μSystem Annotated Reference Manual

Version 4.4.1

Peter A. Buhr, Hamish I. Macdonald,

and Richard A. Stroobosscher ©1991

October 14, 1991

Contents

Preface

The goal of this work was to introduce concurrency into the language C. To achieve this goal a set of library routines and definitions were created in C. The set of routines and definitions 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 routines and definitions are considered important.

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

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

□ Annotation discussion like this is quoted with quads. □

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

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

Abridged Manual

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

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

Chapter 1

μSystem Structure

The μSystem [BS90] is a library of C [KR88] routines and definitions that provide light-weight concurrency on uniprocessor and multiprocessor computers running the UNIX[1] operating system. Operations in the μSystem are expressed at a high-level, that is, the abstractions provided are used to structure an algorithm into a set of objects that interact, possibly in parallel, to complete a computation. This is in contrast to low-level schemes that attempt to *discover* concurrency in a sequential program, for example, by parallelizing loops and access to data structures. While the μSystem does not provide automated parallelization, neither does its design preclude it. We believe that low-level schemes are limited in their capacity to *discover* parallelism, and therefore, we provide a sufficient set of constructs to allow arbitrarily complex algorithms to be expressed in a direct manner.

The μSystem uses a **single-memory model**. This single-memory is populated by routine-activations, coroutines, monitors and concurrently executing light-weight tasks all of which have a uniform addressing scheme for accessing each other. This memory may be the address space of a single UNIX process or a memory shared among a set of UNIX processes. Because all tasks use the same memory, there is a low execution cost for creating and maintaining the tasks and communicating among them; consequently, the tasks are **light-weight**. This has its advantages as well as its disadvantages. Tasks need not communicate by sending large data structures back and forth, but can simply pass pointers to data structures. However, there is no address space protection among tasks so one faulty task may overwrite another task's data area.

The μSystem does not call the UNIX kernel to perform a coroutine or task context switch or to schedule tasks, and uses shared memory for communication among tasks. As a result, performance for execution of and communication among large numbers of tasks is significantly increased over UNIX processes (e.g. two orders of magnitude in some cases). The maximum number of tasks that can be active is restricted only by the amount of memory available in a program.

1.1 Compile Time Structure of a μSystem Program

A μSystem program is constructed exactly like a normal C program with one exception: the main (starting) routine is called `uMain` instead of the normal C name, `main`, for example:

```
... normal C declarations and routines

void uMain( int argc, char *argv[], char *envp[] ) {
    ...
}
```

The μSystem supplies and uses the `main` routine to initialize the μSystem runtime environment and create the first task which starts execution at `uMain`. The task `uMain` is passed the same three arguments that are passed to routine `main`: `argc`, `argv`, and `envp`.

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

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

1.2 Runtime Structure of a μSystem Program

The dynamic structure of an executing μSystem program is significantly more complex than a normal C program. There are four new runtime entities: coroutine, task, virtual processor, and cluster.

1.2.1 Coroutine

A coroutine is a program component whose execution can be suspended and resumed (see [Mar80] for a complete discussion of coroutines). Execution of a coroutine is suspended as control leaves it, only to carry on where it left off when control re-enters the coroutine at some later time. This means that coroutines are not entered from the beginning on each activation. In contrast, when a subroutine is invoked, it always starts execution from the beginning and its local variables only persist for that particular invocation. The state of a coroutine consists of:

- a current location which is initialized to a starting point and then traverses whatever part of the program that is reachable through the normal control-flow facilities.
- an execution state – blocked or active or terminated – which is changed by the coroutine constructs of the μSystem.
- a memory which holds the data items created by the code the coroutine is executing. This is the stack that contains the local variables for the coroutine and any subroutines called by the coroutine. This stack is the mechanism by which the local variables persist between successive activations of the coroutine.

As well, a coroutine identifier exists to reference the coroutine.

A coroutine executes synchronously with other coroutines created by the same task, and hence there is no concurrency among coroutines associated with a particular task. (Although, multiple instances of the same coroutine could be executing concurrently in different tasks.) While coroutines have no concurrency, they are valuable constructs in a programming language. A coroutine properly handles the class of problems that require state information to be retained between successive calls (e.g. finite state problems). Solutions to such problems without coroutines require variables with external visibility, or local visibility and static storage class. But since these variables are only allocated once, only one instance of such routines can be active. Because each coroutine has its own data area, multiple instances of the same coroutine can be active. Further, this class of problems illustrates the forms of control flow that are present in concurrent programs without the added complexity and expense of dealing with concurrent execution. Hence, coroutines are an intermediate step between subroutines and concurrent tasks, and valuable as a teaching device.

A μSystem coroutine is a C routine that is "cocalled". It can call or cocall any other C routine. A coroutine can interact with other coroutines by executing communication routines from the C routine started as the coroutine or from any of the routines it has called. The μSystem also provides an exception handling mechanism that allows coroutines to handle exceptional conditions raised by the μKernel (see Section 1.3), UNIX operations, and the user application.

1.2.2 Task

A task is a program component with its own thread of control and has the same state information as a coroutine plus a task identifier. A task's thread of control is scheduled separately and independently from threads associated with other tasks. It is this thread of control that results in concurrent execution. On a multiprocessor computer, task execution is performed in parallel. On a uniprocessor computer, concurrency is achieved by interleaving of task execution to give the appearance of parallel execution. Because there may be more tasks to execute than processors to execute them, it is possible for a task to be ready to execute but not executing. Hence, tasks have one more execution state over a coroutine, the ready state.

Tasks are light-weight because of the low execution time cost and space overhead for creating a task and the many forms of communication which are easily and efficiently implemented for them. This is possible as all tasks in the μSystem execute within a single shared memory. This memory may be the address space of a single UNIX process or a memory shared between a set of UNIX processes. This has its advantages as well as its disadvantages. Tasks need not communicate by sending large data structures back and forth, but can simply pass pointers to data structures. However, there is no address space protection between tasks so one faulty task may overwrite another task's data area.

A μSystem task is a C routine that is "emitted". A task is composed of number of communicating coroutines. In theory, a C routine can be called, cocalled and emitted; in practise, the forms of communication used by a routine dictate how it must be started. When a C routine is emitted, a new thread of control is created and begins execution by cocalling the C routine; hence, the emitted task is also a coroutine. The coroutines created by a task's thread belong to that task and cannot communicate with coroutines from other tasks. This restriction follows naturally from the fact that only one thread can be using a coroutine's state, and in the μSystem, that thread is the one associated with the task that created it.

While a task that creates another task is conceptually the parent and the created task its child, the μSystem makes no implicit use of this relationship nor does it provide any facilities that perform actions based on this relationship. Once a task is emitted it has no special relationship with its emitter.

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

1.2.3 Virtual Processor

A μSystem virtual processor is a "software processor" that executes threads. A virtual processor is implemented as a UNIX process (or kernel thread) that is subsequently scheduled for execution on the actual processor(s) by the underlying operating system. On a multiprocessor, UNIX processes are usually distributed across the hardware processors and so some UNIX processes are able to execute in parallel. This, in turn, means the tasks executing on them execute in parallel. The μSystem uses virtual processors instead of actual 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 actual processors. Thus, the way in which the μSystem accesses the concurrency of the underlying hardware is through an intermediate resource, the UNIX process (or kernel thread). In this way, the μSystem is kept portable across uniprocessor and different multiprocessor hardware designs.

