.1 Asynchronous Propagation

Asynchronous propagation is when one execution-state propagates an event in another, which provides an additional kind of communication over normal member call. The purpose for asynchronous propagation is to force a communication when an execution-state might otherwise be computing instead of accepting calls. For example, two tasks may begin searching for a key in different sets; the first task to find the key needs to inform the other task to stop searching, as in:

```
uTask searcher {
    searcher &partner;          // other searching task
    void main() {
        for ( ... ) {
            if ( key == ... ) {
                uThrow stopEvent() uAt partner;   // inform partner search is finished
    ...
```

Without asynchronous propagation, both tasks would have to poll for a call from the other task at regular intervals to know if the other task found the key. Asynchronous propagation handles this case and others.

Asynchronous propagation blocks only long enough to deliver the event to the target execution-state and then continues execution. Hence, the communication is asynchronous, whereas member call communication is synchronous. Once the event is delivered to the target execution-state, the runtime system delivers it at the soonest possible opportunity, such as the next context switch of the target.

Some additional features are necessary to fully utilize asynchronous propagation. μC++ allows dynamic enabling and disabling of asynchronous event delivery. An event is blocked if its delivery is disabled. (Note, only asynchronous events can be postponed in this way.) The constructs for controlling asynchronous propagation are the **uEnable** and the **uDisable** blocks, as in:

```
uEnable <AE1><AE2>... {
    // code in uEnable block
}
```

```
uDisable <AE1><AE2>... {
    // code in uDisable block
}
```

The template arguments of the **uEnable** or **uDisable** block specify the events allowed to be delivered or blocked. Specifying no template arguments is shorthand for specifying all abnormal asynchronous events. Though an event being delivered asynchronously may match with more than one event specified in the **uEnable** or **uDisable** block, it is unnecessary to define a precise matching scheme because the event is either enabled or disabled regardless of which event it matches with.

An execution can enter multiple **uEnable** and **uDisable** blocks. Upon entry to a **uEnable** block, the specified events become deliverable, even if the events were previously blocked. Similarly, upon entry to a **uDisable** block, the specified events become blocked, even if the events were previously enabled. Upon exiting a **uEnable** or **uDisable** block, the delivery of the specified asynchronous events are restored to their state prior to entering the block.

Initially, asynchronous delivery of events is turned off in a coroutine or task, so handlers can set up for various events, resulting in the following μC++ idiom for the coroutine or task main:

```
void main() {
    // initialization, no asynchronous delivery
    uEnable { // enable delivery of all events
        // rest of the code for this coroutine or task
    }
    // finalization, no asynchronous delivery
}
```

Several of the predefined kernel events are implicitly enabled in certain contexts to ensure their prompt delivered when propagated asynchronously (see Section 4.8.1).

The μC++ kernel implicitly polls for asynchronous delivery after thread unblocking or a context switch. If this level of polling is insufficient, explicit polling for asynchronous delivery is possible by calling:

```
void uAEHM::uDeliverEvents();
```

Explicit polling is only necessary if pre-emption is disabled, a large number of events are arriving, or timely asynchronous delivery is important.

4.5 Handler

A handler catches a propagated event and attempts to deal with the abnormal situation. Each handler is associated with a particular block of code, called a **guarded block**. μC++ supports two kinds of handlers: terminating and resuming.

> □ Note, when subclassing is used, it is better to catch an event by reference for terminating and resuming handlers. Otherwise, the event is truncated from the actual type to the static type specified at the handler, and cannot be dynamically down-cast to the actual type. Notice, catching event truncation is different from propagation event truncation, which does not occur in μC++ (see Section 4.4). □

An unhandled thrown event for a task causes termination of the program; an unhandled thrown event for a coroutine terminates the coroutine and asynchronously propagates an event in the coroutine's starter (see Section 2.6.1). Finally, an unhandled resuming event is thrown if it is a dual event; otherwise, it is ignored.

4.5.1 Terminating Handler

A terminating handler is a corrective action replacing the action of the guarded block with the handler block. In μC++, a terminating handler is specified identically to that in C++: with the **catch** clause of a **try** statement. (The details of terminating handlers can be found in a C++ textbook.) Figure 4.1 shows how C++ and μC++ use a throw event, throwing propagation and a terminating handler. The differences are using **uThrow** instead of **throw**, throwing the actual type instead of the static type, and requiring a special event type for all events.

> □ In the example, the C++ case is simpler; however, the usefulness of propagating types like **int** in a complex program is highly questionable [Str97, p. 188]. In general, a complex program, and especially library programs, must create unique types to identify specific abnormal events, e.g., underflow, overflow, etc. A propagated **int** could easily be caught by a wrong handler because the type is so general, which subsequently misinterpret the event's value. Therefore, most C++ exceptions ultimately have definitions for throw events very similar to those in μC++. □

C++ Syntax	μC++ Syntax

// possibly an exception type	**uThrowEvent** ex {
	  public:
	  int i;
	  ex(**int** i) : i(i) {}
	};
	uInitEvent(ex);

void f() {
  throw 3;
}
int main() {
  try {
  f();
  } **catch**(**int** i) {
  cout << i << endl;
  throw;
  } *// try*
}
	void f() {
	  uThrow ex(3);
	}
	void uMain::main() {
	  try {
	  f();
	  } **catch**(ex e) {
	  cout << e.i << endl;
	  uThrow;
	  } *// try*
	}

Figure 4.1: C++ versus μC++ Throwing Propagation

4.5.2 Resuming Handler

A resuming handler is a corrective action for a failing operation or it determines correction is impossible and throws an event. Effectively, the corrective action is an intervention in the normal computation when a problem is encountered. Unlike normal routine calls, the call to the resuming handler is dynamically bound rather than statically bound, so that different corrections can occur for the same static context.

Figure 4.2 shows the ideal syntax for specifying resuming handlers on the left, and the compromise syntax provided by μC++ on the right. On the left, the resuming handler is, in effect, a nested routine called when a propagated resume event is caught by the handler; when the resuming handler completes, control returns back to the point of the resume. Values at the resume point can be modified directly in the handler if variables are visible in both contexts, or indirectly through reference or pointer parameters; there is no concept of a return value from a resuming handler, as is possible with a normal routine. Unfortunately, C/C++ has no notion of nested routines, so it is largely impossible to simulate the ideal resuming handler syntax, nor is there any precedence for it.

On the right is the simulation of the ideal resuming handler syntax. The most significant change is the movement of the resuming handler bodies to routines rtn1 and rtn2, respectively. Also, the direct access of local variables x and y in the first resume handler necessitates creating a closure of this context so that rtn2 can access them.

In detail, μC++ extends the **try** block to set up resuming handlers, where the resuming handler is a routine. Any number of resuming handlers can be associated with a **try** block and there are 3 different forms for specifying a resuming handler, as in.

```
try <AE1,h1,arg1><AE2,h2><AE3>... {
    // statements to be guarded
} // possible catch clauses
```

The three forms of resuming handler are:

1. <AE1,h1,arg1>
2. <AE2,h2>
3. <AE3>

The first form specifies routine h1 handles resuming event AE1. arg1 is used for passing local information to handler h1 when handling the event. Therefore, the signature of h1 must be:

```
void h1( AE1 &, ARG1 & )
```

where type ARG1 must be publicly derived from uAEHM::uClosure. The difference between the second form and the first is that the handler h2 for event AE2 does not take additional arguments. Therefore, its signature must be:

Ideal Syntax

```
uRaiseEvent R1 {
  public:
    int &i; char &c;
    R1( int &i, char &c ) : i( i ), c( c ) {}
};
uInitEvent(R1);

uRaiseEvent R2 {};
uInitEvent(R2);

void f( int x, char y ) { uRaise R2; }
void g( int &x, char &y ) { uRaise R1( x, y ); }









void uMain::main() {
    try {
        int x = 0;
        char y = 'a';


        g( x, y );
        try {
            f( x, y );
        } resume( R2 ) {
            x = 2; y = 'c'; // modify local variables
        } // try
        try {
            g( x, y );
        } resume( R1 ) { // ignore R1
        } // try
    } resume( R1 &r ) {
        // cannot see variables x & y
        r.i = 1; r.c = 'b'; // modify arguments
    } // try
}
```

Actual μC++ Syntax

```
uRaiseEvent R1 {
  public:
    int &i; char &c;
    R1( int &i, char &c ) : i( i ), c( c ) {}
};
uInitEvent(R1);

uRaiseEvent R2 {};
uInitEvent(R2);

void f( int x, char y ) { uRaise R2(); }
void g( int &x, char &y ) { uRaise R1( x, y ); }

struct args : public uAEHM::uClosure {
    int &i; char &c;
    args( int &i, char &c ) : i( i ), c( c ) {}
};

void rtn1( R1 &r ) { r.i = 1; r.c = 'b'; }
void rtn2( R2 &r, args &a ) { a.i = 2; a.c = 'c'; }

void uMain::main() {
    try <R1,rtn1> {
        int x = 0;
        char y = 'a';
        args a( x, y ); // closure for locals

        g( x, y );
        try <R2,rtn2,a> {
            f( x, y );


        } // try
        try <R1> { // ignore R1
            g( x, y );

        } // try




    } // try
}
```

Figure 4.2: Syntax for Resuming Handlers

```
void h2( AE2 & )
```

This form is the most common, particularly for resuming handlers in library code. The third form specifies the event AE3 is handled by an empty handler. This form eliminates having to create a handler routine with an empty routine body. Type checking is performed on the first two forms, to ensure a proper handler is chosen to handle the designated abnormal event.

□ The reason there is no re-resuming form for **uRaise**, as there is a rethrowing form for **uThrow**, is because of the μC++ separation of the resuming block from the handler it represents. For example, given a μC++ resuming routine with a re-resume statement:

```
void g( R & ) { uRaise; }
```

it is impossible to know statically which event is to be resumed because the connect to the event is at the **try** statement. As opposed to the ideal form for a resuming handler:

```
try {
    ...
} resume( R ) {
    uRaise;      // event R is implicitly resumed
}
```

where the resuming block is part of the **try** statement. This problem is solely an artifact of the μC++ implementation of resuming handlers caused by lack of nested routines in C++. □

4.5.2.1 Recursive Resuming

Resuming does not unwind the stack. As result, handlers defined in previous scopes continue to be present during resuming propagation. In throwing propagation, the handlers in previous scopes disappear as the stack is unwound. In some languages with a resuming propagation [Mac77, BMZ92, Geh92], the presence of resuming handlers in previous scopes can cause a situation called **recursive resuming**. The simplest situation where recursive resuming can potentially occur is when a handler for a resuming event resumes the same event, as in:

```
uRaiseEvent R {};
uInitEvent(R);

void f( R & ) {
    uRaise R();
}
void uMain::main() {
    try <R,f> {
        uRaise R();
    }
}
```

Routine uMain::main sets up a try block for resuming event R with handler routine f, respectively. Handler f is invoked by the resume, and the blocks on the call stack are:

uMain::main $\rightarrow$ **try**<R,f> $\rightarrow$ f()

Then f resumes event R again, which finds the handler just above it for R, which invokes handler routine f again and this continues until the runtime stack overflows. Recursive resuming is similar to infinite recursion, and is difficult to discover both at compile time and at runtime because of the dynamic choice of a handler. While an event may be resumed synchronously or asynchronously, only synchronously resumed events need to be considered as candidates for causing recursive resuming, because appropriate nested blocks must exist with appropriate handlers to cause the recursion on a particular execution stack.

4.5.2.2 Preventing Recursive Resuming

Recursive resuming is probably the only legitimate criticism against resuming propagation. However, not all abnormal events handled by a resuming handler cause recursive resuming. Even if a resuming handler resumes the event it handles, which guarantees activating the same resuming handler again, (infinite) recursive resuming may not happen because the handler can take a different execution path as a result of a modified execution state. Because the resuming

propagation suggested previously searches for a handler by simply going up the runtime stack one stack frame at a time, it has the recursive resuming problem. μC++ has a modified propagation mechanism, called **linear propagation**, that provides a solution to the recursive resuming problem. Furthermore, the mechanism is extended to cover asynchronous events.

An execution resuming an abnormal event synchronously indicates that an abnormal condition is encountered. A handler can catch an event and then resume another synchronous event when encountering another abnormal condition, resulting in a second synchronous abnormal event. The second event is considered a **consequent event** of the first. More precisely, every synchronous event is an **immediate consequent event** of the most recent abnormal event being handled in the execution (if there is one). For example, in the previous recursive resuming example, the second resume of R in f is a consequence event of the first event R resumed in uMain. (Note, it is unnecessary for the event type to be the same as the consequence event type.) Therefore, a consequent event is either the immediate consequent event of an event, or the immediate consequent event of another consequent event. The consequence relation is transitive, but not reflexive. The only events that are not a consequent event are asynchronous events, and synchronous events that are propagated when no other events are being handled. An asynchronous abnormal event is not considered as a consequent event of other abnormal events propagated in the target execution because the condition resulting in the event is encountered by another execution, and in general, is not related to the target execution. Hence, any synchronous event resumed before a handler activated by the last asynchronous event terminates is a consequent event of the asynchronous event.

The new propagation mechanism goes up the execution stack one level at a time as it normally does to find a handler capable of handling the event being propagated. In addition, all the resuming handlers at the level being "visited" are marked ineligible (denoted by italics) for the event and its consequent events, whether or not a handler is found. The mark is cleared only if the event is handled either by a terminating or resuming handler.

How does this new propagation mechanism make a difference? Given the previous runtime stack:

uMain::main $\rightarrow$ *try<R,f>* $\rightarrow$ f()

the handler <R,f> is marked ineligible when R is caught at the **try** block and f is called. Hence, the event cannot be handled by <R,f>, and the recursion is avoided and the default action occurs for R. In general, the mechanism eliminates recursive resuming because a resuming handler marked for a particular event cannot be invoked to handle its consequent events. Not only that, propagating a synchronously resumed event out of a handler does not invoke a handler bound to a stack frame between the handler and the handler body, similar to a thrown event propagated out of a guarded block because of stack unwinding.

When handling an event, the flow of the execution can enter additional guarded blocks. For example, if the resuming handler block g is augmented to:

```
void f( R & ) {
    try <R,j> {
        uRaise R();
    }
}
```

where j is an additional resuming handler block, the call stack is extended to the following:

uMain::main $\rightarrow$ *try<R,f>* $\rightarrow$ f() $\rightarrow$ **try**<R,j>

When the execution encounters another synchronous (thrown or resumed) abnormal event, the handler of the new guarded block is first examined as it is unmarked.

Linear propagation does not preclude all infinite recursions with respect to propagation, as in:

```
uRaiseEvent R {};
uInitEvent(R);

void f( R & ) {
    try <R,f> {
        uRaise R();
    }
}
void uMain::main() {
    try <R,f> {
        uRaise R();
    }
}
```

The problem here is each call to f creates a new **try** block to handle the next recursion, as in:

uMain::main → *try<R,f>* → f() → **try**<R,f> → ...

As a result, there is always an eligible handler to catch the next event in the recursion. Linear propagation is not supposed to handle this situation as it is considered a programming error with respect to recursion not propagation.

Finally, linear propagation does not affect throwing propagation because once handlers are marked, they are subsequently removed because of stack unwinding. Hence, the application of linear propagation is consistent to both throwing and resuming. As well, if a resuming handler throws an event, a terminating handler for the same event bound to a prior block of the resuming handler is still eligible, as in:

```
uDualEvent R {};
uInitEvent(R);

void f() { uRaise R(); }
void g( R & ) { uThrow R(); }

void uMain::main() {
    try <R,g> {
        f();
    } catch( R ) {
    }
}
```

which results in the following call stack:

uMain::main → *try<R,g>* **catch**(R) → f() → g()

Notice that while the resuming handler for R is marked ineligible, the terminating handler for the same **try** block is still eligible. (Also note the event R had to be changed to a dual event so that it could be both resumed and thrown.)

All handlers are considered unmarked for a propagated asynchronous event because an asynchronous event is not a consequent event. Therefore, the propagation mechanism searches every handler on the runtime stack. Hence, a handler ineligible to handle an event and its consequent events can be chosen to handle a recently arrived asynchronous event, reflecting the lack of consequentiality of the asynchronous event.

In summation, linear propagation is better than other existing propagation mechanisms because:

- it supports throwing and resuming propagation, and the search for a handler occurs in a uniformly defined way,
- it prevents recursive resuming and handles synchronous and asynchronous abnormal events according to a sensible consequence relation among abnormal events,
- the context of a handler closely resembles its guarded block with respect to lexical location; in effect, an event propagated out of a handler is handled as if the event is directly propagated out of its guarded block.

4.5.2.3 Commentary

Of the few languages with resuming handlers, the language Mesa [MMS79] is probably the only one that also solved the recursive resuming problem. The Mesa scheme prevents recursive resuming by not reusing a handler clause bound to a specific invoked block, i.e., once a handler is used as part of handling an event, it is not used again.

The propagation mechanism always starts from the top of the stack to find an unmarked handler for a resume event. However, this unambiguous semantics is often described as confusing and incomprehensible.

The following program demonstrates how μC++ and Mesa solve recursive resuming, but with different solutions:

```
uRaiseEvent R1 {};
uInitEvent(R1);
uRaiseEvent R2 {};
uInitEvent(R2);

void f() { uRaise R1(); }
void g( R2 & ) { uRaise R1(); }
void h( R1 & ) { uRaise R2(); }
void j( R2 & ) {}

void uMain::main() {
    try <R2,j> {
        try <R1,h> {
            try <R2,g> {
                f();
            }
        }
    }
}
```

The following stack frame is generated at the point when resuming handler h is called from f:

uMain::main → **try**<R2,j> → **try**<R1,h> → **try**<R2,g> → f() → h()

The potential infinite recursion occurs because h resumes R2, and there is resuming handler **try**<R2,g>, which resumes R1, while handler **try**<R1,h> is still on the stack. Hence, handler body h invokes handler body g and vice versa with no case to stop the recursion.

Linear propagation prevents the infinite recursion by marking *both* resuming handlers as ineligible before invoking resuming body h, as in:

uMain::main → **try**<R2,j> → *try<R1,h>* → *try<R2,g>* → f() → h()

Therefore, when h resumes R2 the next eligible handler is the one with resume body j. Mesa prevents the infinite recursion by only marking *active* handlers as ineligible, resulting in the following:

uMain::main → **try**<R2,j> → *try<R1,h>* → **try**<R2,g> → f() → h()

Therefore, when h resumes R2 the next eligible handler is the one with resume body g. As a result, R1 is resumed from g and there is no infinite recursion. However, the confusion with the Mesa semantics is that there is no handler for R1, even though the nested **try** blocks appear to properly deal with this situation. In fact, looking at the static structure, a programmer might incorrectly assume there is an infinite recursion between handlers h and g, as they resume one another. This programmer confusion results in a reticence by language designers to incorporate resuming facilities in new languages. In detail, the Mesa semantics has the following negative attributes:

- Resuming an event in a block and in one of its handlers can invoke different handlers, even though the block and its handlers are in the same lexical scope. For instance, in the above example, an event generated in a guarded block is handled by handlers *at* or *above* the block on the stack, but an event generated in a handler body can be handled by handlers *below* it on the stack. Clearly, lexical scoping does not reflect the difference in semantics.