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

In the μSystem, virtual processors are manipulated indirectly through a cluster.

1.2.4 Cluster

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

A cluster uses a single-queue multi-server queueing model for scheduling its tasks on its virtual processors. This results in automatic load balancing of tasks on virtual processors. Figure 1.1 illustrates the runtime structure of a μSystem program. An executing task is illustrated by its containment in a virtual processor. Because of appropriate defaults for virtual processors and clusters, it is possible to begin writing μSystem programs after learning about coroutines or tasks. More complex concurrency work may require the use of virtual processors and clusters. If several clusters exist, tasks can be explicitly migrated from one cluster to another. No automatic load balancing among clusters is performed by μSystem.

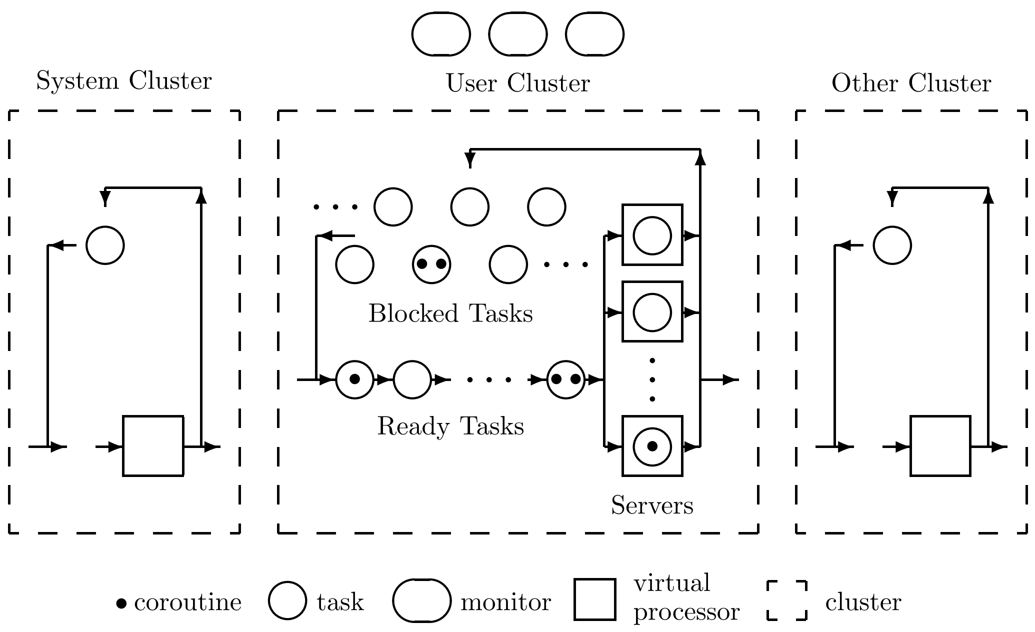

Figure 1.1: Runtime Structure of a μSystem Program

When a μSystem program begins execution, it creates two clusters: a system cluster and a user cluster. The system cluster contains a virtual processor which does not execute user tasks. Instead, the system cluster catches errors that occur on the user clusters, prints appropriate error information and shuts down μSystem. A user cluster is created to contain the user tasks; an implicit first task is created in the user cluster that begins executing the routine `main`. Having all tasks execute on the one cluster often maximizes utilization of virtual processors, which minimizes execution time. 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 concurrency when used indiscriminately. However, in some situations it will be shown that partitioning is essential. For example, on some systems concurrent UNIX I/O operations are only possible by exploiting the clustering mechanism.

1.3 μKernel

After a μSystem program has been compiled, a runtime concurrency library is linked in with the resulting program, called the μKernel. There are two versions of the μKernel: the unikernel, which is designed to use a single virtual processor (the system, user and any other clusters are automatically combined); and the multikernel, which is designed to use several virtual processors. Thus, the unikernel is sensibly used on systems with a single hardware processor or nonshared memory; the multikernel is sensibly used on systems that have multiple hardware processors and that permit memory to be shared among UNIX processes. Table 1.1 shows the situations where each kernel can be used. The unikernel can be used in a system with multiple hardware processors and shared memory but does not take advantage of either of these capabilities. The multikernel can be used on a system with a single hardware processor and shared memory but performs less efficiently than the unikernel because it uses multi-processor mutual exclusion unnecessarily.

	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 1.1: When to Use the Unikernel and Multikernel

While the interfaces to the unikernel and multikernel are identical, there are several differences between them, which all result from the unikernel having only one virtual processor. In particular, the semantics of the cluster routines are different for each kernel. In the unikernel, operations to increase or decrease the number of virtual processors are ignored, creation of a new cluster simply returns the current cluster, and destroying a cluster is ignored as there is only one. Hence, the system and user clusters are combined into a single cluster. The uniform interface allows almost all concurrent applications to be designed and tested on the unikernel, and then run on the multikernel after re-linking.

The μKernel provides no support for automatic growth of stack space for coroutines and tasks because this would require compiler support. The μKernel has a debugging form which performs a number of runtime checks, one of which is to check for stack overflow whenever flow of control transfers between coroutines and between tasks. This catches most stack overflows; however, stack overflow can still occur if insufficient stack area is provided, which can cause an immediate error or unexplainable results.

Chapter 2

Using the μSystem

2.1 Include Files

To use the μSystem in a C program, include the file:

```
#include <uSystem.h>
```

at the beginning of each source file. This file also includes the following system files: `<stdio.h>`, `<sys/file.h>`, `<sys/types.h>`. These files are included to provide access to UNIX I/O, signal, and timing facilities.

2.2 Compiling μSystem Programs

The command `concc` is used to compile μSystem program(s). This command works just like the UNIX `cc` command to compile C programs (the actual C compiler used is GNU C [Sta89]), for example:

`concc` *[C options]* *yourprogram.c* *[assembler and loader files]*

The following additional options are available on the `concc` command:

`-debug` The user's program is loaded with the debug version of the unikernel or multikernel. The debug version performs runtime checks to help during the debugging phase of a μSystem program. This slows the execution of the program. The runtime checks should only be removed after the program is completely debugged. **This is the default.**

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

`-multi` The user's program is loaded with the multikernel.

`-nomulti` The user's program is loaded with the unikernel. **This is the default.**

`-profile` The user's program is loaded with a version of either the unikernel or multikernel that has been compiled with the `-p` flag. It also compiles the user's program with the `-p` option. This allows a complete profile of both the user and the μKernel code. Further, this option is not available with the `-debug` option as debugging information would bias the profile.

`-noprofile` The user's program is loaded with a version of the either the unikernel or multikernel that has *not* been compiled with the `-p` flag. **This is the default.**

`-unixrc` The user's program is loaded with a version of UNIX I/O cover routines that return standard UNIX return codes. **This is the default.**

`-nounixrc` The user's program is loaded with a version of UNIX I/O cover routines that do *not* return standard UNIX return codes; instead, the UNIX I/O cover routines raise exceptions.

`-quiet` This suppresses printing of the μSystem compilation message at the beginning of a compilation.