- Abstraction states that a routine should be treated as a "client" of routines it invokes directly or indirectly, and should have no access to the implementation it uses. However, if resuming from a resuming handler is a useful feature, some implementation knowledge about the handlers bound to the stack *below* it must be available to successfully understand how to make corrections. This requirement violates abstraction.

- Finally, abnormal events are designed for communicating abnormal conditions from callee to caller. However, resuming an event inside a resuming handler is like abnormal condition propagating from caller to callee because of the use of handlers *below* it on the stack.

4.6 Inheritance

The following explains the forms of inheritance allowed with abnormal events. First, an abnormal event may only inherit publicly from an event. Public derivation of events is for building an event hierarchy, and the restriction on public inheritance enhances the distinction between class and event hierarchies. The distinction stems from the difference in usage: handlers for events and members for classes. The former is a dynamic choice; the latter static. Second, an abnormal event can inherit non-publicly from an ordinary class object. An example for using such inheritance is a set of abnormal events using similar logging routines. An ordinary class can implement the logging functionalities and consequently be reused by the events.

□ There does not appear to be any circumstances where an event needs to inherit from a coroutine or a task. Why should an abnormal event need context switching or a thread to carry out computation? Furthermore, what should happen when a hybrid of execution and event encounters another abnormal condition? Consequently, any form of inheritance from a coroutine or a task by an abnormal event is rejected.

While the same abnormal condition may be encountered by different executions, the abnormal event is not a shared resource. For example, arithmetic overflow can be encountered by different executions but each arithmetic overflow is independent. Hence, there is no race condition for events, and therefore, non-public inheritance from a mutex object by an abnormal event is rejected. □

As mentioned, events should be kept distinct from non-events objects. Therefore, non-event classes cannot inherit because operations defined for events may cause problems when accessed through a non-event object. These restrictions do not mean events and non-events cannot share code. Rather, shared code must be factored out as an ordinary class and then inherited by events and non-events.

4.7 Programming with Abnormal Events

Like many other programming features, an AEHM aims to make certain programming tasks easier and improve the overall quality of a program. Indeed, choosing abnormal events over other available flow control mechanisms is a tradeoff. For example, a programmer may decide to use abnormal events over some conditional statement for clarity. This decision may sacrifice runtime efficiency and memory space. In other words, universal, crisp criteria for making a decision do not exist. Nevertheless, some important guidelines are given to encourage good use of abnormal events.

First, use abnormal events when writing a library to indicate rare conditions to prevent the library client from ignoring them if the client "forgets" to set up a handler. Hence, abnormal events improve safety and robustness. Second, use abnormal events to improve clarity and maintainability over techniques like status return values and status flags where normal flow and abnormal flow are mixed together within a block. The abnormal event technique not only separates clearly the normal flow in a guarded block from the abnormal flow in handlers, but also avoids mixing normal return values with error return values. This separation makes subsequent changes easier. Third, use abnormal events to indicate conditions that happen rarely at runtime for the following reasons:

- The normal flow of the program should represent what should happen most of the time, allowing programmers to easily understand the common functionality of a code segment. The abnormal flow then represents subtle details to handle rare situations, such as boundary conditions.
- Because the propagation mechanism requires a search for the handler, it is usually expensive. Part of the cost is a result of the dynamic choice of a handler. Furthermore, this dynamic choice can be less understandable than a normal routine call. Hence, there is a potential for high runtime cost with abnormal events and control flow can be more difficult to understand.

4.7.1 Throw Event

Typical use of a throw event is for graceful termination of an operation, coroutine, task or program. A termination is graceful if the termination triggers a sequence of clean up actions in the execution context. Examples of abrupt (or non-graceful) termination include the uAbort routine (abort in C) and the kill -9 command in Unix. Graceful termination is more important in a concurrent environment because one execution can terminate while others continue. The

terminating operation must be given a chance to release any shared resources it has claimed (the clean up action) in order to maintain the integrity of the execution environment. For example, deadlock is potentially a rare condition and a thrown event can force graceful termination of a blocked operation, consequently leading to the release of some shared resources and breaking of the deadlock.

4.7.2 Resume Event

A resume event causes the faulting execution to do additional computation in the form of a resuming handler that it does not do normally. The additional computation may modify the state of the execution, which can be considered as error recovery. Alternatively, it may cause information about the execution to be gathered and saved without effectively modifying the execution state.

4.7.3 Dual Event

A dual event is either thrown or resumed, and hence can be used as a throw event as well as a resume event. In situations without a clear choice of termination or resumption, a dual event can be resumed initially so that resumption is feasible to avoid loss of local information. If no resumption handler can handle the event, the same event can be thrown.

One possibility of such use is in real-time applications. Missing a real-time constraint, say an execution cannot finish before a deadline, is considered an abnormal condition. For some applications, the constraint violation can result in termination. Other applications can modify some internal parameters to make its execution faster by sacrificing the quality of the solution or by acquiring more computing resources so execution can continue. The dual event is ideal for this kind of abnormal condition.

4.8 Predefined Abnormal Events

μC++ provides a number of predefined abnormal events, which are structured into the hierarchy in Figure 4.3. Notice that all predefined abnormal events are dual events so that they can be both resumed or thrown. Also, the predefined events are divided into two major groups: kernel and I/O. The kernel abnormal events are propagated by the μC++ runtime kernel when problems arise using the μC++ concurrency extensions. The I/O abnormal events are propagated by the μC++ I/O library when problems arise using the file system. Only the kernel abnormal events are discussed, as the I/O events are OS specific.

4.8.1 Implicitly Enabled Events

Certain of the predefined kernel events are implicitly enabled in certain contexts to ensure their prompt delivery when propagated asynchronously. The predefined event uBaseCoroutine::uFailure is implicitly enabled and asynchronous events are polled when a coroutine restarts after a **uSuspend** or **uResume**. The predefined event uSerial::uFailure is implicitly enabled and asynchronous events are polled when a task restarts from blocking on entry to a mutex member. This situation also occurs when a task restarts after being accept blocked on a **uAccept** or a **uWait** statement. The predefined event uSerial::uRendezvousFailure is implicitly enabled and asynchronous events are polled when an acceptor task restarts after blocking for a rendezvous to finish.

4.8.2 Breaking a Rendezvous

As mentioned in Section 2.8.2.2, the accept statement forms a rendezvous between the acceptor and the accepted tasks, where a rendezvous is a point in time at which both tasks wait for a section of code to execute before continuing. It can be crucial to correctness that the acceptor know if the accepted task does not complete the rendezvous code, otherwise the acceptor task continues under the incorrect assumption that the rendezvous action has occurred. To this end, an asynchronous event (see Section 4.4.1) is propagated to the acceptor task if the accepted member terminates abnormally. It may also be necessary for a mutex member to know if the acceptor has restarted, and hence, the rendezvous has ended. This situation can happen if the mutex member calls a private member, which may conditionally wait, which ends the rendezvous. The macro uRendezvousAcceptor can be used only inside mutex types to determine if a rendezvous has ended:

```
uAEHM::uDualClass
    uKernelFailure
        uSerial::uFailure
            uSerial::uEntryFailure
            uSerial::uRendezvousFailure
            uCondition::uWaitingFailure
        uBaseCoroutine::uFailure
            uBaseCoroutine::uTerminated
    uIOFailure
        uFile::uFailure
            uFile::uTerminateFailure
            uFile::uStatusFailure
            uFileAccess::uFailure
                uFileAccess::uOpenFailure
                uFileAccess::uCloseFailure
                uFileAccess::uSeekFailure
                uFileAccess::uSyncFailure
                uFileAccess::uReadFailure
                    uFileAccess::uReadTimeout
                uFileAccess::uWriteFailure
                    uFileAccess::uWriteTimeout
        uSocket::uFailure
            uSocket::uOpenFailure
            uSocket::uCloseFailure
            uSocketServer::uFailure
                uSocketServer::uOpenFailure
                uSocketServer::uCloseFailure
                uSocketServer::uReadFailure
                    uSocketServer::uReadTimeout
                uSocketServer::uWriteFailure
                    uSocketServer::uWriteTimeout
            uSocketAccept::uFailure
                uSocketAccept::uOpenFailure
                    uSocketAccept::uOpenTimeout
                uSocketAccept::uCloseFailure
                uSocketAccept::uReadFailure
                    uSocketAccept::uReadTimeout
                uSocketAccept::uWriteFailure
                    uSocketAccept::uWriteTimeout
            uSocketClient::uFailure
                uSocketClient::uOpenFailure
                    uSocketClient::uOpenTimeout
                uSocketClient::uCloseFailure
                uSocketClient::uReadFailure
                    uSocketClient::uReadTimeout
                uSocketClient::uWriteFailure
                    uSocketClient::uWriteTimeout
uAEHM::uThrowClass
    // no predefines
uAEHM::uRaiseClass
    // no predefines
```

Figure 4.3: μC++ Predefined Abnormal Event Hierarchy

```
uBaseCoroutine *uRendezvousAcceptor();
```

It returns NULL if the rendezvous is ended; otherwise it returns the address of the rendezvous partner. In addition, calling uRendezvousAcceptor has the side effect of cancelling the automatic propagation of an asynchronous event to the acceptor. This capability allows a mutex member to terminate with an abnormal event without informing the acceptor.

4.8.3 Messages

The following examples show different error situations, the error message generated and an explanation of the error. While not all error situations are enumerated, the list covers the common errors present in most μC++ programs.

The following program:

```
uTask T1 {
    uCondition c;
  public:
    mem() { uWait c; }
  private:
    void main() {
        uAccept( mem );          // let T2 in so it can wait
        uSignal c;               // put T2 on acceptor/signalled stack
        uAccept( ~T1 );          // uMain is calling the destructor
    }
};
uTask T2 {
    T1 &t1;
    void main() { t1.mem(); }
  public:
    T2( T1 &t1 ) : t1( t1 ) {}
};
void uMain::main() {
    T1 *t1 = new T1;
    T2 *t2 = new T2( *t1 );
    delete t1;                   // delete in same order as creation
    delete t2;
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:7852) Unhandled event uSerial::uEntryFailure : while executing mutex destructor, task 0x17c640 (uMain) found task 0x18d640 (T2) blocked on acceptor/signalled stack. Error occurred while executing task 0x18d640 (T2).
```

because task t2 is allowed to wait on condition variable c in t1.mem, and then task t1 signals condition c, which moves task t2 to the acceptor/signalled stack, and accepts its destructor. As a result, when task uMain attempts to delete task t1, it finds task t2 still blocked on the acceptor/signalled stack. Similarly, the following program:

```
uTask T1 {
  public:
    mem() {}
  private:
    void main() { uAccept( ~T1 ); }
};
uTask T2 {
    T1 &t1;
  public:
    T2( T1 &t1 ) : t1( t1 ) {}
  private:
    void main() { t1.mem(); }
};
void uMain::main() {
    T1 *t1 = new T1;
    T2 *t2 = new T2( *t1 );
    delete t1;
    delete t2;
}
```

generates this error:

> uC++ Runtime error (UNIX pid:11655) Unhandled event uSerial::uEntryFailure : while executing mutex destructor, task 0x13b240 (uMain) found task 0x14c240 (T2) blocked on entry queue. Error occurred while executing task 0x14c240 (T2).

because task t2 happens to block on the call to t1.mem, and then task t1 accepts its destructor. As a result, when task uMain attempts to delete task t1, it finds task t2 still blocked on the entry queue of t1.

The following program:

```
uThrowEvent ev {};
uInitEvent(ev);

uTask T {
    uBaseTask &t;
  public:
    T( uBaseTask &t ) : t( t ) {}
    mem() {
//      uRendezvousAcceptor();
        uThrow ev();
    }
  private:
    void main() {
        uAccept( mem );
    }
};
void uMain::main() {
    T t( uThisTask() );
    try {
        t.mem();
    } catch( ev &evt ) {
    }
}
```

generates this error:

> uC++ Runtime error (UNIX pid:10281) Unhandled event uSerial::uRendezvousFailure : accepted call fails from task 0xe6b88 (uMain) to mutex member of task 0xee7f0 (T). Error occurred while executing task 0xee7f0 (T).

because in the call to t.mem from task uMain, the rendezvous terminates abnormally by thrown event ev. As a result, uMain implicitly resumes event uSerial::uRendezvousFailure asynchronously at task t so it knows the call did not complete and can take appropriate corrective action (see Section 4.8.2). If the call uRendezvousAcceptor() is uncommented, event uSerial::uRendezvousFailure is not raised at task t, and task t restarts as if the rendezvous completed. A most complex version of this situation occurs when a blocked call is aborted, i.e., before the call even begins. The following program:

```
uThrowEvent ev {};
uInitEvent(ev);

uTask T {
    uBaseTask &t;
  public:
    T( uBaseTask &t ) : t( t ) {}
    mem() {}
  private:
    void main() {
        uThrow ev() uAt t;
        uAccept( mem );
    }
};
void uMain::main() {
    T t( uThisTask() );
    uEnable {
        try {
            t.mem();
        } catch( ev &evt ) {
        }
    }
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:12960) Unhandled event uSerial::uRendezvousFailure : accepted call fails
from task 0x13b488 (uMain) to mutex member of task 0x1431a0 (T). Error occurred while executing task
0x1431a0 (T).
```

because the blocked call to t.mem from task uMain is interrupted by asynchronous event ev. When the blocked call from uMain is accepted, uMain immediately detects the asynchronous event and does not start the call. As a result, uMain implicitly resumes event uSerial::uRendezvousFailure asynchronously at task t so it knows the call did not occur and can take appropriate corrective action (see Section 4.8.2).

The following program:

```
uTask T1 {
    uCondition c;
  public:
    mem() { uWait c; }
  private:
    void main() { uAccept( mem ); }
};
uTask T2 {
    T1 &t1;
    void main() { t1.mem(); }
  public:
    T2( T1 &t1 ) : t1( t1 ) {}
};
void uMain::main() {
    T1 *t1 = new T1;
    T2 *t2 = new T2( *t1 );
    delete t1;
    delete t2;
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:23979) Unhandled event uCondition::uWaitingFailure : task 0x13b370
(uMain) found blocked task 0x14c370 (T2) on condition variable 0x144174. Error occurred while executing
task 0x14c370 (T2).
```

because the call to t1.mem blocks task t2 on condition queue c and then task t1 implicitly accepts its destructor when its main terminates. As a result, when task uMain attempts to delete task t1, it finds task t2 still blocked on the condition queue.

The following program:

```
uCoroutine C {
    void main() {}
  public:
    mem() { uResume; }
};
void uMain::main() {
    C c;
    c.mem();                    // first call works
    c.mem();                    // second call fails
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:8302) Unhandled event uBaseCoroutine::uTerminated : coroutine 0x17bd60
(uMain) context switching to terminated coroutine 0x184000 (C). Error occurred while executing task
0x17bd60 (uMain).
```

because the first call to c.mem resumes coroutine c and then coroutine c terminates. As a result, when uMain attempts the second call to c.mem, it finds coroutine c terminated. A similar situation can be constructed using **uSuspend**, but is significantly more complex to generate, and hence, is not discussed in detail.

The following program:

```
uThrowEvent ev {};
uInitEvent(ev);

uCoroutine C {
    void main() { uThrow ev(); }
  public:
    mem() { uResume; }
};
void uMain::main() {
    C c;
    c.mem();                        // first call fails
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:24996) Unhandled event uBaseCoroutine::uFailure : Unhandled thrown
event propagated to starter coroutine 0x13b118 (uMain) from coroutine 0x1433b8 (C). Error occurred
while executing task 0x13b118 (uMain).
```

because the call to c.mem resumes coroutine c and then coroutine c throws an event it does not handle. As a result, when the top of c's stack is reached, event uBaseCoroutine::uFailure is raised at uMain, since it started c.

4.9 Static (Compile-time) Errors

Static errors are generated by the μC++ translator not by the underlying C++ compiler. These errors are specific to usage problems with the μC++ concurrency extensions.

4.9.1 Messages

The following examples show different error situations, the error message generated and an explanation of the error. While not all error situations are enumerated, the list covers the common errors present in most μC++ programs.

The following program:

```
uTask fred {
    void main() {
        uAccept( mem );
    }
  public:
    mem() {}
};
```

generates this error:

```
test.cc:3: uC++ translator error: accept on a non-mutex member, possibly caused by uAccept appearing
before mutex member definition
```

because the accept of member mem appears *before* the definition of member mem, and hence, the μC++ translator encounters the identifier mem before it knows it is a mutex member. C++ requires definition before use for all identifiers.

The following program:

```
uTask fred {
  public:
    mem() {}
  private:
    void main() {
        uAccept( mem );
        uOr uAccept( mem );
    }
};
```

generates this error:

```
test.cc:7: uC++ translator error: multiple accepts of mutex member "mem" in accept statement
```

because the accept statement specifies the same member, mem, twice. The second specification is superfluous.

The following program:

```
uTask fred {};
typedef fred f;
uTask mary : public f {}
```

generates this error:

```
test.cc:3: uC++ translator error: unsupported use of typedef "f" as base type
```

because of a restriction in the implementation of the μC++ translator (see Section 2.16).

The following program:

```
uMutex class fred {};
uTask mary : fred {};
```

generates this error:

```
test.cc:2: uC++ translator error: non-public inheritance disallowed between the base kind for "fred" and the
derived kind for "mary"
```

because inheritance between a monitor type and a task type is not allowed in μC++ (see Section 2.13). Similar restrictions exist for public and multiple inheritance.

The following program:

```
uMutex class fred {};
uMutex class mary {};
uMutex class jack : public fred, public mary {};
```

generates this error:

```
test.cc:3: uC++ translator error: can inherit from at most one mutex class
```

because only one base type can be a mutex type.

The following program:

```
uTask fred;
uCoroutine fred {};
```

generates this error:

```
test.cc:2: uC++ translator error: "fred" redeclared with different kind
```

because the kind of type for the prototype, **uTask**, does not match the kind of type for the definition, **uCoroutine**.

The following program:

```
uTask fred {
  public:
    uNoMutex mem();
};
uMutex fred::mem() {}
```

generates this error:

```
test.cc:5: uC++ translator error: uMutex attribute of "fred::mem" conflicts with previously declared uNoMu-
tex attribute
```

because the kind of mutual exclusion for the prototype, **uNoMutex**, does not match the kind of mutual exclusion in the definition, **uMutex**.

The following program:

```
uTask fred {
  public:
    uMutex fred() {}
    uMutex operator new() {}
    uMutex operator delete() {}
    uNoMutex ~fred() {}
};
```

generates these errors:

```
test.cc:3: uC++ translator error: constructor must be no-mutex, uMutex attribute ignored test.cc:4: uC++
translator error: "new" operator cannot be mutex, uMutex attribute ignored test.cc:5: uC++ translator error:
"delete" operator cannot be mutex, uMutex attribute ignored test.cc:7: uC++ translator error: destructor of
mutex type must be mutex, uNoMutex attribute ignored
```

because certain members may or may not have the mutex property. A constructor must be no-mutex because it is superfluous to make it mutex when it is called only once during a declaration. Operators **new** and **delete** must be no-mutex because it is superfluous to make them mutex when the constructor and destructor already ensure the correct form of mutual exclusion. Finally, a destructor must be mutex if it is a member of a mutex type because deletion requires mutual exclusion.

The following program:

```
uTask /* no name */ {};
```

generates this error:

```
test.cc:1: uC++ translator error: cannot create anonymous coroutine or task because of the need for
named constructors and destructors
```

because a type without a name cannot have constructors or destructors since both are named after the type, and the μC++ translator needs to generate constructors and destructors if not present for certain kinds of types.

4.10 Dynamic (Run-time) Errors

4.10.1 Assertions

Assertions define runtime checks that must be true or the basic algorithm is incorrect; if the assertion is false, a message is printed and the program is aborted. Assertions are written using the macro uAssert:

```
uAssert( boolean-expression );
```

This macro is the same as the C macro assert, except it works correctly in both unikernel and multikernel. Asserts can be turned off by defining the preprocessor variable NDEBUG before including uAssert.h.

To use assertions in a μC++ program, include the file:

```
#include <uAssert.h>
```

4.10.2 Termination

To terminate a program, print an error message, and generate a core file for subsequent debugging in μC++ use the free routine uAbort.

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

format is a string containing text to be printed and printf style format codes describing how to print the following variable number of arguments. The number of elements in the variable argument list must match with the number of format codes, as for printf. This routine is the same as the UNIX routine abort, except it works correctly in both unikernel and multikernel. The routine uAbort prints a user specified string, which is normally 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.

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 it works correctly in both unikernel and multikernel. Note, global destructors are *not* invoked when exiting because there may be outstanding coroutines and tasks. Therefore, this technique for terminating a μC++ program is not recommended and should be used only in error situations where uAbort is inappropriate (e.g., no error message or core dump).

- uExit should not be used to end uMain::main to pass back a successful return code to the shell, as in:

```
void uMain::main() {
    ...
    uExit( 0 );
}
```

Use the variable uRetCode from uMain::main instead (see Section 2.2). The reason is that uExit terminates the program, and hence, all running threads. If task variables are declared in the global scope or local scope of uMain::main, those threads are terminated at some arbitrary point during their execution, which is usually not the desired behaviour for successful execution. □

4.10.3 Messages

The following examples show different error situations, the error message generated and an explanation of the error. While not all error situations are enumerated, the list covers the common errors present in most μC++ programs. Finally, most of these errors are generated only when using the -debug compilation flag (see Section 2.5.1).

4.10.3.1 Coroutine

The suspend statement resumes the last resumer, and therefore, there must be a resume before a suspend can execute (see Section 2.6.3). The following program:

```
uCoroutine fred {
    void main() {}
  public:
    void mem() {
        uSuspend; // suspend before any resume
    }
};
void uMain::main() {
    fred f;
    f.mem();
}
```

generates this error:

uC++ Runtime error (UNIX pid:6362) : attempt to suspend coroutine 0x178390 (fred), which has never been resumed. Possible cause is a uSuspend statement executed in a member called by a coroutine user rather than by the coroutine main. Error occurred while executing task 0x170020 (uMain).

because the call to fred::mem executes a suspend statement before the coroutine's main member has been started, and hence, there is no resumer to reactivate. In general, the suspend statement is used only within the coroutine main or non-public members called directly or indirectly from the coroutine main, not in public members called by other coroutines.

When a coroutine (or task) is created, there must be sufficient memory to allocate its execution-state. The following program:

```
int uMainStackSize() {
    return 1000000000; // very large stack size for uMain
}
void uMain::main() {
}
```

generates this error:

uC++ Runtime error (UNIX pid:19645) attempt to allocate 1000000000 bytes of storage for coroutine or task execution-state but insufficient memory available. Error occurred while executing task 0x145c08 (uBootTask).

because the declaration of uMain by the uBootTask fails due to the request for a 1000000000 byte stack for uMain.

As mentioned in Section 2.4, the μC++ kernel provides no support for automatic growth of stack space for coroutines and tasks. Several checks are made to mitigate problems resulting from lack of dynamic stack growth. The following program:

```
void uMain::main() {
    char x[uThisCluster().uGetStackSize()];      // array larger than stack space
    uVerify();
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:21473) uVerify(): stack overflow detected: stack pointer 0x175c48 below
limit 0x176030. Possible cause is allocation of large stack frame(s) and/or deep call stack. Error occurred
while executing task 0x175830 (uMain).
```

because the declaration of the array in uMain uses more than the current stack space.

The following program:

```
void uMain::main() {
    {
        char x[uThisCluster().uGetStackSize()];
        for ( int i = 0; i < uThisCluster().uGetStackSize(); i += 1 ) {
            x[i] = 'a';          // write outside stack space
        }
    } // delete array
    uVerify();
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:23846) uVerify(): stack corruption detected. Possible cause is corrupted
stack frame via overwriting memory. Error occurred while executing task 0x175790 (uMain).
```

because the declaration of the array in uMain uses more than the current stack space, and by writing into the array, the current stack space is corrupted (and possibly another stack, as well).

4.10.3.2 Task

One task cannot yield another task; a task can only yield itself (see Section 2.11.2). The following program:

```
uTask fred {
    void main() {}
};
void uMain::main() {
    fred f;
    f.uYield();          // yielding another task
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:15066) : attempt to yield the execution of task 0x184890 (fred). A task can
only yield itself. Error occurred while executing task 0x17c9f0 (uMain).
```

One task cannot migrate another task; a task can only migrate itself for the same reason as for yielding (see Section 2.11.2). The following program:

```
uTask fred {
    void main() {}
};
void uMain::main() {
    fred f;
    f.uMigrate( uThisCluster() ); // migrating another task
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:6386) : attempt to migrate task 0x1780a0 (fred) to cluster 0x16ef58 (uUser-
Cluster). A task can only migrate itself to another cluster. Error occurred while executing task 0x1700f8
(uMain).
```

The destructor of a task cannot execute if the thread of that task has not finished (halted) because the destructor destroys the environment in which the task's thread is executing. The following program:

```
uTask fred {
    uCondition c;
    void main() {
        uAccept( ~fred );      // uMain invokes destructor
        uWait c;               // fred continues but blocks, which restarts uMain
    }
};
void uMain::main() {
    fred f;
} // implicitly invoke fred::~fred
```

generates this error:

```
uC++ Runtime error (UNIX pid:6399) : attempt to delete task 0x178068 (fred) that is not halted. Possible cause is that the task is blocked on a condition queue. Error occurred while executing task 0x170250 (uMain).
```

because the call to the destructor restarts the accept statement (see Section 2.8.2.3), and the thread of f blocks on condition c, which restarts the destructor. However, the destructor cannot clean up without invalidating any subsequent execution of task f.

4.10.3.3 Condition Variable

Only the owner of a condition variable can wait and signal on it (see Section 2.8.3.1). The following program:

```
uTask fred {
    uCondition &c;
    void main() {
        uWait c;
    }
  public:
    fred( uCondition &c ) : c( c ) {}
};
void uMain::main() {
    uCondition c;
    fred f( c );
    uWait c;
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:6427) : attempt to wait on a condition variable not owned by this mutex object. Possible cause is passing the condition variable outside of the mutex object in which it is created but after it has been used. Error occurred while executing task 0x177f78 (fred).
```

because the condition variable c is passed from uMain to f, and then there is a race to wait on the condition. The error message shows that uMain waited first, and hence, became the condition owner, and then f's attempt to wait fails. Changing **uWait** in fred::main to **uSignal** generates a similar message with respect to signalling a condition not owned by mutex object f. It is possible for one mutex object to create a condition and pass it to another, as long as the creator does not wait on it before passing it.

It is also the case that a signal or wait can only occur in a mutex member, since the signal and wait statement access internal data structures of the mutex object (e.g., condition variable, entry queue and acceptor/signalled stack). Therefore, all operations on these data structures must be performed with mutual exclusion. The following program:

```
uTask fred {
    uCondition c;
    void main() { uWait c; }
  public:
    uNoMutex void mem() {
        uSignal c;
    }
};
void uMain::main() {
    fred f;
    uYield();
    f.mem();
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:18508) : attempt to signal a condition variable in a mutex object not locked by this task. Possible cause is attempting to signal a condition variable in a no mutex member routine. Error occurred while executing task 0x170200 (uMain).
```

Changing **uSignal** in fred::mem to **uWait** generates a similar message with respect to waiting on a condition not locked by mutex object f.

A condition variable must be non-empty before examining data stored with the front task blocked on the queue (see Section 2.8.3.1). The following program:

```
void uMain::main() {
    uCondition c;
    int i = c.uFront();
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:20313) : attempt to access data with uFront on an empty condition. Possible cause is not checking if the condition is empty before attempting to read stored data. Error occurred while executing task 0xc0738 (uMain).
```

because the condition variable c is empty, there is no data to return.

4.10.3.4 Locks

The argument for the uLock constructor (see Section 2.14.3) must be 0 or 1. The following program:

```
void uMain::main() {
    uLock l(3);
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:18523) : attempt to initialize uLock 0x178008 to 3, which exceeds range 0-1. Error occurred while executing task 0x16fc20 (uMain).
```

because the value 3 passed to the constructor of uLock is outside the range 0-1.

4.10.3.5 Calendar

When creating an absolute time value using uTime (see Section 6.1), the value must be in the range 00:00:00 UTC, January 1, 1970 to 03:14:07 UTC, January 19, 2038, which is the UNIX start and end epochs. The following program:

```
void uMain::main() {
    uTime t( -17 );
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:18249) : attempt to create uTime( year=1970, month=0, day=0, hour=0, min=0, sec=-17, nsec=0 ), which exceeds range 00:00:00 UTC, January 1, 1970 to 03:14:07 UTC, January 19, 2038. Error occurred while executing task 0x17a670 (uMain).
```

4.10.3.6 Cluster

A cluster cannot be deleted with a task still on it, regardless of what state the task is in (i.e., blocked, ready or running). The following program:

```
uTask fred {
    void main() {}
};
void uMain::main() {
    fred &f = *new fred;
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:18545) : attempt to delete cluster 0x16eef8 (uUserCluster) with task
0x178898 (fred) still on it. Possible cause is that the task has not been deleted. Error occurred while
executing task 0x14abf8 (uBootTask).
```

because the uBootTask happens to delete the user cluster (see Section 5.3) after uMain::main terminates before the dynamically allocated task f has terminated. Deleting the task associated with f *before* uMain::main terminates solves the problem.

Similarly, a cluster cannot be deleted with a processor still located on it, regardless of what state the processor is in (i.e., running or idle). The following program:

```
void uMain::main() {
    uProcessor &p = *new uProcessor( uThisCluster() );
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:18556) : attempt to delete cluster 0x16eac0 (uUserCluster) with task
0x1784e0 (uProcessorTask) still on it. Possible cause is that the task has not been deleted. Error oc-
curred while executing task 0x14a840 (uBootTask).
```

because the uBootTask deletes the user cluster (see Section 5.3) after uMain::main terminates but the dynamically allocated processor p is still on the user cluster. Deleting the processor associated with p *before* uMain::main terminates solves the problem.

An I/O cluster can have only one processor because of the way some UNIX systems share file descriptors (see Section 3.1.2). The following program:

```
void uMain::main() {
    uIOCluster c;
    uProcessor p( c );      // adding another processor to an I/O cluster
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:18567) : attempt to add or move processor 0x1780b0 to or from I/O cluster
0x178100 (uIOCluster). An I/O cluster (uIOCluster) can have only one processor. Error occurred while
executing task 0x16fdb8 (uMain).
```

because the declaration of new processor p is on I/O cluster c, which already has a processor and can have only one processor.

4.10.3.7 Heap

μC++ provides its own concurrent dynamic memory allocation routines. Unlike most C/C++ dynamic memory allocation routines, μC++ does extra checking to ensure that some aspects of dynamic memory usage are done correctly. The following program:

```
void uMain::main() {
    int *p = (int *)1;      // invalid pointer address
    delete p;
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:18677) : attempt to free storage 0x1 outside the current heap range:0x15c438
to 0x178418. Possible cause is corrupted pointer. Error occurred while executing task 0x16fc20 (uMain).
```

because the value of pointer p is not within the heap storage area, and therefore, cannot be deleted.

The following program:

```
void uMain::main() {
    int *p = new int[10];
    delete &p[5];           // not the start of the array
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:18688) : attempt to free storage 0x178444 with corrupted header. Possible
cause is duplicate free on same block or overwriting of header information. Error occurred while executing
task 0x16fc30 (uMain).
```

because the pointer passed to **delete** must always be the same as the pointer returned from **new**. In this case, the value passed to **delete** is in the middle of the array instead of the start.

As well, a warning message is issued at the end of a program if all storage is not freed.

```
uC++ Runtime warning (UNIX pid:3914) : program terminating with 32(0x20) bytes of storage allocated
but not freed. Possible cause is unfreed storage allocated by the program or system/library routines called
from the program.
```

While this message indicates unfreed storage, it does not imply this storage was allocated by the user's code. Many system and library operations, such as I/O, allocate storage (such as buffers) that is not freed before program termination. There is nothing that can be done about this unfreed storage. Therefore, the value printed is only a guide in determining if all of a user's storage is freed.

4.10.3.8 I/O

There are many different I/O errors; only those related to the μC++ kernel are discussed. The following program:

```
void uMain::main() {
    uIOCluster c;
    c.uSelect( -1, 0, NULL );
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:18799) : attempt to select on file descriptor -1, which exceeds range 0-1023.
Error occurred while executing task 0x16fd60 (uMain).
```

The following program:

```
void uMain::main() {
    uIOCluster c;
    c.uSelect( -1, NULL, NULL, NULL, NULL );
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:18810) : attempt to select with a file descriptor set size of -1, which exceeds
range 0-1024. Error occurred while executing task 0x16fd68 (uMain).
```

4.10.3.9 Processor

The following program:

```
void uMain::main() {
    uSemaphore s(0);
    s.uP();             // block only thread => synchronization deadlock
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:19217) : no ready or pending tasks. Possible cause is tasks in a synchro-
nization or mutual exclusion deadlock. Error occurred while executing task 0x15cf30 (uProcessorTask).
```

4.10.3.10 UNIX

There are many UNIX related errors, of which only a small subset are handled specially by μC++.

A common error in C++ programs is to generate and use an invalid pointer. This situation can arise because of an incorrect pointer calculation, such as an invalid subscript. The following program:

```
void uMain::main() {
    int *p = NULL;    // set address to 0
    *p += 1;          // writing to protected memory
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:20061) attempt to address location 0x0. Possible cause is reading outside
the address space or writing to a protected area within the address space with an invalid pointer. Error
occurred while executing task 0x16fc60 (uMain).
```

because the value of pointer p is probably within the executable code, which is read only, but an attempt to write is occurring.

If a μC++ program is looping for some reason, it may be necessary to terminate its execution. Termination is accomplished using a shell kill command, sending signal SIGTERM to the UNIX process. μC++ receives the termination signal and attempts to shutdown the application, which is important in multikernel mode with multiple processors. The following program:

```
void uMain::main() {
    kill( getpid(), SIGTERM );        // send SIGTERM signal to program
}
```

generates this error:

```
uC++ Runtime error (UNIX pid:18517) application interrupted by a termination signal. Error occurred while
executing task 0x175628 (uMain).
```

because the μC++ program sent itself a termination (SIGTERM) signal.

Chapter 5

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. Some 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 and the scheduling of tasks on virtual processors is performed by the μC++ kernel.

5.1 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, which 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, e.g., POSIX style cover routines. In general, this is not practical for users as too many cover routines have to be created; many operating system vendors are attempting to make their libraries "thread-safe" and/or provide POSIX style cover routines..

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. μC++ adopts this particular approach because it cannot rely on all operating system vendors having thread-safe libraries.

Determining whether an address is in user code is done by relying on the placement of programs in memory in a particular order by the loader. 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. The loader then loads all object code in the order specified in the compile command, which 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 situation 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 case is

handled 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.

5.2 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++, as are the C routines malloc and free. These memory management operations provide identical functionality to the C++ and C equivalent ones. Furthermore, the μC++ versions of these memory management operators provide mutual exclusion on calls to them.

5.3 Cluster

As mentioned in Section 2.3.1, a cluster is a collection of μC++ tasks and processors; it provides a runtime environment for execution. This environment controls the amount of parallelism and contains variables to affect how coroutines and tasks behave on a cluster. Environment variables are used implicitly, unless overridden, when creating execution-state on a cluster:

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

The variable(s) is either explicitly set or implicitly assigned a μC++-dependent default value when the cluster is created. A cluster is used in operations like task or processor creation to specify the cluster on which the task or processor is associated. After a cluster is created, it the user's responsibility to associate at least one processor with the cluster so it can execute tasks.

The cluster interface is the following:

```
class uCluster {
  public:
    uCluster( unsigned int stacksize = uDefaultStackSize(), const char *name = "" );
    uCluster( const char *name );
    uCluster( uBaseSchedule<uBaseTaskDL> &ReadyQueue,
        unsigned int stacksize = uDefaultStackSize(), const char *name = "" );
    uCluster( uBaseSchedule<uBaseTaskDL> &ReadyQueue, const char *name = "" );

    const char *uSetName( const char *name );
    const char *uGetName() const;
    unsigned int uSetStackSize( unsigned int stacksize );
    unsigned int uGetStackSize() const;
};

uCluster clus( 8196, "fred" )  // 8K default stack size, cluster name is "fred"
```

The overloaded constructor routine uCluster has the following forms:

uCluster(**unsigned int** stacksize = uDefaultStackSize(), **const char** *name = "") – this form uses the user specified stack size and cluster name (see Section 5.5 for the first default value).

uCluster(**const char** *name) – this form uses the user specified name for the cluster and the current cluster's default stack size.

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.

The member routine uSetName associates a name with a cluster and returns the previous name. The member routine uGetName returns the string name associated with a cluster.

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

The 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.

The free routine:

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

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

5.4 Processors

As mentioned in Section 2.3.2, a μC++ virtual processor is a "software processor"; it provides a runtime environment for parallel thread execution. This environment contains variables to affect how thread execution is performed on a processor. Environment variables are used implicitly, unless overridden, when executing threads on a processor:

pre-emption time is the default time, in milliseconds, to the next implicit yield of the currently executing task to simulate non-deterministic execution (see Section 5.4.1).

spin time is the default number of iterations a processor spins before becoming idle (see Section 5.4.2).

The variable(s) is either explicitly set or implicitly assigned a μC++-dependent default value when the processor is created.

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

As stated previously, there are two versions of the μC++ kernel: the unikernel, which is designed to use a single processor; and the multikernel, which is designed to use several processors. The interfaces to the unikernel and multikernel are identical; the only difference is that the unikernel has only one virtual processor. In particular, in the unikernel, operations to increase or decrease the number of virtual processors are ignored. 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.