`-noquiet` This prints the μSystem compilation message at the beginning of a compilation. **This is the default.**

`-compiler` *name* This specifies the name of the compiler used to compile the μSystem program(s). This allows compilers other than the default GNU C compiler to be used to compile a μSystem program using `concc`.

The `concc` command is available by including `/u3/usystem/bin` in your command search path, which is usually located in the `.cshrc` file.

2.3 Preprocessor Variables

When compiling a μSystem program, the following preprocessor variables are available during the C preprocessing phase of compilation. The preprocessor variable `__U_SYSTEM__` is always available during preprocessing. If the `-debug` compilation option is specified, then the preprocessor variable `__U_DEBUG__` is available during preprocessing. If the `-multi` compilation option is specified, then the preprocessor variable `__U_MULTI__` is available during preprocessing. If the `-profile` compilation option is specified, then the preprocessor variable `__U_PROFILE__` is available during preprocessing. If the `-unixrc` compilation option is specified, then the preprocessor variable `__U_UNIXRC__` is available during preprocessing. This allows conditional compilation of programs that must work differently in these situations.

2.4 Using the μSystem with C++

The μSystem can be used with both AT&T C++ and GNU C++ (and possibly with other C++ compilers) by doing the following. First, use the `-compiler` option to specify the appropriate compiler in a compilation command, as in:

```
% concc -compiler CC myprogram.cc
```

In this case, the AT&T C++ compiler is named `CC`. With both these C++ compilers, it is essential that the `main` routine be compiled by the specified compiler. This is because "magic" statements are inserted in `main` to cause the constructors and/or destructor for objects declared in the external area to be invoked. The μSystem normally supplies a precompiled `main` routine (compiled using GNU C) in the μSystem library. To override the C compiled `main` code, place the macro `U_MAIN` in your source code, as in:

```
U_MAIN;
```

The macro `U_MAIN` expands into the same code as the precompiled `main` routine, but is now compiled by the C++ compiler. While the `U_MAIN` macro can be placed anywhere, it can only appear once (there can only be one `main` routine), and therefore, it is probably best to place it before or after routine `uMain`. Finally, it is important *not* to perform any μSystem operations in the constructors and destructors of objects declared in the external area because the μSystem has not been initialized until `uMain` is called and is not active after `uMain` terminates.

When a version of C++ greater than AT&T version 1.2.1 is used, it is necessary to specify that routine `uMain` is a C routine and not a C++ routine, as in:

```
extern "C" void uMain( int argc, char *argv[] ) { ... }
```

The μSystem include file is automatically defined to contain C declarations and not C++ declarations.

Finally, when starting a member routine of a class running as a task, it is necessary to explicitly pass a class instance as the first argument to match with the implicit `this` parameter of the member routine, as in:

```
class foo {
  public:
    void bar( int i ) { ... }
}

foo f;
uTask t;

t = uEmit( foo::bar, &f, 3 );   // f is the implicit "this" parameter
```

2.5 General Information

2.5.1 Context Switching

A context switch occurs when control transfers from a coroutine or task to another coroutine or task. The switch involves saving the state of the currently executing party and restoring the state of the other party. In most systems, the entire state of the virtual processor is saved because there is no way to determine if a coroutine or task is using only a subset of the virtual processor state. All coroutines and tasks use the fixed-point registers, while only some use the floating-point registers. Hence, the fixed-point registers are always saved during a context switch, but it may or may not be necessary to save the floating-point registers. Because there is a significant execution cost in saving and restoring the floating-point registers, we have decided not to automatically save them. However, a mechanism is provided to request that the float-point registers be saved as part of a context switch.

If a coroutine or task performs floating-point operations, then it must invoke the routine `uSaveFloat` immediately after starting execution. From that point on, both the fixed-point and floating-point registers are saved during a context switch. It is possible to revert back to saving just the fixed-point registers by invoking the routine `uSaveFixed`. However, in general, switching between saving fixed and floating registers in the same task is likely a dangerous programming practise. It is too easy to accidently put a floating point operation outside the range where the floating-point registers are saved.

2.5.2 Message Passing

Except for arguments passed at creation, communication of information between coroutines and tasks is done by message passing. A message is a block of untyped bytes that is copied, as is, between communicating parties. It is not possible to use arguments to communicate with a coroutine or task after they are invoked because a C routine has only one entry point; therefore, it is necessary to call out of the routine passing and receiving untyped messages. The message is specified by a pointer to the block of bytes and the length of the block. The message receiver must provide the address of an area that is large enough to contain the message that is sent or the communication fails and an exception is raised. The length of the message can be less than the length of the receiving area, but then the message must contain information on the actual length of the message. The message and the receiving area should be specified as data items of the same or structurally equivalent types. Unless the message type is an array or a pointer, the message data-item must be preceded by an `&`. The length of the message and the receiving area should be specified with `sizeof(`*data-item*`)`, unless the data item varies in size. There is essentially no limit on the size of a message (on some machines the implementation limits the length to 64K bytes). If no message is to be sent during a communication, the message pointer must be set to `NULL` and its length set to zero.

□ It is not possible to use parameter-argument passing to communicate with a coroutine or task after it is created because a C routine has only one entry point; therefore, it is necessary to call into the kernel, passing and receiving untyped messages, to get data back to another coroutine or task. □

2.6 Coroutine

A coroutine is a program component whose execution can be suspended and resumed (see [Mar80] for a complete discussion of coroutines). Execution of a coroutine is suspended as control leaves it, only to carry on from that point when control re-enters the coroutine at some later time. This means that a coroutine is not restarted at the beginning on each activation and that the local variables of the currently active functions invoked by the coroutine are preserved. In contrast, a routine always runs to completion before returning so that its local variables only persist for a particular invocation. 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. A coroutine executes synchronously, and hence, there is no concurrency implied by the coroutine construct.

Like a subroutine, a coroutine can access all the external variables of a C program and the heap area. Also, any `static` variables declared within the definition of a coroutine are shared among all instances of that coroutine.

Two slightly different mechanisms are provided for passing control between coroutines. The first permits a coroutine to restart its invoker; the second permits it to restart an arbitrary coroutine. These two mechanisms accommodate two somewhat different styles of coroutine usage: a semi-coroutine, which acts much like a subroutine by always restarting its invoker, and a full coroutine, which acts somewhat like a task by restarting some other coroutine.

A coroutine is associated with the task that created it. If another task attempts to restart a coroutine that it did not create, an exception is raised. Since coroutines determine flow of control within a task, their execution is performed by one of the virtual processors associated with the cluster on which the task is executing.

> □ Simulating a coroutine with a subroutine requires retaining data in variables with global scope or automatic variables with `static` storage-class between invocations. However, retaining state in this ways violates abstraction and does not generalize to multiple instances, since there is only one copy of the storage. Simulating a coroutine with a task, which can retain state between invocations, is inefficient 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 a task, unless the communication is dominated by transferring data. □

2.6.1 Coroutine Type

`uCoroutine` is the type of a coroutine identifier, as in:

```
uCoroutine x, y, z;
```

which creates three variables that contain coroutine-identifier values.