The processor interface is the following:

```
class uProcessor {
  public:
    uProcessor( int ms = uDefaultPreemption(), int spin = uDefaultSpin() );
    uProcessor( uCluster &uClus, int ms = uDefaultPreemption(), int spin = uDefaultSpin() );

    uClock &uGetClock() const;
    int uGetPid() const;
    uCluster &uGetCluster() const;
    uCluster &uSetCluster( uCluster &uClus );
    uBaseTask &uGetTask() const;
    int uSetPreemption( int ms );
    int uGetPreemption() const;
    int uSetSpin( int spin );
    int uGetSpin() const;
    bool uIdle() const;
    void *uMmap( void *begin, int size, int prot, int flags, int fd, int off );
};

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

The overloaded constructor routine uProcessor has the following forms:

uProcessor(**int** ms = uDefaultPreemption(), **int** spin = uDefaultSpin()) – creates the processor on the current cluster with the user specified time-slice and processor-spin duration (see Section 5.5 for the default values).

uProcessor(uCluster &uClus, **int** ms = uDefaultPreemption(), **int** spin = uDefaultSpin()) – creates a processor on the specified cluster using the user specified time-slice and processor-spin duration.

The member routine uGetClock) returns the clock used to control timing on this processor (see Section 6.4).

The member routine uGetPid returns the current UNIX process id that the processor is associated with.

The member routine uGetCluster returns the current cluster the processor is associated with, and hence, executing tasks for. The member routine uSetCluster moves a processor from its current cluster to another cluster and returns the current cluster.

The member routine uGetTask returns the current task that the processor is executing.

The member routine uSetPreemption is used to set the default pre-emption duration for a processor (see Section 5.4.1) and returns the previous default pre-emption duration. The time duration between interrupts is specified in milliseconds. For example, the call proc.uSetPreemption(50) sets the default pre-emption time to 0.05 seconds for a processor. To turn pre-emption off, call proc.uSetPreemption(0). The member routine uGetPreemption is used to read the current default pre-emption time for a processor. For example, the statement i = proc.uGetPreemption() sets i to the value 50.

The member routine uSetSpin is used to set the default spin-duration for a processor (see Section 5.4.2) and returns the previous default spin-duration. The spin duration is specified as the number of times the cluster's ready queue is checked for an available task to execute before the processor sleeps. For example, the call proc.uSetSpin(500) sets the default spin-duration to 500 checks for a processor. To turn spinning off, call proc.uSetSpin(0). The member routine uGetSpin is used to read the current default spin-duration for a processor. For example, the statement i = proc.uGetSpin() sets i to the value 500.

The member routine uIdle indicates if this processor is currently idle, i.e., the UNIX process has been put to sleep because there were no tasks to execute on the cluster it is associated with.

The routine uMmap allows the virtual memory for the underlying UNIX process to be mapped. This facility is used by persistent storage facilities.

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 does 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 underlying kernel threads or operating system processes (see Section 5.4.3) 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 occurs at the operating system level. Finally, a μC++ program usually shares the hardware processors with other user programs. Therefore, the overall operating system load affects how many processors should be allocated to avoid unnecessary context switching at the operating system level.

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

5.4.1 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 pre-emptive scheduling occurs and another task is scheduled. A task is pre-empted at a non-deterministic location in its execution when the processor's pre-emptive scheduling occurs. Processors on a cluster may have different pre-emption times. 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 might be dependent on order or speed of execution of tasks.

□ On many systems the minimum pre-emption time 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. Furthermore, because interrupts involve entering the UNIX kernel, they are relatively expensive if they occur frequently. An interrupt interval of 0.05 to 0.1 seconds gives adequate concurrency and increases execution cost by less than 1% for most programs. □

5.4.2 Idle Virtual Processors

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

5.4.3 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 is examined.

There are two forms of blocking that can occur:

heavy blocking which is done by the operating system 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 with heavy blocking is that it 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, which is accomplished by keeping the number of tasks that block heavily to a minimum and 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 segregation 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.

□ μC++ tasks are not implemented with kernel threads or operating system processes for two reasons. First, kernel threads have a high runtime cost for creation and context switching. Second, an operating system process is normally allocated as a separate address space (or perhaps several) and if the system does not allow memory sharing among address spaces, tasks 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 operating system, page table creation, and management of the address space of each process. Therefore, kernel threads and 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 kernel threads only indirectly through virtual processors (see Section 2.3.2). A user is not prohibited from creating kernel threads or processes explicitly, but such threads are not administrated by the μC++ runtime environment. □

5.5 Default Values

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

```
int uDefaultStackSize();          // default cluster coroutine/task stack size (bytes)
int uMainStackSize();             // default uMain task stack size (bytes)
int uDefaultPreemption();         // default processor scheduling pre-emption durations (milliseconds)
int uDefaultSpin();               // default processor spin time for idle task (context switches)
```

which are called by a cluster or processor or task 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 or processors created at different times are initialized with different default values (unless overridden on the declaration). However, for stack size, 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, unless the user supplies their own uMainStackSize routine.

Chapter 6

Real-Time

Real-time programming is defined by the correctness of a program depending on both the accuracy of the result *and* when the result is produced. The latter criterion is not present in normal programming. Without programming language facilities to specify time constraints, real-time programs are usually built in ad-hoc ways (e.g., cyclic executive), and the likelihood of encountering timing errors increases through manual calculations. The introduction of real-time constructs is a necessity for accurately expressing time behaviour, as well as providing a means for the runtime system to evaluate whether any timing constraints have been broken. Furthermore, explicit time-constraint constructs can drastically minimize coding complexity as well as analysis. Various programming language constructs for real-time environments are discussed in [SD92, Mar78, LN88, KS86, KK91, ITM90, HM92, GR91, CD95, Rip90].

6.1 Time-Defined Delays

In the Ada programming language [Int95, Sha86], a time-defined delay is expressed by either of two statements:

```
delay delaytime
delay until delaytime
```

delay specifies a delay time relative to the start of execution of the statement (i.e., a duration), whereas delay until specifies a delay to an absolute time in the future.

In μC++, time-defined delays is expressed by either of two statements:

```
uTimeout( duration );                // parenthesis required
uTimeout( time );                    // parenthesis required
```

With a duration value, **uTimeout** works in the same way as Ada's delay statement. That is, a task blocks for the span of time indicated by the duration value; the task does not consume any resources during this period, nor does it respond to any requests. A time value behaves the same as Ada's delay until statement. That is, a task sleeps until the specified absolute time in the future occurs. If the duration value be less than or equal to zero, the task does not block. Similarly, if the time value that has already occurred, the task does not block.

6.2 Duration and Time

The convenient manipulation of **time** is an essential characteristic in any time constrained environment. Manipulating time, in turn, yields another metric that expresses a span or **duration** of time. uDuration is a class whose instances represent a span of time, e.g., subtracting two time values results in a difference that is a time duration (2:00 - 1:30 = 30 minute duration). The creation and manipulation of uDuration values are performed through the member routines of class uDuration (see Figure 6.1).

Often, uDuration objects are created implicitly when manipulating time; however, specific values can be created by specifying the seconds and nanoseconds in a uDuration's constructor, e.g.:

```
uDuration x( 4, 447398253);   // 4 second & 447398253 nanosecond duration
uDuration y();                // 0 duration
uDuration z( 5 );             // 5 second duration
```

Arithmetic manipulation of uDuration objects is illustrated by:

```
class uDuration {
  public:
    uDuration();
    uDuration( long int sec );
    uDuration( long int sec, long int nsec );

    uDuration &operator-=( uDuration op );
    uDuration &operator+=( uDuration op );
    uDuration &operator*=( long int op );
    uDuration &operator/=( long int op );
    operator timeval();
}; // uDuration

uDuration operator-( uDuration op );
uDuration operator-( uDuration op1, uDuration op2 );
uDuration operator+( uDuration op );
uDuration operator+( uDuration op1, uDuration op2 );
uDuration operator*( uDuration op1, long int op2 );
uDuration operator*( long int op1, uDuration op2 );
uDuration operator/( uDuration op1, long int op2 );
long long int operator/( uDuration op1, uDuration op2 );
bool operator>( uDuration op1, uDuration op2 );
bool operator<( uDuration op1, uDuration op2 );
bool operator>=( uDuration op1, uDuration op2 );
bool operator<=( uDuration op1, uDuration op2 );
bool operator==( uDuration op1, uDuration op2 );
bool operator!=( uDuration op1, uDuration op2 );
uOStream &operator<<( uOStream &os, const uDuration op );
```

Figure 6.1: Duration Class

```
uDuration x, y, z;
int n;

x = y + z;    // add two uDurations producing a uDuration
x = y - z;    // subtract two uDurations producing a uDuration
x = y * n;    // multiply a uDuration n times
x = n * y;    // multiply a uDuration n times
x = y / n;    // divide a uDuration by n
timeval  t1 = x; // convert to UNIX time value (seconds, microseconds)
timespec t2 = x; // convert to UNIX time value (seconds, nanoseconds)
```

In addition, relational comparison operators are defined for uDuration objects.

uTime is a class, whose instance represent an absolute time. Time can be specified using some combination of year, month, day, hour, minute, second, and nanosecond. It is important to note that a time value must be in the range 00:00:00 UTC, January 1, 1970 to 03:14:07 UTC, January 19, 2038, which is the UNIX start and end epochs. The creation and manipulation of uTime values are performed through the member routines of class uTime (see Figure 6.2).

The overloaded constructor routines uTime provide a choice of specifying a time value. The parameters have the following meanings:

year – a year greater than or equal to 1970 and less than or equal to 2038.

month – a number between 0 and 11 inclusive, where 0 represents January and 11 represented December. The default value for a constructor without this argument is 0.

day – a number between 0 and 30 inclusive, where 0 represents the first day of the month and 30 the last day. The default value for a constructor without this argument is 1.

```
class uTime {
  public:
    uTime();
    uTime( long int sec );
    uTime( long int sec, long int nsec );
    uTime( int min, int sec, long int nsec );
    uTime( int hour, int min, int sec, long int nsec );
    uTime( int day, int hour, int min, int sec, long int nsec );
    uTime( int month, int day, int hour, int min, int sec, long int nsec );
    uTime( int year, int month, int day, int hour, int min, int sec, long int nsec );

    uTime &operator-=( uDuration op );
    uTime &operator+=( uDuration op );
    operator timeval();
}; // uTime

uDuration operator-( uTime op1, uTime op2 );
uTime operator-( uTime op1, uDuration op2 );
uTime operator+( uTime op1, uDuration op2 );
uTime operator+( uDuration op1, uTime op2 );
bool operator>( uTime op1, uTime op2 );
bool operator<( uTime op1, uTime op2 );
bool operator>=( uTime op1, uTime op2 );
bool operator<=( uTime op1, uTime op2 );
bool operator==( uTime op1, uTime op2 );
bool operator!=( uTime op1, uTime op2 );
uOStream &operator<<( uOStream &os, const uTime op );
```

Figure 6.2: Time Class

hour – a number between 0 and 23 inclusive, where 0 represents 12:00am and 23 represents 11:00pm. The default value for a constructor without this argument is 0.

min – a number between 0 and 59 inclusive, where 0 is the first minute of the hour and 59 the last. The default value for a constructor without this argument is 0.

sec – a number between 0 and 59 inclusive, where 0 is the first second of the minute and 59 the last.

nsec – a number between 0 and 999999999 inclusive, where 0 is the first nanosecond of the second and 999999999 the last.

It is permissible to *exceed* the logical ranges for the time components; any excess is accumulative, e.g., the following declarations are valid:

```
uTime t1( 0,48,0,60,1000000000 );      // 1970 Jan 3 0:01:01:000000000 (GMT)
uTime t2( 818227413, 0 );              // 1995 Dec 6 05:23:33:000000000 (GMT)
```

As for uDuration values, arithmetic and logical operations may be performed on uTime values. As well, mixed mode operations are possible involving durations and time. A duration may be added to or subtracted from a time to yield a new time; two times can be subtracted from each other producing a duration.

6.3 Timeout Operations

It is undesirable to have operations that might block indefinitely. Two such common operations are waiting for an accepted call to occur and waiting for I/O to complete. In order to mitigate this problem, both these operations provide a timeout facilities, which terminates the operation after a certain period of time.

6.3.1 Accept

The **uAccept** statement is extended to allow the specification of a timeout value: As mentioned briefly (Section 6.1, the simple form of the **uTimeout** statement is:

```
uWhen ( conditional-expression )                // optional guard
    uTimeout( duration or time value )          // optional timeout clause
```

A **uWhen** guard is considered true if it is omitted or its *conditional-expression* evaluates to non-zero. Before the **uTimeout** statement is executed, the guard must be true.

The extended form of the **uAccept** statement may use the **uTimeout** clause, as in:

```
uWhen ( conditional-expression )                // optional guard
    uAccept( mutex-member-name )
        statement                               // statement
...
    ...
        ...
uOr uWhen ( conditional-expression )            // optional guard
    uTimeout( duration or time value )          // optional timeout clause
        statement                               // statement
```

The **uTimeout** clause and the **uElse** clause (see Section 2.8.2.1) cannot be used in the same **uAccept** statement. Like the **uElse** clause, the **uTimeout** clause must be the last clause in a **uAccept** statement. If a **uAccept** clause can be accepted immediately, the statement behaves exactly like a normal **uAccept** statement. If no **uAccept** clause can be accepted immediately and the optional guard on the **uTimeout** statement is true, the task only remains accept blocked until either a call arrives to an accepted member or the specified delay expires. If a call to an appropriate member occurs before the delay value expires, the **uAccept** statement behaves normally. If the delay expires, the acceptor is removed from the acceptor/signalled stack, restarts, and executes the statement associated with the **uTimeout** clause.

□ **WARNING:** Beware of the following mistake when using **uTimeout**:

```
uAccept( fred ) {                   uAccept( fred ) {
} uTimeout( uDuration( 1, 0 ) );    } uOr uTimeout( uDuration( 1, 0 ) );
```

The left example accepts a call to member fred and then delays for 1 second. The right example accepts a call to member fred or times out in 1 second. The left example is two separate accept statements, while the right example is a single accept statement. □

6.3.2 I/O

Similarly, timeouts can be set for certain I/O operations that block waiting for an event to occur (see details in Appendix C.5 & 3.4). Only a duration is allowed as a timeout because a relationship between absolute time and I/O seems unlikely. A pointer to the duration value is used so it is possible to distinguish between no timeout value (NULL pointer) and a zero timeout value. The former usually means to wait until the event occurs (i.e., no timeout), while the latter can be used to poll by trying the operation and returning immediately if the event has not occurred. The I/O operations that can set timeouts are read, (readvIf the specified I/O operation has not completed when the delay expires, the I/O operation fails by throwing an abnormal event. The abnormal events are uReadTimeout for read, (readvFor example, in:

```
try {
    uDuration d( 3, 0 );        // 3 second duration
    fa.read( buf, 512, &d );
    // handle successful read
} catch( uFileIO::uReadTimeout ) {
    // handle read failure
}
```

the read operation expires after 3 seconds if no data has arrived.

As well, a timeout can be set for the constructor of a uSocketAccept and uSocketClient object, which implies that if the acceptor or client has not made a connection when the delay expires, the declaration of the object fails by throwing an abnormal event. For example, in:

```
try {
    uDuration d( 60, 0 );       // 60 second duration
    uSocketAccept acceptor( sockserver, &d );       // accept a connection from a client
    // handle successful accept
} catch( uSocketAccept::uOpenTimeout ) {
    // handle accept failure
} // try
```

See, also, the server examples in Appendix C.6.

6.4 Clock

A clock defines an absolute time and is used for interrogating the current time. Multiple clocks can exist; each one can be set to a different time. In theory, all clocks tick together at the lowest clock resolution available on the computer.

The type uClock creates a clock object, and is defined:

```
class uClock {
  public:
    uClock();
    uClock( uTime adj );
    uTime uGetTime();
    void uGetTime( int &year, int &month, int &day, int &hour, int &minutes,
                   int &seconds, long int &nsec );
    void uResetClock();
    void uResetClock( uTime adj );
}; // uClock
```

The overloaded constructor routine uClock has the following forms:

uClock() – creates the clock as real-time clock running at the same time as the underlying virtual process.

uClock(uTime adj) – creates the clock as a virtual clock starting at time adj.

The overloaded member routine uGetTime returns the current time, and each routine has the following form:

uGetTime() – this form returns the current time as a uTime value, i.e., in nanoseconds from the start of the UNIX epoch.

void uGetTime(**int** &year, **int** &month, **int** &day, **int** &hour, **int** &minutes, **int** &seconds, **long int** &nsec) – this form returns the current time broken up into the traditional non-fixed radix units of time.

The overloaded member routine uResetClock resets the kind of clock between real-time and virtual, and each routine has the following form:

uResetClock() – this form sets the clock to a real-time clock, so it returns the current time of the underlying virtual processor.

void uResetClock(uTime adj) – this form sets the clock to a virtual clock starting at time adj.

As mentioned, each virtual processor has its own real-time clock. The current time is available from a virtual processor via the call uThisProcessor().uGetClock().uGetTime(); hence, it is unnecessary to create a clock to get the current time.

6.5 Periodic Task

Without a programming language construct to specify periodicity, and without programming language facilities to express time, it is almost impossible to accurately express time specifications within a program. Specifying a periodic task in a language without proper time constructs, for example, can introduce catastrophic inaccuracies. For example, in:

```
1   for ( ; ; ) {
2       // periodic work
3       uDuration DelayTime = NextTime - CurrentTime();
4       uTimeout( DelayTime );
5   }
```

if the task is context-switched after executing line 3 (or context-switched after the call to CurrentTime in line 3), the DelayTime would be inaccurate. As a result, the sleep time of the program is erroneous.

The above problem can be eliminated by specifying an absolute time to **uTimeout** (specifying NextTime as the parameter to **uTimeout**). However, with this form of periodic task specification, it is infeasible to specify other forms of deadlines. Ada only supports the periodic task specification using delays, and the system guarantees a periodic task delays for a minimum time specified in DelayTime, but makes no guarantee as to when the periodic task actually gets to execute [BP91]. As a result, a task can request to be put to sleep for 10 seconds (and Ada guarantees it sleeps for at least 10 seconds), but end up executing 20 seconds later.

To circumvent this problem, μC++ provides a periodic task. The general form of the periodic task type is the following:

```
uPeriodicTask 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
};
```

Like a task, a periodic task type has one distinguished member, named main, in which the new thread starts execution. If not derived from some other periodic task type, each periodic task type is implicitly derived from the task type uPeriodicBaseTask, as in:

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

where the interface for the base class uPeriodicBaseTask is:

```
uTask uPeriodicBaseTask {
  protected:
    uTime uFirstActivateTime;
    uEvent uFirstActivateEvent;
    uTime uEndTime;
    uDuration uPeriod;
  public:
    uPeriodicBaseTask( uDuration Period, uCluster &cluster = uThisCluster() );
    uPeriodicBaseTask( uDuration Period, uTime FirstActivateT, uTime EndTime,
                       uDuration Deadline, uCluster &cluster = uThisCluster() );
    uPeriodicBaseTask( uDuration Period, uEvent FirstActivateE, uTime EndTime,
                       uDuration Deadline, uCluster &cluster = uThisCluster() );
    uPeriodicBaseTask( uDuration Period, uTime FirstActivateT, uEvent FirstActivateE,
                       uTime EndTime, uDuration Deadline, uCluster &cluster = uThisCluster() );
    uDuration uGetPeriod() const;
    uDuration uSetPeriod( uDuration period );
};
```

A periodic task starts by one of two mechanisms. The first is by specifying a start time, FirstActivateT, at which the periodic task begins execution. The second is by specifying an event, FirstActivateE (an interrupt), upon receipt the event the periodic task begins execution. If both start time and event are specified, the task starts either on receipt of an event or when the specified time arrives, whichever comes first. If both time and event are not specified, the periodic task starts immediately. An end time, EndTime, may also be specified. When the specified end time occurs, the periodic task halts after execution of the current period. A deadline, Deadline, may also be specified. A deadline

is expressed as the duration from the beginning of a task's period by which its computation must be finished. A zero argument for any of the parameters indicates the task is free from the constraints represented by the parameter (the exception is Period, which cannot have a zero argument). For example, if the FirstActivate parameter is zero, the task is scheduled for initial execution at the next available time it can be accommodated. Finally, the cluster parameter specifies which cluster the task should be created in. Should this parameter be omitted, the task is created on the current cluster.

An example of a periodic task declaration that starts at a specified time and executes indefinitely (without any deadline constraints) is:

```
uPeriodicTask task-name {
    void main() { periodic task body }
  public:
    task-name( uDuration period, uTime time ) : uPeriodicBaseTask( period, time, 0, 0 ) { };
};
```

The task body, i.e., routine main, is implicitly surrounded with a loop that performs the task body periodically. As a result, terminating the task body requires a **return** (or use an end time); falling off the end of the main routine does not terminate a periodic task.

6.6 Sporadic Task

A sporadic task is similar to a periodic task, except there is a minimum duration between executions instead of a fixed period. In the declaration of a sporadic task, this minimum duration is specified as a **frame**. It is the user's responsibility to ensure the execution does not exceed the specified minimum duration (i.e., frame); otherwise, the scheduler cannot ensure correct execution. The reason the scheduler cannot automate this process, as it does for periodic tasks, is because of the unpredictable nature of the inter-arrival time of sporadic tasks.

In μC++, a sporadic task is similar to a periodic task. A **uSporadicTask** task type, if not derived from some other sporadic task type, is implicitly derived from the task type uSporadicBaseTask, as in:

```
uTask uSporadicBaseTask {
  protected:
    uTime uFirstActivateTime;
    uEvent uFirstActivateEvent;
    uTime EndTime;
    uDuration uFrame;
  public:
    uSporadicBaseTask( uDuration Frame, uCluster &cluster = uThisCluster() );
    uSporadicBaseTask( uDuration Frame, uTime FirstActivateT, uTime EndTime,
                       uDuration Deadline, uCluster &cluster = uThisCluster() );
    uSporadicBaseTask( uDuration Frame, uEvent FirstActivateE, uTime EndTime,
                       uDuration Deadline, uCluster &cluster = uThisCluster() );
    uSporadicBaseTask( uDuration Frame, uTime FirstActivateT, uEvent FirstActivateE,
                       uTime EndTime, uDuration Deadline, uCluster &cluster = uThisCluster() );
    uDuration uGetFrame() const;
    uDuration uSetFrame( uDuration frame );
};
```

6.7 Aperiodic Task

An aperiodic task has a non-deterministic start pattern. As a result, aperiodic tasks should only be used in soft real-time applications.

In μC++, an aperiodic task is similar to a periodic task. A **uRealTimeTask** task type, if not derived from some other aperiodic task type, is implicitly derived from the task type **uRealTimeTask**, as in:

```
uTask uRealTimeBaseTask {
  protected:
    uTime uFirstActivateTime;
    uEvent uFirstActivateEvent;
    uTime EndTime;
  public:
    uRealTimeBaseTask( uCluster &cluster = uThisCluster() );
    uRealTimeBaseTask( uTime FirstActivateT, uTime EndTime, uDuration Deadline,
                       uCluster &cluster = uThisCluster() );
    uRealTimeBaseTask( uEvent FirstActivateE, uTime EndTime, uDuration Deadline,
                       uCluster &cluster = uThisCluster() );
    uRealTimeBaseTask( uTime FirstActivateT, uEvent FirstActivateE, uTime EndTime,
                       uDuration Deadline, uCluster &cluster = uThisCluster() );
    uDuration uGetDeadline() const;
    uDuration uSetDeadline( uDuration Deadline );
};
```

6.8 Priority Inheritance Protocol

The **priority-inheritance protocol**, attacks the problem of priority-inversion, where a high priority task waits while lower priority tasks prevent it from executing. Rajkumar proposed the **basic priority-inheritance protocol** [Raj91, RSL88, SRL90], which puts a bound on the occurrence of priority inversion. The solution is to execute a critical section at the priority of the highest blocked task waiting to enter it. The basic priority-inheritance protocol bounds the time priority inversion occurs: should there be n lower priority tasks in the system, and the n lower priority tasks access m distinct critical sections, a task can be blocked by at most $min(n, m)$ critical sections. Despite this bound, the blocking duration for a task can still be significant, however. Suppose at time t_0, a low priority task τ_2 arrives and locks monitor M_0. At time t_1, a medium priority task τ_1 arrives, pre-empts τ_2, and locks monitor M_1. At time t_2, a high priority task τ_0 arrives, needing to sequentially access both monitors M_0 and M_1. Since both monitors are locked by two lower priority tasks, τ_0 must wait for the duration of two critical sections (till M_1 is released by τ_1, then, M_0 is released by τ_2). This problem is known as **chain blocking**. Finally, this priority-inheritance protocol does not deal with the problem of deadlock.

In μC++, tasks waiting for entry into a mutex object on a prioritized entry queue. More specifically, each of the mutex object's member routines have an associated prioritized entry queue. When the mutex object becomes unlocked, the next task that enters is the one with the highest priority among all the entry queues. Should a mutex object be locked and a higher priority task arrives, the current task executing inside the mutex object "inherits" the priority of the highest priority task awaiting entry. This semantics ensures the task inside the mutex object can only be interrupted by a higher priority task, allowing the task in the mutex object to complete and leave as soon as possible, which speeds entry of a waiting higher priority task.

Condition variables and their associated queues of waiting tasks are also a fundamental part of mutex objects. Signalling a condition variable makes the highest priority task on the queue eligible to run. In μC++, the signaller continues execution after signalling, at the priority of the highest priority task awaiting entry to the mutex object. As well, the signalled task is given preference over other tasks awaiting entry to the mutex object. Therefore, the signalled task is the next to execute in the mutex object, regardless of whether there are higher priority tasks waiting in the entry queues. This behaviour, in turn, creates the possibility of priority-inversion. Should a high priority task be awaiting entry to the mutex object, and a lower priority task executing in the mutex object signals a condition queue whose most eligible task has a lower priority than a task awaiting entry to the mutex object, priority-inversion results. Hence, the semantics of μC++ mutex objects increases the original algorithm's bound for priority-inversion by the amount it takes to complete the execution of all the tasks in the signalled stack.

Finally, in μC++, tasks running inside a mutex object have the additional capability of specifically accepting any one of the mutex member routines. This capability also brings about the possibility of bypassing higher priority tasks waiting on other entry queues. When a member routine is accepted, the acceptor is moved to the signalled stack, thus causing the acceptor to block; the highest priority task waiting on the accepted member routine then executes. When a task leaves a mutex object, the next task that executes is selected first from the signalled stack not the entry queues. Thus, the amount of time when priority-inversion can take place when accepting specific member routines is

unbounded, since tasks can continually arrive on a member routine's entry queue, and tasks executing in the mutex object can continually accept the same specific member routine.

6.9 Real-Time Scheduling

The notion of priority becomes a crucial tool for implementing various forms of scheduling paradigms [AGMK94, BW90, Gol94]. In general, the term **priority** has no single meaning. The priority of a task may signify its logical importance to a programmer, or may simply be a property determined by its periodic characteristics, as is the case with certain scheduling algorithms.

In μC++, the notion of priority simply determines the order by which a set of tasks executes. As far as the real-time system is concerned, the ready task with the highest priority is the most eligible task to execute, with little or no regard for the possible starvation of lower priority ready tasks. This form of scheduling is referred to as a **prioritized pre-emptive scheduling**.

Each task's priority can be redefined and queried by the routines provided from the following abstract class (discussed further in the next section):

```
template<class Node> class uBaseSchedule {
  public:
    enum uTaskCategory { uNonRealtime, uRealtime, uPeriodic, uSporadic };
  protected:
    uTaskCategory uGetCategory( uBaseTask &t ) const;
    uBaseTask &uGetInheritTask( uBaseTask &t ) const;
    int uGetActivePriority( uBaseTask &t ) const;
    int uSetActivePriority( uBaseTask &t1, uBaseTask &t2 );
    int uGetBasePriority( uBaseTask &t ) const;
    int uSetBasePriority( uBaseTask &t, int p );
}; // uBaseScheduleFriend
```

Scheduler objects inherit from uBaseSchedule to use these routines, not replace them. These routines provide sufficient information about the dynamic behaviour of tasks on a cluster to schedule them in various ways.

To provide the facilities for implementing various priority-changing scheduling algorithms (such as priority-inheritance), a μC++ task has two priorities associated with it: a **base priority** and an **active priority**. It is up to the scheduler implementor or programmer to set the appropriate priority values, or to determine whether the base priority or the active priority is the priority utilized in scheduling tasks, if used at all.[1]

The member routine uGetCategory returns the kind of task: uNonRealtime, uRealtime, uPeriodic, and uSporadic. Since tasks can migrate from one cluster to another, knowing the kind of task in the scheduler may be important in its scheduling. The member routine uGetInheritTask returns the task that this task inherited its current active priority from or NULL. The member routines uGetActivePriority and uSetActivePriority read and write a task's active priority, respectively. The member routines uGetBasePriority and uSetBasePriority read and write a task's base priority, respectively.

A task's priority can be used for more than just determining which task executes next; priorities can also dictate the behaviour of various synchronization primitives such as semaphores and monitors [BW90]. μC++ monitors have been extended so that entry queues (see Section 2.8.1) are prioritized.[2] The highest priority task that calls into a monitor always enters the monitor first, unless a particular entry queue is explicitly accepted (see Section 2.8.2.1), in which case, the highest priority task in the particular entry queue executes. Condition queues (see Section 2.8.3.1) within a monitor are also prioritized: signaling a condition queue schedules the highest priority task waiting on the queue. Thus, both the monitor entry queues and the condition queues are prioritized, with FIFO used within each priority level. The current implementation provides 32 priority levels. Support for more or less priority levels can be implemented (see Section 6.10).

If an application is not real-time, all tasks are assigned an equal, default priority level. Thus, all tasks have one active priority, and the scheduling is FIFO.

[1] μC++ sets the base and the active priority of a task to a uniform default value, if no other priority is specified.

[2] A task's active priority is utilized by a μC++ monitor to determine a task's priority value

6.10 User Supplied Scheduler

One of the goals of real-time in μC++ is to provide a flexible system, capable of being adapted to various real-time environments and applications. The wide availability of various real-time scheduling algorithms, coupled with each algorithm's suitability for different forms of real-time applications, makes it essential that the language and runtime system provide as few restrictions as possible on which algorithms may be utilized and implemented.

Scheduling is the mechanism by which the next task to run is chosen from a set of runnable tasks. However, this selection mechanism is closely tied to the data-structure representing the set of runnable tasks. In fact, the data-structure containing the set of runnable tasks is often designed with a particular scheduling algorithm in mind.

To provide a flexible scheduler, the ready "queue"[3] is packaged as an independent entity – readily accessible and replaceable by a scheduler designer. Consequently, the rules and mechanisms by which insertion and removal take place from the ready data structure is completely up to the implementor.

A ready data structure is generic in the type of nodes stored in the structure and must inherit from the abstract class:

```
template<class Node> class uBaseSchedule {
  public:
    virtual void uAdd( Node *node ) = 0;
    virtual Node *uPop() = 0;
    virtual bool uEmpty() const = 0;
    virtual bool uCheckPriority( Node &owner, Node &calling ) = 0;
    virtual void uResetPriority( Node &owner, Node &calling ) = 0;
    virtual void uAddInitialize( uSequence<uBaseTaskDL> &TaskList ) = 0;
    virtual void uRemoveInitialize( uSequence<uBaseTaskDL> &TaskList ) = 0;
    virtual void uRescheduleTask( uBaseTaskDL *TaskNode, uBaseTaskSeq &TaskList ) = 0;
};
```

The μC++ kernel uses the routines provided by uBaseSchedule to interact with the user-defined ready queue.[4] A user can construct different scheduling algorithms by modifying the behaviour of member routines uAdd and uPop, which add and remove tasks from the ready queue, respectively. To implement a dynamic scheduling algorithm, an analysis of the set of runnable tasks is performed for each call to uAdd and/or uPop by the kernel; these routines alter the priorities of the tasks accordingly. The member routine uEmpty returns true if the ready queue is empty and false otherwise. The member routine uCheckPriority provides a mechanism to determine if a calling task has a higher priority than another task, which is used to compare priorities in priority changing protocols like priority inheritance. Its companion routine uResetPriority performs the same check, but also raises the priority of the owner task to that of the calling task if necessary. uAddInitialize is called by the kernel whenever a task is added to the cluster, and uRemoveInitialize is called by the kernel whenever a task is deleted from the cluster. In both cases, a pointer to the ready queue for the cluster is passed as an argument so it can be reorganized if necessary. The type uSequence<uBaseTaskDL> is the type of a system ready queue (see Section B for information about the uSequence collection). The list node type, uBaseTaskDL, stores a reference to a task, and this reference can be retrieved with member routine uGet:

```
class uBaseTaskDL : public uSeqable {
  public:
    uBaseTaskDL( uBaseTask &w );
    uBaseTask &uGet();
}; // uBaseTaskDL
```

Note, adding (or deleting) tasks to (or from) a cluster is not the same as adding or popping tasks from the ready queue. With a static scheduling algorithm, for example, task set analysis is only performed upon task creation, making the uAddInitialize function an ideal place to specify such analysis code. The member routine uRescheduleTask is used to recalculate the priorities of the tasks on a cluster based on the fact that a given task, TaskNode, may have changed some of its scheduling attributes.

[3] The term "ready queue" is no longer appropriate because the data structure may not be a queue.

[4] Operating systems such as Amoeba[TvRvS+90], Chorus[RAA+88], and Apertos[Yok92] employ a similar mechanism by which the kernel utilizes external modules to modify its behaviour.

6.11 Real-Time Cluster

A **real-time cluster** behaves just like a normal μC++ cluster, except a real-time cluster can have a special ready data-structure associated with it (the ready data-structure, in turn, has a scheduling or task dispatching policy associated with it). The ready data-structure must inherit from the uBaseSchedule class, however, and passed as an argument when creating a real-time cluster. A real-time cluster has the following constructors:

```
class uRealTimeCluster : public uCluster {
  public:
    uRealTimeCluster( uBaseSchedule<uBaseTaskDL> &rq, int size = uDefaultStackSize(),
                      const char *name = "" );
    uRealTimeCluster( uBaseSchedule<uBaseTaskDL> &rq, const char *name );
    ~uRealTimeCluster() {};
};
```

6.11.1 Deadline Monotonic Scheduler

The **deadline monotonic** scheduling algorithm is an example of particular task-dispatching policy that requires a special ready data structure, which can be plugged into a real-time cluster. The underlying ready data structure for the deadline monotonic implementation is a prioritized ready-queue, with support for 32 priority levels. The uAdd routine adds a task to the ready-queue in a FIFO manner within a priority level. The uPop routine returns the most eligible task with the highest priority from the ready-queue. Both uAdd and uPop utilize a constant-time algorithm for the location of the highest priority task. Figure 6.3 illustrates this prioritized ready-queue.

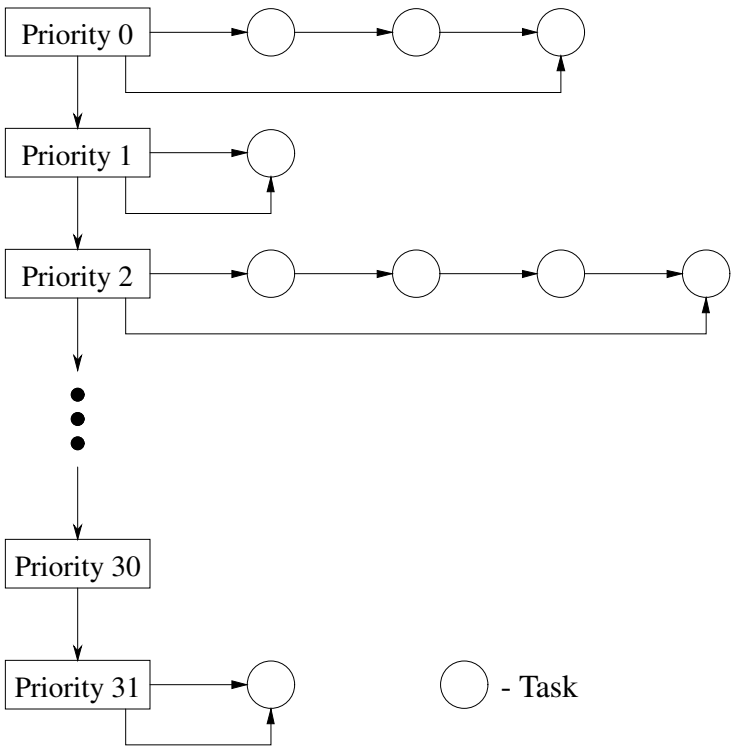


Figure 6.3: Deadline Monotonic Ready-Queue

The uAddInitialize routine contains the heart of the deadline monotonic algorithm. In uAddInitialize, each task in the ready-queue is examined, and tasks are ordered in increasing order by deadline. Priorities are, in turn, assigned to every task. With the newly assigned priorities, the ready queue is re-evaluated, to ensure it is in a consistent state. As indicated in Section 6.10, this routine is usually called only by the kernel. If a task is removed from the cluster, the relative order of the remaining tasks is unchanged; hence, the task is simply deleted without a need to re-schedule.

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

uPeriodicTask PeriodicTask1 {
  public:
    PeriodicTask1( uDuration period, uTime endtime, uDuration deadline, uCluster &clust ) :
                uPeriodicBaseTask( period, uTime(0), endtime, deadline, clust ) {
    }
    void main() {
        // periodic task body
    }
};

uPeriodicTask PeriodicTask2 {
  public:
    PeriodicTask2( uDuration period, uTime endtime, uDuration deadline, uCluster &clust ) :
                uPeriodicBaseTask( period, uTime(0), endtime, deadline, clust ) {
    }
    void main() {
        // periodic task body
    }
};

void uMain::main() {
    uDeadlineMonotonic dm;                  // create real-time scheduler
    uRealTimeCluster RTClust( dm );         // create real-time cluster with scheduler
    uProcessor *processor;
    {
        // These tasks are created, but they do not begin execution until a
        // processor is created on the "RTClust" cluster. This is ideal, as
        // "uAddInitialize" is called as each task is added to the cluster.

        uTime currTime = uThisProcessor().uGetClock().uGetTime();
        PeriodicTask1 t1( 15, currTime+90,  5, RTClust );       // 15 sec period, 5 sec deadline
        PeriodicTask2 t2( 30, currTime+90, 10, RTClust );       // 30 sec period, 5 sec deadline
        PeriodicTask1 t3( 60, currTime+90, 20, RTClust );       // 60 sec period, 20 sec deadline

        // Only when all tasks are on the cluster, and the scheduling algorithm
        // as ordered the tasks, is a processor associated with cluster
        // "RTClust" to execute the tasks on the cluster.

        processor = new uProcessor( RTClust );
    } // wait for t1, t2, and t3 to finish
    delete processor;
}
```

Figure 6.4: Sample Real-Time Program

A sample real-time program is illustrated in Figure 6.4. To utilize the deadline-monotonic algorithm include header file uDeadlineMonotonic.h. In the example, the creation of the real-time scheduler and cluster is done at the beginning of uMain::main. Note, the argument passed to the constructor of uRealTimeCluster is a instance of uDeadlineMonotonic, which is a ready data-structure derived from uBaseSchedule.