2.6.2 Coroutine Creation

The routine `uLongCocall` starts a C routine running as a coroutine.

```
uCoroutine uLongCocall( void * reply-area, int reply-area-length, long stack-size,
                        void (* routine)(), long argument-length, ... );
```

return-value is the coroutine identifier of the newly created coroutine and must be retained to subsequently communicate with the coroutine. No coroutine will ever have the identifier value `NULL`.

reply-area is the address of the reply area into which the reply message from the first suspend or resume of the newly created coroutine will be copied.

reply-area-length is the size in bytes of the reply area.

stack-size is the size in bytes of the stack that will be allocated for the coroutine.

routine is the name of a C routine to be called as a coroutine. The routine cannot return a value (i.e. it must have return type `void`) but may have any parameters allowed by C.

argument-length is the number of bytes that will be copied as arguments to the coroutine. This number should be the sum of the $max(sizeof(int), sizeof(argument_i))$ for all arguments passed to `routine`.

... are any number of arguments passed as-is to *routine*. Because all arguments in C are passed by value, it is necessary to pass an argument's address if the argument is to be modified (e.g. `&`*argument*).

The cocaller suspends its execution at the `uLongCocall` and the coroutine begins execution just as if the C routine is called directly. The difference is that the coroutine is executing on its own stack.

The following example creates a new coroutine with a stack size of 8000 bytes, starting execution in routine `f` with an argument length set to the size of two floating point values and passing two floating point arguments:

```
uCoroutine corid;
float a, b, reply;
void f(float x, float y) { ... }     /* routine to be cocalled */
...
corid = uLongCocall( &reply, sizeof(reply), 8000, f, sizeof(a) + sizeof(b), a, b );
```

Because users rarely want to bother specifying explicit stack sizes and argument lengths, there exists a short form of the `uLongCocall` routine. `uCocall` performs the same function as `uLongCocall` using a default stack size and argument length. The following example starts `f` running as a coroutine using `uCocall`.

```
corid = uCocall( &reply, sizeof(reply), f, a, b );
```

The default values start at machine dependent values, which are no less than 4000 bytes for the stack size and 64 bytes for the argument length. Changing the defaults is discussed in chapter 3

The routine `uThisCoroutine` is used to determine the identifier of the current coroutine.

```
uCoroutine uThisCoroutine( void );
```

return-value is the coroutine identifier of the calling coroutine.

2.6.3 Coroutine Communication

The `uResume` and `uSuspend` routines are used to transfer control and communicate among coroutines.

The routine `uResume` suspends execution of the current coroutine and restarts execution of a specifically named coroutine.

```
void uResume( uCoroutine coroutine-id, void * reply-area, int reply-area-length,
              void * send-message, int send-message-length );
```

coroutine-id is an identifier to a coroutine that is be restarted, passing it a particular message.

reply-area is the address of a reply area into which the reply message of a suspending coroutine will be copied.

reply-area-length is the size in bytes of the reply area.

send-message is the address of the message to be sent to the restarted coroutine.

send-message-length is the size in bytes of the message to be sent.

The `uResume` operation establishes an implicit link from the restarted coroutine back to the resumer. This link is used by the `uSuspend` operation to perform an implicit restart.

The routine `uSuspend` suspends execution of the current coroutine and restarts execution in the cocaller/resumer.

```
void uSuspend( void * reply-area, int reply-area-length,
               void * send-message, int send-message-length );
```

reply-area is the address of a reply area into which the reply message of a suspending coroutine will be copied.

reply-area-length is the size in bytes of the reply area.

send-message is the address of the message to be sent to the restarted coroutine.

send-message-length is the size in bytes of the message to be sent.

Routine call `uSuspend(...)` is essentially equivalent to `uResume(`*cocaller-id/resumer-id*`, ...)` except that the suspender's implicit link back to its restarter is set to `NULL`. Therefore, it is not possible to establish suspend-suspend cycles between coroutines.

2.6.4 Coroutine Termination

A coroutine is terminated by calling either the `uResumeDie` or the `uSuspendDie` routine. These routines can be invoked at any level of nested subroutine invocation to terminate the coroutine.

The routine `uResumeDie` terminates execution of the current coroutine and restarts execution of a specifically named coroutine.

```
void uResumeDie( uCoroutine coroutine-id, void * send-message, int send-message-length );
```

coroutine-id is an identifier to a coroutine that is be restarted, passing it a particular message.

send-message is the address of a message to be sent to the restarted coroutine.

send-message-length is the size in bytes of the message to be sent.

The `uResumeDie` operation does *not* establishes an implicit link from the restarted coroutine back to the terminating coroutine.

The routine `uSuspendDie` terminates execution of the current coroutine and restarts execution of the last cocaller/resumer.

```
void uSuspendDie( void * send-message, int send-message-length );
```

send-message is the address of a message to be sent to the restarted coroutine.

send-message-length is the size in bytes of the message to be sent.

Routine call `uSuspendDie(...)` is equivalent to `uResumeDie(`*cocaller-id/resumer-id*`, ...)`.

Executing a `return` statement in a cocalled routine is the same as the routine call `uSuspendDie(NULL, 0)`. This restarts the last cocaller/resumer and returns no message. The same action occurs if control runs off the end of the cocalled routine. Therefore, if a value is to be returned at coroutine termination, it must be passed back using one of `uSuspendDie` or `uResumeDie`.

The following example shows the simple case of a coroutine being used as a function.

```
void f(float x, float y) {
    float result;
    ...
    uSuspendDie(&result, sizeof(result));   /* return function result */
}

void uMain() {
    float result, a, b;
    uCoroutine corid;
    ...
    corid = uCocall(&result, sizeof(result), f, a, b);
    ...
}
```

Appendix A.1 contains a complete coroutine program.

2.7 Task

Like a coroutine, a task can access all the external variables of a C program and the heap area. However, because tasks execute concurrently, there is the general problem of concurrent access to such shared variables. Further, this may also happen with `static` variables within a task that is instantiated multiple times. The same problem can occur if a coroutine makes global references or has `static` variables and is instantiated by different tasks. Therefore, it is suggested that these kinds of references be used with extreme caution. The μSystem provides routines to safely communicate information between tasks and allow safe access to the heap.

2.7.1 Task Type

`uTask` is the type of a task identifier, as in:

```
uTask x, y, z;
```

which creates three variables that contain task identifier values.

2.7.2 Task Creation

The routine `uLongEmit` starts a C routine running asynchronously with the calling task.

```
uTask uLongEmit( uCluster cluster, long stack-size, void (* routine)(),
                 long argument-length, ... );
```

return-value is the task identifier of the newly created task and must be retained to subsequently communicate with the task. No task will ever have the identifier value `NULL`.

cluster is a `uCluster` identifier that this task is associated with. (Creating clusters is discussed in chapter 3 Most tasks are created on the current cluster, which is given by calling routine `uThisCluster()`.)

stack-size is the size in bytes of the stack allocated for the task.

routine is the name of a C routine to be executed asynchronously. The routine cannot return a value (i.e. it must have return type `void`), but may have any parameters allowed by C.

argument-length is the number of bytes copied as arguments to the task. This number should be the sum of the $max(sizeof(int), sizeof(argument_i))$ for all arguments passed to `routine`.

... Any number of arguments passed as-is to *routine*. Because all arguments in C are passed by value, it is necessary to pass an argument's address if the argument is to be modified (e.g. `&`*argument*).