The technique used to ensure that the tasks start at a critical instance is not to associate a processor with the cluster until after all tasks are created *and scheduled* on the cluster. As each task is added to the cluster uAddInitialize is called, and cluster's task set is analyzed and task priorities are (re)assigned. After priority assignment, the task is added to the ready queue, and made eligible to execute. Only when all tasks are created is a processor finally associated with the real-time cluster. This approach ensures that when the processor is put in place, the task priorities are fully determined, and the critical instant is ensured.

□ Beware of implicit task migration resulting from I/O operations. The migration causes calls to uAddInitialize and uRemoveInitialize as a task moves from and subsequently back to the original cluster. During the time it is gone, task priorities may be changed to reflect the current task set, which can result in unexpected behaviour in certain cases. □

Chapter 7

Miscellaneous

7.1 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. Furthermore, 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
```

7.2 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).

7.3 Installation Requirements

μC++ comes configured to run on any of the following platforms:

- sun-sparc-{sunos,svr4} (single and multiple processor)
- dec-ultrix-mips (single processor)
- dec-alpha (Digital UNIX V3.2) (single and multiple processor)
- sgi-mips-r4000 (single and multiple processor)
- sgi-mips-r3000 (single processor)
- ibm-rs6000 (single processor)
- hp-hppa (single processor)
- pc-i386-linux (multiple processor for 2.0.x)
- gizmo (embedded version for stand-alone m68k board) (single processor)

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.

μC++ requires at least GNU C++ 2.91 (egcs-1.1) [Tie90] and egcs-1.1.1 is required on Linux. These compilers can be obtained free of charge. Currently, μC++ does **NOT** compile using other compilers.

7.4 Installation

μC++ requires at least version 4.0 of dmake. The most recent version of dmake is available by anonymous ftp from the following location (remember to set your ftp mode to "binary"):

```
plg.uwaterloo.ca:pub/dmake
```

If the tar.gz version is taken, execute the following command to unpack the source:

```
% gunzip -c dmake{current-version-number}.tar.gz | tar -xf -
```

Look in the readme directory for 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++-{current version number}.tar.gz
```

Execute the following command to unpack the source:

```
% gunzip -c u++-{current-version-number}.tar.gz | tar -xf -
```

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

7.5 Reporting Problems

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

μSystem Project
c/o Peter A. Buhr
Dept. of Computer Science
University of Waterloo

Waterloo, Ontario
N2L 3G1
CANADA

As well, visit the μSystem web site at:
http://plg.uwaterloo.ca/~usystem

7.6 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. Version 4.0 kernel was designed and implemented by Peter Buhr. Nikita Borisov and Peter Buhr fixed several problems in the translator. Amir Michail started the real-time work and built a working prototype. Philipp Lim and Peter Buhr designed the first version of the real-time support and Philipp did most of the implementation with occasional help from Peter Buhr. Ashif Harji and Peter Buhr designed the second version of the real-time support and Ashif did most of the implementation with occasional help from Peter Buhr. Russell Mok started the abnormal even handling and built the first version with occasional help from Peter Buhr. Tom, Sasha, Tom, Raj, and Martin, the "gizmo guys", all helped Peter Buhr with the gizmo port. Finally, the many contributions made by all the students in CS342 (Waterloo) and CSC372 (Toronto), who struggled with earlier versions of μC++, is recognized.

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

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
    uDualEvent
    uRaiseEvent
    uThrowEvent
    uRealTimeTask
    uPeriodicTask
    uSporadicTask
statement:
    ...
    uWait *expression* ;
    uWait *expression* **uWith** *expression* ;
    uSignal *expression* ;
    uSignalBlock *expression* ;
    uSuspend ;
    uResume ;
    accept-statement ;
    throw-statement ;
    uDisable *statement* ;
    uEnable *statement* ;
accept-statement:
    when-clause$_{opt}$ **uAccept** (*dname*) *statement*
    when-clause$_{opt}$ **uAccept** (*dname*) *statement* **uOr** *accept-statement*
    when-clause$_{opt}$ **uAccept** (*dname*) *statement* **uTimeout** (*expression*) *statement*
    when-clause$_{opt}$ **uAccept** (*dname*) *statement* **uElse** *statement*
    when-clause$_{opt}$ **uTimeout** (*expression*) *statement*
when-clause:
    uWhen (*expression*)
throw-statement:
    uThrow *expression* ;

uThrow *expression* **uAt** *expression* ;
uRaise *expression* ;
uRaise *expression* **uAt** *expression* ;

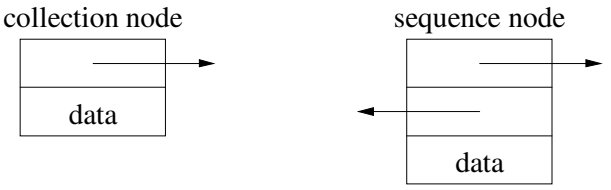
Appendix B

Data Structure Library (DSL)

μC++ makes use of several basic data structures to manage objects in its runtime environment: stack, queue and sequence. Since these data structures are needed at compile time because of inlining, it is possible to use them in a μC++ application program. When appropriate, reusing code by an application programmer can save significant time and effort. However, be forewarned that the μC++ DSL is only as extensive as needed to implement μC++; it is not meant to be a complete data structure library (such as LEDA or the STL).

A data structure is defined to be a group of nodes, containing user data, organized into a particular format, with specific operations peculiar to that format. For all data structures in this library, it is the user's responsibility to create and delete all nodes. Because a node's existence is independent of the data structure that organizes it, all nodes are manipulated by address not value; hence, all data structure routines take and return pointers to nodes and not the nodes themselves.

Nodes are divided into two kinds: those with one link field, which form a collection, and those with two link fields, which form a sequence.



uStack and uQueue are collections and uSequence is a sequence. To get the appropriate link fields associated with a user node, it must be a public descendant of uColable or uSeqable, respectively, e.g.:

```
class stacknode : public uColable { ... }
class queuenode : public uColable { ... }
class seqnode : public uSeqable { ... }
```

A node inheriting from uSeqable can be put in a collection data structure but not vice versa. Along with providing the appropriate link fields, the types uColable and uSeqable also provide one member routine:

```
bool uListed() const;
```

which returns **true** if the node is an element of any collection or sequence and **false** otherwise.

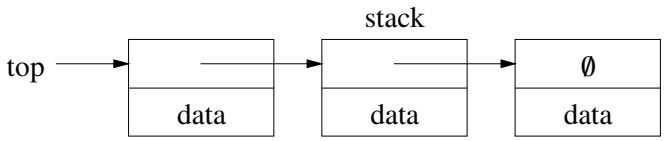
Finally, no header files are necessary to access the μC++ DSL; all necessary definitions are included when file <uC++.h> is included.

Some μC++ DSL restrictions are:

- None of the member routines are virtual in any of the data structures for efficiency reasons. Therefore, pointers to data structures must be used with care or incorrect member routines may be invoked.

B.1 Stack

A uStack is a collection that defines an ordering among the nodes: nodes are returned by uPop in the reverse order that they are added by uPush.




```
template<class T> class uStack {
  public:
    uStack();
    bool uEmpty() const;
    T *uTop() const;
    T *uHead() const
    void uPush( T *n );
    void uAdd( T *n );
    T *uPop();
    T *uDrop();
};
```

T must be a public descendant of uColable.

The member routine uEmpty returns **true** if the stack has no nodes and **false** otherwise. The member routine uTop returns a pointer to the top node of the stack without removing it or NULL if the stack has no nodes. The member routine uHead is a synonym for uTop. The member routine uPush adds a node to the top of the stack. The member routine uAdd is a synonym for uPush. The member routine uPop removes a node from the top of the stack and returns a pointer to it or NULL if the stack has no nodes. The member routine uDrop is a synonym for uPop.

B.1.1 Generator

The generator uStackGen<T> generates a stream of the elements of a uStack<T>.

```
template<class T> class uStackGen {
  public:
    uStackGen();
    uStackGen( const uStack<T> &s );
    void uOver( const uStack<T> &s );
    bool operator>>( T *&tp );
};
```

It is used to iterate over the nodes of a stack from the top of the stack to the bottom.

The overloaded constructor routine uStackGen has the following forms:

uStackGen() – creates the generator without associating it with a particular stack; the association must be done subsequently with member uOver.

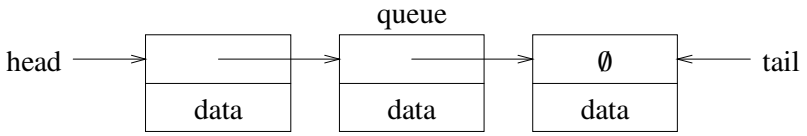
uStackGen(**const** uStack<T> &s) – creates the generator and associate it the specified stack; the association can be changed subsequently with member uOver.

The member routine uOver resets the generator to the top of the specified stack. The member routine >> attempts to move the generator's internal cursor to the next node. If the bottom (end) of the stack has not been reached, the argument is set to the address of the next node and **true** is returned; otherwise the argument is set to NULL and **false** is returned.

Figure B.1 illustrates creating and using a stack and stack generator.

B.2 Queue

A uQueue is a collection that defines an ordering among the nodes: nodes are returned by uDrop in the same order that they are added by uAdd.



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

struct stackNode : public uColable {
    int v;
    stackNode( int v ) : v( v ) {}
};

void uMain::main() {
    uStack<stackNode> stack;
    uStackGen<stackNode> stackgen;
    stackNode *sp;
    int i;

    for ( i = 0; i < 10; i += 1 ) {                    // fill stack
        stack.uPush( new stackNode( 2 * i ) );
    } // for

    for ( stackgen.uOver(stack); stackgen >> sp; ) {   // print stack
        uCout << sp->v << " ";
    } // for
    uCout << endl;

    for ( i = 0; i < 10; i += 1 ) {                    // empty stack
        sp = stack.uPop();
        delete sp;
    } // for
}
```

Figure B.1: DSL Stack

```
template<class T> class uQueue {
  public:
    uQueue();
    bool uEmpty() const;
    T *uHead() const
    T *uTail() const;
    void uAdd(T *n);
    T *uDrop();
};
```

T must be a public descendant of uColable.

The member routine uEmpty returns **true** if the queue has no nodes and **false** otherwise. The member routine uHead returns a pointer to the head or first node of the queue without removing it or NULL if the queue has no nodes. The member routine uTail returns a pointer to the tail or last node of the queue without removing it. The member routine uAdd adds a node to the tail or end of the queue. The member routine uDrop removes a node from the head or front of the queue and returns a pointer to it or NULL if the queue has no nodes.

B.2.1 Generator

The generator uQueueGen<T> generates a stream of the elements of a uQueue<T>.

```
template<class T> class uQueueGen {
  public:
    uQueueGen();
    uQueueGen( const uQueue<T> &q );
    void uOver( const uQueue<T> &q );
    bool operator>>( T *&tp );
};
```

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

struct queueNode : public uColable {
    int v;
    queueNode( int v ) : v( v ) {}
};

void uMain::main() {
    uQueue<queueNode> queue;
    uQueueGen<queueNode> queuegen;
    queueNode *qp;
    int i;

    for ( i = 0; i < 10; i += 1 ) {                    // fill queue
        queue.uAdd( new queueNode( 2 * i ) );
    } // for

    for ( queuegen.uOver(queue); queuegen >> qp; ) {   // print queue
        uCout << qp->v << " ";
    } // for
    uCout << endl;

    for ( i = 0; i < 10; i += 1 ) {                    // empty queue
        qp = queue.uDrop();
        delete qp;
    } // for
}
```

Figure B.2: DSL Queue

It is used to iterate over the nodes of a queue from the head of the queue to the tail.

The overloaded constructor routine uQueueGen has the following forms:

uQueueGen() – creates the generator without associating it with a particular queue; the association must be done subsequently with member uOver.

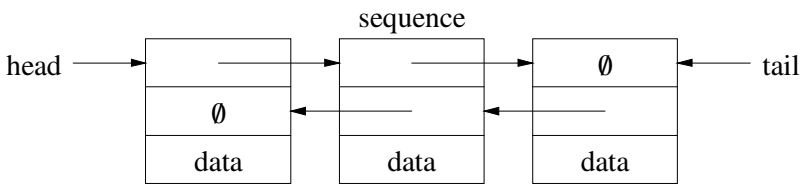
uQueueGen(**const** uQueue<T> &q) – creates the generator and associate it the specified queue; the association can be changed subsequently with member uOver.

The member routine uOver resets the generator to the head of the specified queue. The member routine >> attempts to move the generator's internal cursor to the next node. If the tail (end) of the queue has not been reached, the argument is set to the address of the next node and **true** is returned; otherwise the argument is set to NULL and **false** is returned.

Figure B.2 illustrates creating and using a queue and queue generator.

B.3 Sequence

A uSequence is a collection that defines a bidirectional ordering among the nodes: nodes can be added and removed from either end of the collection; furthermore, nodes can be inserted and removed anywhere in the collection.



```
template<class T> class uSequence {
  public:
    uSequence();
    bool uEmpty() const;
    T *uHead() const
    T *uTail() const;
    T *uSucc(T *n) const;
    T *uPred(T *n) const;
    void uInsertBef(T *n, T *bef);
    void uInsertAft(T *aft, T *n);
    void uRemove(T *n);
    void uAddHead(T* n);
    T *uDropHead();
    T *uDrop();
    void uAddTail(T* n);
    void uAdd(T* n);
    T *uDropTail();
};
```

T must be a public descendant of uSeqable.

The member routine uEmpty returns **true** if the sequence has no nodes and **false** otherwise. The member routine uHead returns a pointer to the head or first node of the sequence without removing it or NULL if the sequence has no nodes. The member routine uTail returns a pointer to the tail or last node of the sequence without removing it or NULL if the sequence has no nodes. The member routine uSucc returns a pointer to the successor node after the specified node (toward the tail) or NULL if the specified node is the last node in the sequence. The member routine uPred returns a pointer to the predecessor node before the specified node (toward the head) or NULL if the specified node is the first node in the sequence. The member routine uInsertBef adds a node before the specified node or at the end (tail) if bef is NULL. The member routine uInsertAft adds a node after the specified node or at the beginning (head) if aft is NULL. The member routine uRemove removes the specified node from the sequence (any location). The member routine uAddHead adds a node to the head or front of the sequence. The member routine uDropHead removes a node from the head or front of the sequence and returns a pointer to it or NULL if the sequence has no nodes. The member routine uDrop is a synonym for uDropHead. The member routine uAddTail adds a node to the tail or end of the sequence. The member routine uAdd is a synonym for uAddTail. The member routine uDropTail removes a node from the tail or end of the sequence and returns a pointer to it or NULL if the sequence has no nodes.

A sequence behaves like a queue when members uAdd and uDrop are used. The example program in Section C.4 makes use of a sequence and modifies it so that nodes are maintained in order.

B.3.1 Generator

The generator uSeqGen<T> generates a stream of the elements of a uSequence<T>.

```
template<class T> class uSeqGen {
  public:
    uSeqGen();
    uSeqGen(const uSequence<T> &s);
    void uOver(const uSequence<T> &s);
    bool operator>>(T *&tp);
};
```

It is used to iterate over the nodes of a sequence from the head of the sequence to the tail.

The generator uSeqGenRev<T> generates a stream of the elements of a uSequence<T>.

```
template<class T> class uSegGenRev {
  public:
    uSegGenRev();
    uSegGenRev(const uSequence<T> &s);
    void uOver(const uSequence<T> &s);
    bool operator>>(T *&tp);
};
```

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

struct seqNode : public uSeqable {
    int v;
    seqNode( int v ) : v( v ) {}
};

void uMain::main() {
    uSequence<seqNode> seq;
    uSeqGen<seqNode> seqgen;
    seqNode *sp;
    int i;

    for ( i = 0; i < 10; i += 1 ) {                    // fill sequence
        seq.uAdd( new seqNode( 2 * i ) );
    } // for

    for ( seqgen.uOver(seq); seqgen >> sp; ) {         // print sequence forward
        uCout << sp->v << " ";
    } // for
    uCout << endl;

    for ( uSeqGenRev<seqNode> seqgenrev(seq); seqgenrev >> sp; ) { // print sequence reverse
        uCout << sp->v << " ";
    } // for
    uCout << endl;

    for ( seqgen.uOver(seq); seqgen >> sp; ) {         // empty sequence
        seq.uRemove( sp );                             // can remove nodes during iteration
        delete sp;
    } // for
}
```

Figure B.3: DSL Sequence

It is used to iterate over the nodes of a sequence from the tail of the sequence to the head.

The overloaded constructor routine uSeqGen has the following forms:

uSeqGen() – creates the generator without associating it with a particular sequence; the association must be done subsequently with member uOver.

uSeqGen(**const** uSeq<T> &q) – creates the generator and associate it the specified sequence; the association can be changed subsequently with member uOver.

The member routine uOver resets the generator to the head or tail of the specified sequence depending on which generator is used. The member routine >> attempts to move the generator's internal cursor to the next node. If the head (front) or tail (end) of the sequence has not been reached depending on which generator is used, the argument is set to the address of the next node and **true** is returned; otherwise the argument is set to NULL and **false** is returned.

Figure B.3 illustrates creating and using a sequence and sequence generator.

Appendix C

Example Programs

C.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 4.7, Copyright (C) Peter A. Buhr and Richard A. Stroobosscher 1994
//
// BinaryInsertionSort.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 Jan 14 22:25:13 1998
// Update Count : 53
//

#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 = 40;
    BinarySort bs;
    int value;
    int i;

    // sort values

    uCout << "unsorted values:" << endl;
    for ( i = 1; i <= NoOfValues; i += 1 ) {
        value = rand() % 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: //
// tab-width: 4 //
// compile-command: "u++ BinaryInsertionSort.cc" //
// End: //
```