The following example creates a new task executing on the current cluster with a stack size of 8000 bytes, starting execution in routine `f` with an argument length set to the size of two floating point values and passing two floating point arguments:

```
uTask tid;
float a, b;
void f( float x, float y ) { ... }     /* routine to be emitted */
...
tid = uLongEmit( uThisCluster(), 8000, f, sizeof(a) + sizeof(b), a, b );
```

There is a short form of `uLongEmit`, called `uEmit`, that assumes the current cluster, a default stack size, and a default argument length. The following example starts `f` running as a task using `uEmit`:

```
tid = uEmit( f, a, b );
```

The default values for stack and argument length are the same as for coroutines.

The routine `uThisTask` is used to determine the identifier of the current task.

```
uTask uThisTask( void );
```

return-value is the task identifier of the calling task.

2.7.3 Task Synchronization and Communication

2.7.3.1 Counting Semaphore

Semaphores are a mechanism for synchronizing the execution of tasks. The semaphores implemented in the μSystem are counting semaphores as described by Dijkstra [Dij68]. A counting semaphore has two parts: a counter and a list of waiting tasks. The counter is accessible to users, while the list of waiting tasks is managed by the μKernel.

`uSemaphore` is the type of a semaphore and it must be initialized before it is used; appropriate count values are integer values ≥ 0. To initialize a semaphore variable, the macro `U_SEMAPHORE` is used, as in:

```
uSemaphore x = U_SEMAPHORE(0), y = U_SEMAPHORE(1), z = U_SEMAPHORE(4);
```

This declares three variables that are semaphores and initializes them to the value 0, 1, and 4, respectively. The macro can be used at execution time to initialize a declared semaphore or initialize a dynamically allocated one, as in:

```
uSemaphore x = U_SEMAPHORE(0), y, *z;
...
y  = U_SEMAPHORE(1);
z  = uMalloc( sizeof(uSemaphore) );
*z = U_SEMAPHORE(4);
```

(The routine `uMalloc` is discussed in chapter 4 `uMalloc` returns `void *` and so it is unnecessary to cast its result to `uSemaphore *` before assigning to `z`.) Normally, a semaphore is only initialized *once*; any further modification to the semaphore is done *only* by routines `uP` and `uV`. However, if a semaphore is re-initialized, it should have no tasks waiting on it. This is because initialization is done by assignment, and hence, there is no way to generate an error or to unblock the waiting tasks. Any waiting tasks remain blocked and are inaccessible.

The routines `uP` and `uV` are used to perform the classical counting semaphore operations. `uP` decrements the semaphore counter and if the value of the semaphore counter is less than zero, the calling task blocks. `uV` wakes up the task blocked for the longest time if there are tasks blocked on the semaphore and increments the semaphore counter.

```
void uP( uSemaphore * semaphore-address );
void uV( uSemaphore * semaphore-address );
```

semaphore-address is the address of a semaphore variable which is modified by `uP` or `uV`. Unless the argument is already a pointer to a `uSemaphore`, it must be preceded by an `&`.

The routine `uC` returns the current value of a semaphore's counter.

```
int uC( uSemaphore * semaphore-address );
```

return-value is the value of the semaphore's counter *semaphore-address* is the address of a semaphore variable. Unless the argument is already a pointer to a `uSemaphore`, it must be preceded by an `&`.

If the counter is positive, that indicates the number of `uP` operations that can occur before a task blocks. If the counter is zero or negative, then the absolute value of the counter value is the number of blocked tasks waiting on the semaphore.

Appendix A.2 contains a complete P/V program.

2.7.3.2 Monitors

The μSystem supports monitors as a new programming language construct through a preprocessor that generates calls to routines in the μSystem. This preprocessor provides a monitor as a package construct along with new statements to control scheduling of tasks in the monitor. Monitors are described in the document "μMonitor Reference Manual" available with the μSystem.

2.7.3.3 Send/Receive/Reply/Forward

Message passing is used for synchronizing tasks and passing data between them. The two tasks involved in a communication are called the sender and receiver tasks. What characterizes send/receive/reply is that the sender blocks (i.e. does not continue execution) until the receiver receives the message and explicitly replies. All sends must be replied to, but the receiver does not need to reply to messages in the order that they were received. The following routines perform send/receive/reply communication between tasks and are largely derived from Thoth [Che82].

The sender takes on one of two states during a communication:

send-blocked which means the sender has done a send but the message has not been received.

reply-blocked which means the sender's message has been received but a reply has not been performed.

The receiver can be in the following state during a communication:

receive-blocked which means the receiver has done a receive but no message has been sent.

The routine `uSend` is used to transmit a message to another task. `uSend` blocks until the receiver has replied to the sent message.

```
uTask uSend( uTask receiver-task-id, void * reply-area, int reply-area-length,
             void * send-message, int send-message-length );
```

return-value is the task identifier of the task that replied to this send. Because of the ability to forward a message (detailed below), the replying task is not necessarily the same as the task sent to.

receiver-task-id is the task identifier of the receiving task.

reply-area is the address of a reply area into which the reply message of the receiving task will be copied.

reply-area-length is the size in bytes of the reply area.

send-message is the address of a message to be sent to the receiving task.

send-message-length is the size in bytes of the message to be sent.

Send transmits the argument `send_message` to the receiving task's `receive_area`.

The routine `uReceive` is used to receive a message sent from another task. `uReceive` receives a message sent to it from any task; it cannot be used to receive a message from a particular task. `uReceive` blocks if there is no task currently sending to it.

```
uTask uReceive( void * receive-area, int receive-area-length );
```

return-value is the task identifier of the sending task.

receive-area is the address of a receive area into which the sent message of the sending task will be copied.

receive-area-length is the size in bytes of the receive area.

When a message arrives, data from the sender's `send_message` argument is copied into the receiver's `receive_area` argument. After a task has received a message, it is obligated to reply to the sender task or to delegate the reply responsibility to another task by forwarding.

The routine `uReply` is used to reply to another task and to transmit a message back to the sender. `uReply` does not block.

```
void uReply( uTask sender-task-id, void * reply-message, int reply-message-length );
```

sender-task-id is the `uTask` identifier of a task that has sent a message to this task. The reply fails and an exception is raised if the specified sender task did not send a message to the replying task, or the replying task has already replied to this message from that sender.

reply-message is the address of a reply message to be sent back to the sender.

reply-message-length is the length of the reply message to be sent back to the sender.

The reply copies the argument `reply_message` back to the sending task's `reply_area` argument and the sending task is then unblocked and continues execution.

The routine `uForward` is used to transmit a message to another task on behalf of the task that originally sent the message. Once a message is forwarded, only the new receiving task can reply to it, unless the new receiving task forwards the message again. `uForward` blocks until the new receiver has received the forwarded message; no reply is necessary to the forwarder of a message, nor can a receiving task determine if a message was forwarded or sent by the original sender.

```
void uForward( uTask forward-task-id, void * send-message, int send-message-length,
               uTask sender-task-id );
```

forward-task-id is the `uTask` identifier of the task to which the message is forwarded.

send-message is the address of a message that is to be forwarded.

send-message-length is the length of the message to be forwarded.

sender-task-id is the `uTask` identifier of the original message sender. The forward fails and an exception is raised if the specified sender task did not send a message to the forwarding task.

There is no obligation on the part of the forwarder to forward the same message that it originally received from a sender. The forwarder can receive a message and forward a new message to another task on behalf of the original sender. The receiving task will service this new message and reply to the original sender or it can perform another forward.

Appendix A.3 contains a complete message passing program.

2.7.4 Task Termination

A task is terminated by executing the `uDie` routine. This routine can be invoked at any level of nested coroutine or subroutine invocation to terminate a task. `uDie` is used in conjunction with routine `uAbsorb`, which allows a task to wait for the completion of another task. The pair of routines allows a result to be passed from the terminating task back to the task waiting for its completion.

The routine `uAbsorb` waits for completion of a specified task, accepts its last result sent by `uDie`, and deallocates its resources.

```
void uAbsorb( uTask task-id, void * reply-area, int reply-area-length );
```

task-id is the task identifier of the completing task.

reply-area is the address of the reply area into which the message sent from `uDie` will be copied.

reply-area-length is the size in bytes of the reply area.

The routine `uDie` terminates execution of a task and passes back a result to some task awaiting its completion using `uAbsorb`.

`void uDie( void *` *send-message*`, int` *send-message-length* `);`

send-message is the address of the message to be sent to a task waiting for termination of this task.

send-message-length is the size in bytes of the message to be sent.

Each task terminated using `uDie` must be absorbed by only one task. If the terminating task is not absorbed, its resources will not be recovered. If multiple tasks absorb a task, only one will be successful and continue execution. The other absorbing tasks will block forever. Currently, there is no mechanism to explicitly unblock such a task (or any tasks that are blocked on it) or a timeout facility that can be specified on `uAbsorb` to implicitly unblock it.

The following shows how `uAbsorb` and `uDie` can be used to return a result from a task:

```
void f(float x, float y) {
    float result;
    ...                            /* calculate result concurrently with emitter */
    uDie(&result, sizeof(result)); /* terminate task and return result */
}

void uMain() {
    float result, a, b;
    uTask tid;
    ...
    tid = uEmit(f, a, b);    /* start a task running f concurrently */
    ...                      /* continue concurrently with f */
    uAbsorb(tid, &result, sizeof(result)); /* wait for task's completion and result */
}
```

Executing a `return` statement in an emitted routine is the same as the routine call `uDie(NULL, 0)`. This causes the task to terminate and wait to be absorbed. The same action occurs if control runs off the end of the emitted routine. Therefore, if a value is to be returned at task termination, it must be passed back with an explicit call to `uDie`.

2.8 Naming Coroutine and Task

The routine `uSetName` associates a string name with a coroutine or task.

`void uSetName( char *` *string-address* `);`

string-address the address of a string of characters terminated with `'\0'`.

The routine `uGetName` returns the string name associated with a coroutine or task.

`char *uGetName( uCoroutine` *coroutine-id* `);`
`char *uGetName( uTask` *task-id* `);`

return-value the address of a string of characters terminated with `'\0'`.

coroutine-id/task-id is the coroutine or task identifier of the entity whose name is returned.

If the coroutine or task has not been assigned a name, `uGetName` returns a pointer to a string that contains the address of the coroutine or task in hexadecimal.

The μSystem uses the coroutine or task name when printing any error message, which can be helpful when debugging.

Chapter 3

Runtime Environment

A cluster is a collection of μSystem tasks and virtual processors; it provides a runtime environment for their execution. This environment contains a number of variables that can be modified to affect how coroutines, tasks and virtual processors behave in a cluster.

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

There are two forms of blocking that can occur in the μSystem:

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

light blocking which is done by the μKernel on a task as a result of certain μSystem operations (e.g. `uP`, `uSend`.

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

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

3.1 Cluster

Each cluster has a number of environment variables that may be used implicitly when creating coroutines and tasks on that cluster and by virtual processors associated with that cluster (see Figure 3.1):

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

argument length is the default argument length used when coroutines or task are created on a cluster.

number of virtual processors is the number of virtual processors currently allocated on a cluster.

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

spin duration is the longest time that an idle virtual processor on the cluster spins waiting for a task to become ready before it goes to sleep.

Each of these variables is either explicitly set or implicitly assigned a system-wide machine-dependent default value when the cluster is created. The mechanisms to read and reset the values are detailed below.

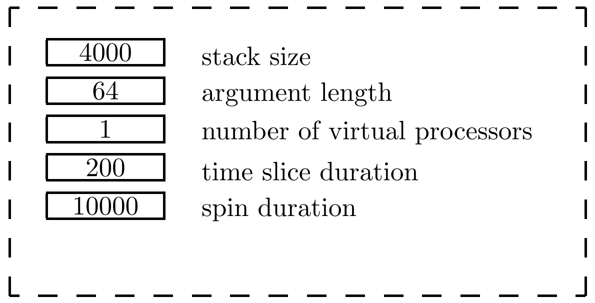

Figure 3.1: Cluster Variables

3.1.1 Cluster Type

`uCluster` is the type of a cluster identifier, as in:

```
uCluster x, y, z;
```

which creates three variables that contain cluster identifier values.

3.1.2 Cluster Creation

`uClusterVars` is the type of a structure that contains the initial defaults for a new cluster created by `uLongCreateCluster` (detailed next):

```
typedef struct {
    long Processors;
    long TimeSlice;
    long Spin;
    long StackSize;
    long ArgLen;
} uClusterVars;
```

`uClusterVars` variables must be initialized with the macro `U_CLUSTER_VARS()` before use, as in:

```
uClusterVars cv = U_CLUSTER_VARS();
```

This initializes the fields of `cv` to the system-wide machine-dependent default values. Individual fields can then be changed to user specified values. By always initializing `uClusterVars` variables with macro `U_CLUSTER_VARS()`, new cluster variables can be added in the future and programs do not have to be changed, only re-compiled.

The routine `uLongCreateCluster` creates a cluster with at least one virtual processor associated with it.

`uCluster uLongCreateCluster( uClusterVars *` *cluster-variable-address* `);`

return-value is the cluster identifier of the new cluster. This value must be retained as it is used to subsequently place tasks on the cluster, or to destroy the cluster.

cluster-variable-address is the address of a variable which contains the initial defaults for the new cluster.

The maximum number of clusters that can be created is indirectly limited by the number of UNIX processes a UNIX user can create, as the sum of the virtual processors on all clusters cannot exceed this limit.

The following shows how a cluster is created with 5 processors, no time slicing, a stack size default of 8000 bytes, and the machine-dependent default for the task argument-length and virtual-processor spin-time.

```
uClusterVars cv = U_CLUSTER_VARS(); /* set machine-specific defaults */
uCluster c;

cv.Processors = 5;                  /* change specific fields */
cv.TimeSlice = 0;
cv.StackSize = 8000;

c = uLongCreateCluster( &cv );
```

There is a short form of `uLongCreateCluster`, called `uCreateCluster`, that assumes the machine specific defaults for all cluster variables except number of processors and virtual-processor time-slice.