C.2 Readers And Writer

The readers and writer problem deals with controlling access to a resource that can be shared by multiple readers, but only one writer can use it at a time (e.g., a sequential file). While there are many possible solutions to this problem, each solution must deal with unbounded waiting of reader and/or writer tasks if a continuous stream of one kind of task is arriving at the monitor. For example, if readers are currently using the resource, a continuous stream of reader tasks should not make an arriving writer task wait forever. Furthermore, a solution to the readers and writer problem should provide FIFO execution of the tasks so that a read that is requested after a write does not execute before the write, thus reading old information. This phenomenon is called the **stale readers** problem. Hoare gave a monitor solution in [Hoa74] that does not have a unbounded waiting problem, but has non-FIFO execution.

```
// -*- Mode: C++ -*-
//
// uC++ Version 4.7, Copyright (C) Peter A. Buhr and Richard A. Stroobosscher 1994
//
// RWEx1.cc – Readers and Writer Problem
//
// Author : Peter A. Buhr
// Created On : Thu Aug 2 11:51:34 1990
// Last Modified By : Peter A. Buhr
// Last Modified On : Thu Jul 4 20:13:54 1996
// Update Count : 82
//

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


uMonitor ReaderWriter {
    int ReadCount, WriteUsage;
    uCondition ReaderAndWriter;
    enum RW { READER, WRITER };
  public:
    ReaderWriter() {
        ReadCount = WriteUsage = 0;
    } // ReaderWriter

    void StartRead() {
        if ( WriteUsage || !ReaderAndWriter.uEmpty() ) {
            uWait ReaderAndWriter uWith READER;
        } // if
        ReadCount += 1;
        if ( ! ReaderAndWriter.uEmpty() && ReaderAndWriter.uFront() == READER ) {
            uSignal ReaderAndWriter;
        } // if
    } // ReaderWriter::StartRead

    void EndRead() {
```

```
        ReadCount -= 1;
        if ( ReadCount == 0 ) {
            uSignal ReaderAndWriter;
        } // if
    } // ReaderWriter::EndRead

    void StartWrite() {
        if ( WriteUsage || ReadCount != 0 ) {
            uWait ReaderAndWriter uWith WRITER;
        } // if
        WriteUsage = 1;
    } // ReaderWriter::StartWrite

    void EndWrite() {
        WriteUsage = 0;
        uSignal ReaderAndWriter;
    } // ReaderWriter::EndWrite
}; // ReaderWriter


int SharedVar = 0;                                  // shared variable to test readers and writers

uTask Worker {
    ReaderWriter &rw;

    void main() {
        uYield( rand() % 100 );                     // don't all start at the same time
        if ( rand() % 100 < 70 ) {                  // decide to be a reader or writer
            rw.StartRead();
            uCout << uAcquire << "Reader:" << this << ", shared:" << SharedVar << endl << uRelease;
            uYield( 3 );
            rw.EndRead();
        } else {
            rw.StartWrite();
            SharedVar += 1;
            uCout << uAcquire << "Writer:" << this << ", wrote:" << SharedVar << endl << uRelease;
            uYield( 1 );
            rw.EndWrite();
        } // if
    } // Worker::main
  public:
    Worker( ReaderWriter &rw ) : rw( rw ) {
    } // Worker::Worker
}; // Worker

#define MaxTask 50

void uMain::main() {
    ReaderWriter rw;
    Worker *workers;

    workers = new Worker[MaxTask]( rw );
    delete [] workers;

    uCout << uAcquire << "successful completion" << endl << uRelease;
} // uMain::main


// Local Variables: //
// tab-width: 4 //
// compile-command: "u++ RWEx1.cc" //
// End: //
```

C.3 Bounded Buffer

Two processes communicate through a unidirectional queue of finite length.

C.3.1 Using Monitor Accept

```
// -*- Mode: C++ -*-
//
// uC++ Version 4.7, Copyright (C) Peter A. Buhr and Richard A. Stroobosscher 1994
//
// 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 : Thu Dec 3 07:05:33 1998
// Update Count : 115
//

#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: //
// tab-width: 4 //
// compile-command: "u++ MonAcceptBB.cc" //
// End: //
```

C.3.2 Using Monitor Condition

```
// -*- Mode: C++ -*-
//
// uC++ Version 4.7, Copyright (C) Peter A. Buhr and Richard A. Stroobosscher 1994
//
// 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 : Sun Nov 26 18:07:34 1995
// Update Count : 50
//

#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: //
// tab-width: 4 //
// compile-command: "u++ MonConditionBB.cc" //
// End: //
```

C.3.3 Using Task

```
// -*- Mode: C++ -*-
//
// uC++ Version 4.7, Copyright (C) Peter A. Buhr and Richard A. Stroobosscher 1994
//
// 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 : Wed Sep 24 10:34:26 1997
// Update Count : 62
//

#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: //
// tab-width: 4 //
// compile-command: "u++ TaskAcceptBB.cc" //
// End: //
```

C.3.4 Using P/V

```
// -*- Mode: C++ -*-
//
// uC++ Version 4.7, Copyright (C) Peter A. Buhr and Richard A. Stroobosscher 1994
//
// SemaphoreBB.cc --
//
// Author : Peter A. Buhr
// Created On : Thu Aug 15 16:42:42 1991
// Last Modified By : Peter A. Buhr
// Last Modified On : Thu Dec 3 07:04:34 1998
// Update Count : 50
//

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

    BoundedBuffer( BoundedBuffer & );                   // no copy
    BoundedBuffer &operator=( BoundedBuffer & );        // no assignment
  public:
    BoundedBuffer( const int size = 10 ) : size( size ), full( 0 ), empty( size ) {
        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: //
// tab-width: 4 //
// compile-command: "u++ SemaphoreBB.cc" //
// End: //
```

C.4 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 4.7, Copyright (C) Peter A. Buhr and Richard A. Stroobosscher 1994
//
// 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 : Thu Dec 3 07:03:23 1998
// Update Count : 254
//

#include <uC++.h>
#include <uIOStream.h>
#include <unistd.h>                                     // prototype: getpid


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

class WaitingRequest : public uSeqable {                    // element for a waiting request list
    WaitingRequest( WaitingRequest & );                     // no copy
    WaitingRequest &operator=( WaitingRequest & );          // no assignment
  public:
    uCondition block;
    IOStatus status;
    IORequest req;
    WaitingRequest( IORequest req ) {
        WaitingRequest::req = req;
    }
}; // WaitingRequest

class Elevator : public uSequence<WaitingRequest> {
    int Direction;
    WaitingRequest *Current;

    Elevator( Elevator & );                                 // no copy
    Elevator &operator=( Elevator & );                      // no assignment
  public:
    Elevator() {
        Direction = 1;
    } // Elevator::Elevator

    void orderedInsert( WaitingRequest *np ) {
        for ( WaitingRequest *lp = uHead();                 // insert in ascending order by track number
              lp != 0 && lp->req.track < np->req.track;
              lp = uSucc( lp ) );
        if ( uEmpty() ) Current = np;                       // 1st client, so set Current
        uInsertBef( np, lp );
    } // Elevator::orderedInsert

    WaitingRequest *uRemove() {
        WaitingRequest *temp = Current;                     // advance to next waiting client
        Current = Direction ? uSucc( Current ) : uPred( Current );
        uSequence<WaitingRequest>::uRemove( temp );         // remove request

        if ( Current == 0 ) {                               // reverse direction ?
            uCout << uAcquire << "Turning" << endl << ::uRelease;
            Direction = !Direction;
            Current = Direction ? uHead() : uTail();
        } // if
        return temp;
    } // Elevator::uRemove
}; // 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;
        uYield( 100 );                              // pretend to perform an I/O operation
        status = IO_COMPLETE;
    } // for
} // Disk::main

void DiskScheduler::main() {
    uSeqGen<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.uEmpty(); ) {          // service pending disk requests before terminating
        uAccept( WorkRequest );
    } // for
#else
    WaitingRequest *client;                         // cancel pending disk requests before terminating

    for ( gen.uOver(PendingClients); gen >> client; ) {
        PendingClients.uRemove();                   // 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.uEmpty() ) {                    // any clients waiting ?
        uWait DiskWaiting;                              // wait for client to arrive
    } // if

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

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

    uYield( rand() % 100 );                             // don't all start at the same time
    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;

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

    p = new DiskClient[NoOfTests]( scheduler );         // start the clients
    delete [] p;                                        // wait for clients to complete

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

// Local Variables: //
// tab-width: 4 //
// compile-command: "u++ LOOK.cc" //
// End: //
```

C.5 UNIX File I/O

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

```
// -*- Mode: C++ -*-
//
```

```
// uC++ Version 4.7, Copyright (C) Peter A. Buhr and Richard A. Stroobosscher 1994
//
// 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 : Sun Nov 26 18:13:16 1995
// Update Count : 32
//

#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: //
// tab-width: 4 //
// compile-command: "u++ File.cc" //
// End: //
```

C.6 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 or its 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 or its 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 deal with multiple simultaneous clients.

C.6.1 Client - UNIX/Datagram

```
// -*- Mode: C++ -*-
//
// uC++ Version 4.7, Copyright (C) Peter A. Buhr 1999
//
// ClientUNIX2.cc – Client for UNIX/datagram socket test. Client reads from
// standard input, writes the data to the server, reads the data from the
// server, and writes that data to standard output.
//
// Author : Peter A. Buhr
// Created On : Thu Apr 29 16:05:12 1999
// Last Modified By : Peter A. Buhr
// Last Modified On : Mon May 24 12:08:29 1999
// Update Count : 9
//

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

#define EOD '\377'
// minimum buffer size is 2, 1 character and string terminator,'\0'
#define BufferSize (8 * 1024)

int rcnt = 0, wcnt = 0;

uTask reader {
    uSocketClient &client;

    void main() {
        char buf[BufferSize];
        int len;

        for ( ;; ) {
            len = client.recvfrom( buf, sizeof(buf) );
            // uCerr << uAcquire << "Client::reader read len:" << len << endl << uRelease;
          if ( buf[0] == EOD ) break;
            // The EOD character can be piggy-backed onto the end of the message.
          if ( buf[len - 1] == EOD ) { rcnt += len - 1; uCout.write( buf, len - 1 ); break; }
            rcnt += len;
            uCout.write( buf, len );
        } // for
        // connection not closed until after this data received
        buf[0] = '\0';
        client.sendto( buf, sizeof(char) );
    } // reader::main
  public:
    reader( uSocketClient &client ) : client ( client ) {
    } // reader::reader
}; // reader

uTask writer {
    uSocketClient &client;

    void main() {
        char buf[BufferSize];

        for ( ;; ) {
            uCin.get( buf, sizeof(buf), '\0' );          // leave room for string terminator
            int len = strlen( buf );
            // uCerr << uAcquire << "Client::writer read len:" << len << endl << uRelease;
          if ( buf[0] == '\0' ) break;
            wcnt += len;
            client.sendto( buf, len );
        } // for
        buf[0] = EOD;
```

```
        client.sendto( buf, sizeof(char) );
    } // writer::main
  public:
    writer( uSocketClient &client ) : client( client ) {
    } // 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], SOCK_DGRAM );          // connection to server
    {
        reader rd( client );                    // emit worker to read from server and write to output
        writer wr( client );                    // emit worker to read from input and write to server
    }
    if ( wcnt != rcnt ) {
        uAbort( "not all data transfered\n" );
    } // if
} // uMain::main

// Local Variables: //
// tab-width: 4 //
// compile-command: "u++-work -o Client ClientUNIX2.cc" //
// End: //
```

C.6.2 Server - UNIX/Datagram

```
// -*- Mode: C++ -*-
//
// uC++ Version 4.7, Copyright (C) Peter A. Buhr 1999
//
// ServerUNIX2.cc – Server for UNIX/datagram socket test. Server reads data
// from multiple clients. The server reads the data from the client and writes
// it back.
//
// Author : Peter A. Buhr
// Created On : Fri Apr 30 16:36:18 1999
// Last Modified By : Peter A. Buhr
// Last Modified On : Mon May 24 12:16:25 1999
// Update Count : 14
//

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

#define EOD '\377'
#define BufferSize (8 * 1024)

uTask reader {
    uSocketServer &server;

    void main() {
        uDuration timeout( 60, 0 );                   // timeout for read
      char buf[BufferSize];
        int len;

        try {
            for ( ;; ) {
                int len = server.recvfrom( buf, sizeof(buf), 0, &timeout );
                // uCerr << uAcquire << "Server::reader read len:" << len << endl << uRelease;
```

```
                server.sendto( buf, len );                      // write byte back to client
            } // for
        } catch( uSocketServer::uReadTimeout ) {
        } // try
    } // reader::main
  public:
    reader( uSocketServer &server ) : server( server ) {
    } // reader::reader
}; // reader

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

    uSocketServer server( argv[1], SOCK_DGRAM );        // create and bind a server socket
    {
        reader rd( server );                            // execute until EOD
    }
} // uMain

// Local Variables: //
// tab-width: 4 //
// compile-command: "u++-work -o Server ServerUNIX2.cc" //
// End: //
```

C.6.3 Client - INET/Stream

```
// -*- Mode: C++ -*-
//
// uC++ Version 4.7, Copyright (C) Peter A. Buhr and Richard A. Stroobosscher 1994
//
// ClientINET.cc – Client for INET/stream socket test. Client reads from
// standard input, writes the data to the server, reads the data from the
// server, and writes that data to standard output.
//
// Author : Peter A. Buhr
// Created On : Tue Jan 7 08:42:32 1992
// Last Modified By : Peter A. Buhr
// Last Modified On : Mon May 24 12:08:30 1999
// Update Count : 127
//

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

#define EOD '\377'
// minimum buffer size is 2, 1 character and string terminator, '\0'
#define BufferSize (8 * 1024)

int rcnt = 0, wcnt = 0;

uTask reader {
    uSocketClient &client;

    void main() {
        char buf[BufferSize];
        int len;

        for ( ;; ) {
            len = client.read( buf, sizeof(buf) );
            // uCerr << uAcquire << "Client::reader read len:" << len << endl << uRelease;
```

```
            if ( buf[0] == EOD ) break;
                // The EOD character can be piggy-backed onto the end of the message.
            if ( buf[len - 1] == EOD ) { rcnt += len - 1; uCout.write( buf, len - 1 ); break; }
                rcnt += len;
                uCout.write( buf, len );
            } // for
            // connection not closed until after this data received
            buf[0] = '\0';
            client.write( buf, sizeof(char) );
        } // reader::main
      public:
        reader( uSocketClient &client ) : client ( client ) {
        } // reader::reader
};  // reader

uTask writer {
        uSocketClient &client;

        void main() {
            char buf[BufferSize];

            for ( ;; ) {
                uCin.get( buf, sizeof(buf), '\0' );          // leave room for string terminator
                int len = strlen( buf );
                // uCerr << uAcquire << "Client::writer read len:" << len << endl << uRelease;
              if ( buf[0] == '\0' ) break;
                wcnt += len;
                client.write( buf, len );
            } // for
            buf[0] = EOD;
            client.write( buf, sizeof(char) );
        } // writer::main
      public:
        writer( uSocketClient &client ) : client( client ) {
        } // writer::writer
};  // writer

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

        uSocketClient client( atoi( argv[1] ) );                    // connection to local server
        {
            reader rd( client );                        // emit worker to read from server and write to output
            writer wr( client );                        // emit worker to read from input and write to server
        }
        if ( wcnt != rcnt ) {
            uAbort( "not all data transfered\n" );
        } // if
} // uMain::main

// Local Variables: //
// tab-width: 4 //
// compile-command: "u++-work -o Client ClientINET.cc" //
// End: //
```

C.6.4 Socket - INET/Stream

```
// -*- Mode: C++ -*-
//
// uC++ Version 4.7, Copyright (C) Peter A. Buhr and Richard A. Stroobosscher 1994
//
```

```
// ServerINET.cc – Server for INET/stream socket test. Server accepts multiple
// connections from clients. Each client then communicates with an acceptor.
// The acceptor reads the data from the client and writes it back.
//
// Author : Peter A. Buhr
// Created On : Tue Jan 7 08:40:22 1992
// Last Modified By : Peter A. Buhr
// Last Modified On : Mon May 24 12:14:07 1999
// Update Count : 157
//

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

#define EOD '\377'
#define BufferSize (8 * 1024)

uTask server;                                           // forward declaration

uTask acceptor {
    uSocketServer &sockserver;
    server &s;

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

uTask server {
    uSocketServer &sockserver;
    acceptor *terminate;
    int acceptorCnt;
    bool timeout;
  public:
    server( uSocketServer &socks ) : sockserver( socks ), timeout( false ), acceptorCnt( 1 ) {
    } // server::server

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

    void complete( acceptor *terminate, bool timeout ) {
        server::terminate = terminate;
        server::timeout = timeout;
    } // 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
                acceptorCnt += 1;
            } uOr uAccept( complete ) {                 // acceptor has completed with client
                delete terminate;                       // delete must appear here or deadlock
                acceptorCnt -= 1;
          if ( acceptorCnt == 0 ) break;                // if no outstanding connections, stop
                if ( timeout ) {
                    new acceptor( sockserver, *this );  // create new acceptor after a timeout
                    acceptorCnt += 1;
                } // if
            }; // uAccept
        } // for
    } // server::main
}; // server

void acceptor::main() {
```

```
    try {
        uDuration timeout( 60, 0 );                     // timeout for accept
        uSocketAccept acceptor( sockserver, &timeout ); // accept a connection from a client
        char buf[BufferSize];
        int len;

        s.connection();                                 // tell server about client connection
        for ( ;; ) {
            len = acceptor.read( buf, sizeof(buf) );    // read byte from client
            // uCerr << uAcquire << "Server::acceptor read len:" << len << endl << uRelease;
            acceptor.write( buf, len );                 // write byte back to client
            // The EOD character can be piggy-backed onto the end of the message.
          if ( buf[0] == EOD || buf[len - 1] == EOD ) break; // end of data ?
        } // for
        acceptor.read( buf, sizeof(char) );             // end of transmission
        s.complete( this, false );                      // terminate
    } catch( uSocketAccept::uOpenTimeout ) {
        s.complete( this, true );                       // terminate
    } // try
} // acceptor::main

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

    short unsigned int port;
    uSocketServer sockserver( &port );                  // create and bind a server socket to free port

    uCout << port << endl;                              // print out free port for clients
    {
        server s( sockserver );                         // execute until acceptor times out
    }
} // uMain