`uCluster uCreateCluster( int` *number-of-processors*`, long` *milliseconds* `);`

return-value is the cluster identifier of the new cluster. This value must be retained as it is used to subsequently place tasks on the cluster, or to destroy the cluster.

number-of-processors is the number of virtual processors initially associated with the new cluster.

milliseconds is an integer value representing the number of milliseconds between interrupts for each virtual processor on the cluster.

The routine `uThisCluster` is used to determine which cluster a task currently resides in.

`uCluster uThisCluster( void );`

return-value is the cluster identifier of the cluster on which the calling task resides.

3.1.3 Cluster Termination

The routine `uDestroyCluster` deallocates the specified cluster, which destroys all virtual processors associated with the cluster.

`void uDestroyCluster( uCluster` *cluster-id* `);`

cluster-id is a cluster identifier of the cluster to be destroyed.

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

3.2 Default Stack Size

The routines `uLongSetStackSize` and `uSetStackSize` are used to set the default stack size value for a coroutine or task stack allocated on a cluster.

```
void uLongSetStackSize( uCluster cluster-id, long new-stack-size );
void uSetStackSize( long new-stack-size );
```

cluster-id is the cluster identifier whose default stack size is modified.

new-stack-size is the number of bytes that is used as the default stack size.

For example, the call `uSetStackSize(8000)` sets the default stack size to 8000 bytes for the current cluster.
The routines `uLongGetStackSize` and `uGetStackSize` are used to read the value of the default stack size for a cluster.

```
long uLongGetStackSize( uCluster cluster-id );
long uGetStackSize( void );
```

return-value is the current default stack-size value.

cluster-id is the cluster identifier whose default stack size is returned.

For example, the call `i = uGetStackSize()` sets `i` to the value 8000.
The μSystem provides the routine `uVerify` to check if the current coroutine or task has overflowed its stack; if it has, an exception is raised.

```
void uVerify( void );
```

When debugging is enabled (`-debug` compilation flag), `verify` is implicitly called on each context switch. Since a coroutine or task often calls no other routines, it is suggested that a call to `uVerify` be included at the beginning of each, as in the following example:

```
void f( ... ) {
    ... declarations
    uVerify();   /* check for stack overflow */
    ... routine body
}
```

Thus, after each coroutine or task has allocated its local variables, a verification is made that the stack is large enough to contain them. If a coroutine or task calls other routines, each would have to start with a call to `uVerify` to check for stack overflow.

3.3 Default Argument Length

The routines `uLongSetArgLen` and `uSetArgLen` are used to set the default argument length for a coroutine or task started on a cluster.

```
void uLongSetArgLen( uCluster cluster-id, long new-argument-length );
void uSetArgLen( long new-argument-length );
```

cluster-id is the cluster identifier whose default argument length is modified.

new-argument-length is the number of bytes that is used as the default argument length.

For example, the call `uSetArgLen(100)` sets the default argument length to 100 bytes for the current cluster.
The routines `uLongGetArgLen` and `uGetArgLen` are used to read the value of the default argument length for a cluster.

```
long uLongGetArgLen( uCluster cluster-id );
long uGetArgLen( void );
```

return-value is the current default argument length.

cluster-id is the cluster identifier whose default argument length is returned.

For example, the call `i = uGetArgLen()` sets integer `i` to the value 100.

> □ In theory, it is possible to determine the argument length automatically from the argument list; however, this is only possible if the facilities to start a coroutine or task are integrated into the programming language. There are problems when an argument length is specified that is not the exact size of the argument list. If the length is greater, extra bytes are copied, which is runtime inefficient. If the length is less, argument information is not copied, which results in unpredictable behaviour or failure of the coroutine or task. □

3.4 Virtual Processors on a Cluster

The routines `uLongSetProcessors` and `uSetProcessors` create or destroy virtual processors as needed to have the specified number of processors on a cluster in the multikernel. This routine does nothing in the unikernel.

```
void uLongSetProcessors( uCluster cluster-id, int number-of-processors );
void uSetProcessors( int number-of-processors );
```

cluster-id is the cluster identifier whose number of virtual processors is modified.

number-of-processors is the number of virtual processors that will exist on the cluster.

For example, the call `uSetProcessors(5)` will increase or decrease the number of virtual processors on the current cluster to 5.

The routines `uLongGetProcessors` and `uGetProcessors` are used to read the number of processors on a cluster.

```
int uLongGetProcessors( uCluster cluster-id );
int uGetProcessors( void );
```

return-value is the current number of virtual processors on the cluster.

cluster-id is the cluster identifier whose number of virtual processors is returned.

For example, the call `i = uGetProcessors()` sets `i` to the value 5.

The system dependent macro `U_PHYSICAL_PROCESSORS` returns the maximum number of hardware processors available on a computer. This facility is not available on all systems.

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

The following are points to consider when deciding how may virtual processors to create for a cluster. First, there is no advantage in creating significantly more virtual processors than the average number of simultaneously active tasks on the cluster. For example, if on average three tasks are eligible for simultaneous execution, then creating significantly more than three virtual processors will not achieve any execution speed up and wastes resources. Second, while it is possible to create more virtual processors than actual hardware processors, there is usually a performance penalty in doing so. Having more virtual processors than actual processors can result in extra context switching of the heavy-weight UNIX processes, 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 on all the clusters exceeds the number of hardware processors, extra context switching of the UNIX processes will occur. However, multiple clusters must be used to handle blocking or non-interchangeable virtual processor problems. For example, the virtual processors associated with I/O clusters spend most of their time blocked and do not interfere with virtual processors on computational clusters. Finally, a μSystem program usually shares the actual hardware processors with other user programs. Therefore, the overall UNIX system load will affect how many virtual processors should be allocated to avoid unnecessary context switching of UNIX processes.

3.5 Implicit Task Scheduling

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

The routines `uLongSetTimeSlice` and `uSetTimeSlice` are used to set the default time slice for each virtual processor on a cluster.

```
void uLongSetTimeSlice( uCluster cluster-id, long milliseconds );
void uSetTimeSlice( long milliseconds );
uSetTimeSlice( milliseconds );
```

cluster-id is the cluster identifier whose default time slice is modified.

milliseconds is the number of milliseconds between interrupts.

For example, the call `uSetTimeSlice(50)` sets the default time slice to 0.05 seconds for each virtual processor on the current cluster. To turn pre-emption off, call `uSetTimeSlice(0)`.

□ On many machines the minimum time slice duration 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. (On System V UNIX, pre-emption occurs at most once a second, which may not be often enough to adequately test that a concurrent program does not depend on order or speed of execution of its tasks.) □

The overhead of pre-emptive scheduling depends on the frequency of the interrupts. Further, because interrupts involve entering the UNIX kernel, they are relatively expensive. We have found that an interrupt interval of 0.05 to 0.1 seconds adequately verifies that a concurrent program does not depend on order or speed of task execution and increases execution cost by less than 1% for most programs.

The routines `uLongGetTimeSlice` and `uGetTimeSlice` are used to read the default time slice for a cluster.

```
long uLongGetTimeSlice( uCluster cluster-id );
long uGetTimeSlice( void );
```

return-value is the current interrupt duration on the cluster.

cluster-id is the cluster identifier whose default time slice is returned.

For example, the call `i = uGetTimeSlice()` sets `i` to the value 50.