// Local Variables: //
// tab-width: 4 //
// compile-command: "u++-work -o Server ServerINET.cc" //
// End: //
```

Bibliography

[AGMK94] B. Adelberg, H. Garcia-Molina, and B. Kao. Emulating Soft Real-Time Scheduling Using Traditional Operating System Schedulers. In *Proc. IEEE Real-Time Systems Symposium*, pages 292–298, 1994.

[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.

[BD92] Peter A. Buhr and Glen Ditchfield. Adding Concurrency to a Programming Language. In *USENIX C++ Technical Conference Proceedings*, pages 207–224, Portland, Oregon, U.S.A., August 1992. USENIX Association.

[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.

[BFC95] Peter A. Buhr, Michel Fortier, and Michael H. Coffin. Monitor Classification. *ACM Computing Surveys*, 27(1):63–107, March 1995.

[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.

[BP91] T. Baker and O. Pazy. Real-Time Features of Ada 9X. In *Proc. IEEE Real-Time Systems Symposium*, pages 172–180, 1991.

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

[Buh95] Peter A. Buhr. Are Safe Concurrency Libraries Possible? *Communications of the ACM*, 38(2):117–120, February 1995.

[BW90] Alan Burns and A. J. Wellings. The Notion of Priority in Real-Time Programming Languages. *Computer Language*, 15(3):153–162, 1990.

[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.

[CD95] Tai M. Chung and Hank G. Dietz. Language Constructs and Transformation for Hard Real-time Systems. In *Proc. Second ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Real-Time Systems*, June 1995.

[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.

[Dij65] Edsger W. Dijkstra. Cooperating Sequential Processes. Technical report, Technological University, Eindhoven, Netherlands, 1965. Reprinted in [Gen68] pp. 43–112.

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

[Geh92] N. H. Gehani. Exceptional C or C with Exceptions. *Software—Practice and Experience*, 22(10):827–848, October 1992.

[Gen68] F. Genuys, editor. *Programming Languages*. Academic Press, New York, 1968. NATO Advanced Study Institute, Villard-de-Lans, 1966.

[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.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, Reading, Massachusetts, 1996.

[Gol94] David B. Golub. Operating System Support for Coexistence of Real-Time and Conventional Scheduling. Technical report, Carnegie Mellon University, November 1994.

[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.

[GR91] N. Gehani and K. Ramamritham. Real-Time Concurrent C: A Language for Programming Dynamic Real-Time Systems. *Real-Time Systems*, 3(4):377–405, December 1991.

[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.

[HM92] W.A. Halang and K. Mangold. Real-Time Programming Languages. In Michael Schiebe and Saskia Pferrer, editors, *Real-Time Systems Engineering and Applications*, chapter 4, pages 141–200. Kluwer Academic Publishers, 1992.

[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.

[Int95] International Standard ISO/IEC. *Ada Reference Manual*, 6.0 edition, 1995.

[ITM90] Y. Ishikawa, H. Tokuda, and C.W. Mercer. Object-Oriented Real-Time Language Design: Constructs for Timing Constraints. In *Proc. ECOOP/OOPSLA*, pages 289–298, October 1990.

[KK91] K.B. Kenny and K.J.Lin. Building Flexible Real-Time Systems using the Flex Language. *IEEE Computer*, 24(5):70–78, May 1991.

[KS86] E. Klingerman and A.D. Stoyenko. Real-Time Euclid: A Language for Reliable Real-Time Systems. *IEEE Transactions on Software Engineering*, pages 941–949, September 1986.

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

[LN88] K.J. Lin and S. Natarajan. Expressing and Maintaining Timing Constratins in FLEX. In *Proc. IEEE Real-Time Systems Symposium*, pages 96–105, 1988.

[Mac77] M. Donald MacLaren. Exception Handling in PL/I. *SIGPLAN Notices*, 12(3):101–104, March 1977. Proceedings of an ACM Conference on Language Design for Reliable Software, March 28–30, 1977, Raleigh, North Carolina, U.S.A.

[Mar78] T. Martin. Real-Time Programming Language PEARL – Concept and Characteristics. In *IEEE Computer Society 2nd International Computer Software and Applications Conference*, pages 301–306, 1978.

[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.

[MMPN93] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-oriented Programming in the BETA Programming Language*. Addison-Wesley, 1993.

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

[RAA+88] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Hermann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. Chorus Distributed Operating Systems. *Computing Systems*, 1(4):305–370, 1988.

[Raj91] Ragunathan Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.

[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.

[Rip90] David Ripps. *An Implementaion Guide to Real-Time Programming*. Yourdon Press, 1990.

[RSL88] Ragunathan Rajkumar, Lui Sha, and John P. Lehoczky. Real-Time Synchronization Protocols for Multiprocessors. In *Proc. IEEE Real-Time Systems Symposium*, pages 259–269, 1988.

[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.

[SD92] A.E.K. Sahraoui and D. Delfieu. ZAMAN, A Simple Language for Expressing Timing Constraints. In *Real-Time Programming, IFAC Workshop*, pages 19–24, 1992.

[Sha86] Alan Shaw. Software Clocks, Concurrent Programming, and Slice-Based Scheduling. In *Proc. IEEE Real-Time Systems Symposium*, pages 14–18, 1986.

[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.

[SRL90] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

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

[Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, third edition, 1997.

[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.

[TvRvS+90] Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, Sape J. Mullender, Jack Jansen, and Guido van Rossum. Experiences with the Amoeba Distributed Operating System. *Communications of the ACM*, 33(12):46–63, December 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.

[Yok92] Yasuhiko Yokote. The Apertos Reflective Operating System: The Concept and Its Implementation. In *Proc. Object-Oriented Programming Systems, Languages, and Applications*, pages 414–434, 1992.

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

Index

-U++ option, 11
-compiler option, 11
-cpp option, 11
-debug option, 10
-multi option, 11
-nodebug option, 11
-nomulti option, 11
-noquiet option, 11
-notrace option, 11
-noverify option, 11
-noyield option, 11
-quiet option, 11
-trace option, 11
-verify option, 11, 15
-yield option, 11, 30
<<, 44, 45
>>, 44, 45, 110, 112, 114
__U_CPLUSPLUS_MINOR__, 11
__U_CPLUSPLUS__, 11
__U_DEBUG__, 12
__U_MULTI__, 12
__U_TRACE__, 12
__U_VERIFY__, 12
__U_YIELD__, 12
_exit, 75

abnormal event, **55**
abnormal events
    inherited members, 57
abort, 75
accept-blocked, 21, 22
acceptor, 48
acceptor/signalled stack, 20, 21, 23, 25
access object, **47**
activation point, 13, 28
active, **4**, **18**
active priority, 97, **97**
aperiodic task, 95
argc, 8
argv, 8
assert, 75
asynchronous propagation, **56**, 58

barrier, 35
base priority, **97**
basic priority-inheritance protocol, **96**
blocked, **4**
busy wait, 35

chain blocking, **96**
class, 7
class type, **7**
class-object, **5**, **7**
client, 48
cluster, 8
    I/O, 41
code reuse, 32
compilation
    -U++ option, 11
    -compiler option, 11
    -cpp option, 11
    -debug option, 10
    -multi option, 11
    -nodebug option, 11
    -nomulti option, 11
    -noquiet option, 11
    -notrace option, 11
    -noverify option, 11
    -noyield option, 11
    -quiet option, 11
    -trace option, 11
    -verify option, 11, 15
    -yield option, 11, 30
    u++, 10
concurrency, 8
condition variable, 23
consequent event, **63**
context switch, **4**, 15, 21
coroutine, **5**, **7**, 12
    full, 13
    inherited members, 14
    semi, 13
    termination, 13
coroutine main, 12
coroutine type, **7**
coroutine-monitor, **5**, **7**, 26
coroutine-monitor type, **7**

data structure library, 109
dbx, 103

deadline monotonic, **99**
deadlock, 13, 24, 25, 27, 45, 67, 81, 96, 104
debugging
    symbolic, 103
Dekker, 5
duration, **89**

entry queue, 20, 24
epoch, 90
event propagation, 56, 57
    resuming, 56, 57
    throwing, 56, 57
event trace, 11
event type, **56**
events, 56
    dual, 56
    resume, 56
    throw, 56
exceptional, **55**
execution-state, **4**
    active, 4
    inactive, 4
external variables, 14, 28

finite-state machine, 12, 26
fixed-point registers, 37
floating-point registers, 37
frame, **95**
free, 84
free routine, **5**
full-coroutine, **13**

gdb, 103
gethostbyname, 54
gethostname, 54
GNU C++, 10, 104
guarded block, **59**

handler, 59
    resuming, 60
    terminating, 59
handlers, 56
    resuming, 56
    terminating, 56
heap area, 14, 28
heavy blocking, 88
heavy-weight process, 88

I/O cluster, 41
immediate consequent event, **63**
implementation problems, 39
implicit scheduler, 20
inactive, **4**, **18**
inheritance, 32
    multiple, 33
    private, 32
    protected, 32
    public, 32
    single, 32
inherited members
    abnormal events, 57
    coroutine, 14
    task, 29
initial task
    uMain, 8
internal scheduler, 24
intervention, **55**, 60

kernel thread, 9, 85
keyword, additions
    uAccept, 21, 92
    uAt, 58
    uCoroutine, 12
    uDualEvent, 56
    uElse, 21
    uMutex, 18
    uNoMutex, 12, 18
    uOr, 21, 92
    uPeriodicTask, 94
    uRaise, 58
    uRaiseEvent, 56
    uRealTimeTask, 95
    uResume, 15
    uSignal, 25
    uSignalBlock, 25
    uSporadicTask, 95
    uSuspend, 15
    uTask, 28
    uThrow, 58
    uThrowEvent, 56
    uTimeout, 92
    uWait, 24
    uWhen, 21, 92
    uWith, 24

light blocking, 88
light-weight process, 8
linear propagation, **63**, 64
lock, 35
locking, **18**

main, 8
malloc, 84
manipulators, 47
monitor, **5**, **7**, 26
    active, 18
    inactive, 18
monitor type, **7**

multikernel, 10, 85
mutex member, **5**, 16
mutex queue, 20
mutex type, 16
mutex type state
    locked, 18
    unlocked, 18
mutual exclusion, **5**

nonblocking I/O, 41

object, **7**
out-of-band data, 54
owner lock, 34

parallel execution, 8
parallelism, **8**
periodic task, 94
pre-emption
    default, 88
pre-emptive scheduling, 87
preprocessor variables
    __U_CPLUSPLUS_MINOR__, 11
    __U_CPLUSPLUS__, 11
    __U_DEBUG__, 12
    __U_MULTI__, 12
    __U_TRACE__, 12
    __U_VERIFY__, 12
    __U_YIELD__, 12
prioritized pre-emptive scheduling, **97**
priority, **97**
priority levels, 97
priority-inheritance protocol, **96**
private memory, 84
process
    heavy-weight, 88
    light-weight, 8
    UNIX, 88
push-down automata, 12

ready, **4**
real-time cluster, **99**
recursive resuming, **62**
rendezvous, 22, 67
resuming handler, 60
rethrow, 58
return code, **55**
running, **4**

select, 41
semaphore, 33
semi-coroutine, **13**
server, 48
shared memory, 84
shared-memory model, **8**
socket, 48
    endpoint, 48
spin
    default, 88
sporadic task, 95
stack, 4, 55, 109
    acceptor/signalled, 20, 21, 23, 25
    amount, 15
    automatic growth, 10, 76
    current, 14
    data structure, 109
    default size, 14, 29, 43, 84, 85, 88
    diddling, 31
    free, 15
    minimum size, 83
    overflow, 15, 30, 76
    recursive resuming, 62
    uMain, 88
    used, 15
stale readers, **117**
static storage, 14, 28
subtyping, 32
system cluster, 9, 42

task, **6, 7**, 27
    aperiodic, 95
    inherited members, 29
    periodic, 94
    sporadic, 95
    termination, 28
task main, **28**
task type, **7**
terminating handler, 59
thread, **4**
    blocked, 4
    running, 4
thread-safe, 83
time, **89**
translator, 7
    problems, 39

u++, 10
uAbort, 75
uAccept, 7, 21, 92
uAcquire, 35, 44
uActive, 15
uAdd, 110, 111, 113
uAddHead, 113
uAddTail, 113
uAEHM::uDualClass
    uDefaultResume, 57
    uDefaultTerminate, 57
    uMsg, 57

uSrc, 57
uSrcName, 57
uAssert, 75
uAssert.h, 75
uAt, 58
uBarrier, 35
uBlock, 36
uLast, 36
uReset, 36
uTotal, 36
uWaiters, 36
uBarrier.h, 36
uBaseCoroutine, 14, 30, 93
uFailure, 14
uGetName, 14
uGetState, 14
uSetName, 14
uStackFree, 14
uStackOverflow, 14
uStackPointer, 14
uStackSize, 14
uStackUsed, 14
uTerminated, 14
uVerify, 14
uBaseCoroutine::uActive, 15
uBaseCoroutine::uHalt, 15
uBaseCoroutine::uInActive, 15
uBaseSchedule
uAdd, 98
uAddInitialize, 98
uCheckPriority, 98
uEmpty, 98
uGetActivePriority, 97
uGetBasePriority, 97
uGetInheritTask, 97
uPop, 98
uRemoveInitialize, 98
uRescheduleTask, 98
uResetPriority, 98
uSetActivePriority, 97
uSetBasePriority, 97
uTaskCategory, 97
uBaseTask, 29
uGetActivePriority, 29
uGetBasePriority, 29
uGetCluster, 29
uGetState, 29
uMigrate, 29
uYield, 29
uBaseTask::uBlocked, 30
uBaseTask::uReady, 30
uBaseTask::uRunning, 30
uBeginUserCode, 83
uBlock, 36
uBlocked, 30
μC++ translator, 7
uC++.h, 10, 26, 27, 109
uCerr, 44
uCin, 44
μC++ kernel, 10, 83
uClock
uGetTime, 93
uResetClock, 93
uCluster, 84
uGetName, 84
uGetStackSize, 84
uSetName, 84
uSetStackSize, 84
uColable, 109
uListed, 109
uCoMonitor, 27
uCondition, 24
owner, 24
uEmpty, 24
uFront, 24
uSuspendedFailure, 24
uContext, 37
uCoroutine, 12
uCoroutine, 7
uCout, 44
uDefaultPreemption, 88
uDefaultSpin, 88
uDefaultStackSize, 88
uDrop, 110, 111, 113
uDropHead, 113
uDropTail, 113
uDualEvent, 14, 56
uDuration, 89
uElse, 21
uEmpty, 24, 34, 110, 111, 113
uEndUserCode, 83
uExit, 75
uFile, 47
status, 47
uGetName, 47
uFile.h, 47
uFileAccess, 47
fsync, 48
lseek, 48
read, 48
write, 48
uFloatingPointContext, 37
uFront, 24
uFStream.h, 46
uGetActivePriority, 30
uGetBasePriority, 30

uGetClock, 86
uGetCluster, 30, 86
uGethostbyname, 54
uGethostname, 54
uGetName, 15, 47, 85
uGetPid, 86
uGetPreemption, 86
uGetSpin, 86
uGetStackSize, 85
uGetState, 15, 30
uGetTask, 86
uGetTime, 93
uHalt, 15
uHead, 110, 111, 113
uIdle, 86
uIFStream, 46
    attach, 46
    close, 46
    open, 46
    rdbuf, 46
uInActive, 15
uInsertAft, 113
uInsertBef, 113
uIOCluster, 43
    uExceptSelect, 43
    uReadSelect, 43
    uSelect, 43
    uWriteSelect, 43
uIOManip.h, 47
uIOStream.h, 45
uIStream, 44
uIStream.h, 45
uLock, 35
    uAcquire, 35
    uRelease, 35
    uTryAcquire, 35
uMain, 8
    argc, 8
    argv, 8
    uRetCode, 8
uMain::main, 8
uMainStackSize, 88
uMigrate, 30
uMonitor, 26
uMutex, 7, 18
uMutex class, 26
    uMonitor, 26
uMutex uCoroutine, 27
    uCoMonitor, 27
unikernel, 10, 85
UNIX epoch, 90
UNIX process, 85
unlocking, **18**
uNoMutex, 7, 12, 18
uOFStream, 46
    attach, 46
    close, 46
    open, 46
    rdbuf, 46
uOr, 21, 92
uOStream, 44
uOStream.h, 45
uOver, 110, 112, 114
uOwnerLock, 34
    uAcquire, 35
    uRelease, 35
    uTryAcquire, 35
uOwnerLock.h, 35
uP, 34
uPeriodicBaseTask
    uGetPeriod, 94
    uSetPeriod, 94
uPeriodicTask, 94
uPop, 110
uPred, 113
uProcessor, 86
    uGetClock, 86
    uGetCluster, 86
    uGetPid, 86
    uGetPreemption, 86
    uGetSpin, 86
    uGetTask, 86
    uIdle, 86
    uMmap, 86
    uSetCluster, 86
    uSetPreemption, 86
    uSetSpin, 86
uPush, 110
uQueue
    uAdd, 111
    uDrop, 111
    uEmpty, 111
    uHead, 111
    uTail, 111
uQueueGen, 111
    >>, 112
    uOver, 112
uRaise, 58
uRaiseEvent, 56
uReady, 30
uRealTimeBaseTask
    uGetDeadline, 96
    uSetDeadline, 96
uRealTimeTask, 95
uRelease, 35, 44
uRemove, 113

uRendezvousAcceptor, 67
uReset, 36
uResetClock, 93
uResume, 7, 15
uRetCode, 8
uRunning, 30
uSelect, 43
uSemaphore, 33
    uEmpty, 34
    uP, 34
    uV, 34
uSemaphore.h, 34
uSeqable, 109
    uListed, 109
uSeqGen, 113
    >>, 113
    uOver, 113
uSeqGenRev, 113
    >>, 114
    uOver, 114
uSequence
    uAdd, 113
    uAddHead, 113
    uAddTail, 113
    uDrop, 113
    uDropHead, 113
    uDropTail, 113
    uEmpty, 113
    uHead, 113
    uInsertAft, 113
    uInsertBef, 113
    uPred, 113
    uRemove, 113
    uSucc, 113
    uTail, 113
user cluster, 9
uSetCluster, 86
uSetName, 15, 85
uSetPreemption, 86
uSetSpin, 86
uSetStackSize, 85
uSignal, 7, 25
uSignalBlock, 25
uSocket.h, 49
uSocketAccept, 53
    getpeername, 53
    getsockname, 53
    read, 53
    readv, 53
    recv, 53
    recvfrom, 53
    recvmsg, 53
    send, 53
    sendmsg, 53
    sendto, 53
    write, 53
    writev, 53
uSocketAcceptor, 92
uSocketClient, 49, 92
    getpeername, 50
    getsockname, 50
    read, 50
    readv, 50
    recv, 50
    recvfrom, 50
    recvmsg, 50
    send, 50
    sendmsg, 50
    sendto, 50
    write, 50
    writev, 50
uSocketServer, 51
    getpeername, 52
    getsockname, 52
    read, 52
    readv, 52
    recv, 52
    recvfrom, 52
    recvmsg, 52
    send, 52
    sendmsg, 52
    sendto, 52
    write, 52
    writev, 52
uSporadicBaseTask
    uGetFrame, 95
    uSetFrame, 95
uSporadicTask, 95
uStack, 109
    uAdd, 110
    uDrop, 110
    uEmpty, 110
    uHead, 110
    uPop, 110
    uPush, 110
    uTop, 110
uStackFree, 15
uStackGen, 110
    >>, 110
    uOver, 110
uStackPointer, 14
uStackSize, 15
uStackUsed, 15
uSucc, 113
uSuspend, 7, 15
uTail, 111, 113

uTask, 7, 28
uThisCluster, 85
uThisCoroutine, 15
uThisProcessor, 86
uThisTask, 30
uThrow, 58
uThrowEvent, 56
uTime, 90
uTimeout, 92
uTop, 110
uTotal, 36
uTryAcquire, 35
uV, 34
uVerify, 15
uWait, 7, 24
uWaiters, 36
uWhen, 21, 92
uWith, 24
uYield, 30

version number, 11
virtual processor, 9, 85, 88

yield, 30, 77, 85
    compilation option, 11, 30
    preprocessor, 12