3.6 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 μKernel, an idle virtual processor spins for a user specified amount of time 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, then the next executable task will have to wait for completion of a UNIX 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 will be processed immediately. Selecting a spin time is application dependent and it can have a significant affect on performance.

The routines `uLongSetSpin` and `uSetSpin` are used to set the default spin-duration for each virtual processor on a cluster.

```
void uLongSetSpin( uCluster cluster-id, long microseconds );
void uSetSpin( long microseconds );
```

cluster-id is the cluster identifier whose default spin-duration is modified.

microseconds is the number of microseconds the idle virtual processor will spin before it sleeps.

For example, the call `uSetSpin(50000)` sets the default spin-duration to 0.05 seconds for each virtual processor on the current cluster. To turn spinning off, call `uSetSpin(0)`.

The routines `uLongGetSpin` and `uGetSpin` are used to read the default spin-duration for a cluster.

```
long uLongGetSpin( uCluster cluster-id );
long uGetSpin( void );
```

return-value is the current default spin-duration value.

For example, the call `i = uGetSpin()` sets `i` to the value 50000. The precision of the spin time is machine dependent and can vary from 1 to 50 microseconds.

3.7 Explicit Task scheduling

The routine `uYield` gives up control of the virtual processor to another ready task.

```
void uYield( void );
```

For example, the routine call `uYield()` returns control to the μKernel to schedule another task, hence immediately giving up control of the virtual processor. If there are no other ready tasks, the yielding task is restarted.

`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. 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 30-50 tasks before pre-emption occurs.) If after the creation of several tasks the creator yields control, then each created task will have an opportunity to begin execution (possibly only one instruction before pre-emption occurs) 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.

The routine `uDelay` invokes `uYield` N times.

```
void uDelay( int number-of-yields );
```

number-of-yields is the number of times `uYield` is invoked.

For example, the routine call `uDelay(5)` calls `uYield()` 5 times, hence immediately giving up control of the virtual processor and ignoring the next 4 times the task is scheduled for execution.

The routine `uReadyTasks` returns the number of tasks on the ready queue of the cluster that the current task is executing on.

```
int uReadyTasks( void );
```

return-value is the number of tasks on the ready queue of the current cluster.

The calling task is not on the ready queue, and hence, is not included in the ready-queue count.

3.8 Defaults for `uMain`

Because all the cluster defaults are set for the user cluster *before* `uMain` begins execution, the initial task is normally created with the machine-dependent cluster-values. This can cause problems if the default stack size is insufficient for the variables declared in `uMain`. If the default stack size for `uMain` is exceeded, `uMain` terminates with an addressing error on the first reference to a local variable beyond the stack. To deal with this, the μKernel calls the special free routine `uStart` before it calls `main`. `uStart` can reset any of the defaults for the initial user cluster, as in:

```
void uStart( void ) {
    if ( uGetStackSize() < 8000 ) {  /* check machine dependent stack size */
        uSetStackSize( 8000 );       /* set stack size to at least 8000 bytes */
    }
    uSetTimeSlice( 0 );              /* turn off time slicing before uMain begins */
}
```

If a `uStart` routine is not supplied, a default routine with a null execution body is supplied.

3.9 Migration

Most tasks execute on only one cluster. The routine `uMigrate` is used to move the calling task from one cluster to another so that it can access resources that are dedicated to that cluster's virtual processors.

`uCluster uMigrate( uCluster` *cluster-id* `);`

return-value is the previous cluster instance that the calling task was executing on.

cluster-id is the cluster instance that the task is moved to.

Chapter 4

Memory Management

All data that the μSystem manipulates must reside in memory that is accessible by all UNIX processes started by the μSystem. 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 the μSystem, all user data is located in the shared memory of the UNIX processes.

In order to make memory sharable across both versions of the μSystem and provide mutual exclusion on memory management operations, the μSystem provides memory management routines, called `uMalloc`, `uRealloc` and `uFree`. These routines provide identical functionality to the UNIX `malloc`, `realloc` and `free` routines.

4.1 Memory Allocation

The routine `uMalloc` allocates memory.

```
void *uMalloc( int number-of-bytes );
```

return-value the address of a block of memory of at least the requested size. `uMalloc` returns `void *` and so it is unnecessary to cast its result to the pointer type expected at the usage site.

number-of-bytes the number of bytes of memory to be allocated.

If `uMalloc` cannot allocate the requested memory, an exception is raised.

The following code shows an example of how to allocate memory.

```
int size;
void *addr;
...
addr = uMalloc( size );
```

The routine `uRealloc` increases or decreases the size of an existing allocated block of memory or moves the block to a new location that is at least of the specified size.

```
void *uRealloc( void * allocated-memory-address, int number-of-bytes );
```

return-value the address of a block of memory of at least the requested size. `uRealloc` returns `void *` and so it is unnecessary to cast its result to the pointer type expected at the usage site.

allocated-memory-address the address of an existing allocated area of memory.

number-of-bytes the number of bytes that the old allocated area is to be re-sized to.

If `uRealloc` cannot allocate the requested memory, an exception is raised.

4.2 Memory Deallocation

The routine `uFree` deallocates memory.

```
void uFree( void * address );
```

address of the block of memory to deallocate.

The following code shows how to deallocate memory.

```
void *addr;
...
uFree( addr );
```

Chapter 5

Interaction with the UNIX File System

As explained in chapter 3, it is desirable to avoid heavy blocking of virtual processes. UNIX I/O operations can be made to be nonblocking, but this requires special action as the I/O operations do not restart automatically when the operation completes. Instead, it is necessary to perform polling for I/O completions and possibly blocking of the UNIX process if all tasks are directly or indirectly blocked waiting for I/O operations to complete. To simplify the complexity of performing nonblocking I/O operations, the μSystem supplies a library of I/O routines that perform the nonblocking operations and polling (detailed below). The I/O cover routines have essentially the same syntax as the normal C or UNIX I/O routines; however, instead of a UNIX file descriptor being passed around to identify a file or socket, a μSystem file descriptor is used. There are two forms of the μSystem I/O routines: one returns UNIX return codes and the other raises exceptions. The compilation flag `-unixrc` chooses which form is loaded with an application. To ensure portability between unikernel and multikernel, the μSystem supplied cover routines should always be used.

5.1 Unikernel File Operations

To retain concurrency in the unikernel during I/O operations, the μSystem I/O routines check the ready queue before performing their corresponding UNIX I/O operation. If there are no tasks waiting to execute, then the single virtual processor is blocked because all tasks in the system must be directly or indirectly waiting for an I/O operation to complete. If there are tasks to execute, a nonblocking I/O operation is performed. The task performing the I/O operation then loops polling for completion of the I/O operation and yielding control of the processor if the operation has not completed. This allows other tasks to progress with a slight degradation in performance due to the polling task(s).

5.2 Multikernel File Operations

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

In detail, a cluster and a virtual processor are automatically created when a file is opened. Hence, each open file has a corresponding UNIX process. The exception to this rule is the standard I/O files, called `uStdin`, `uStdout` and `uStderr` in the μSystem, which are all open implicitly on the system cluster. A user task then performs I/O operations by executing the equivalent μSystem cover routine, which migrates the task to the cluster containing the file descriptor and performs the appropriate operation. When the user task closes the file, the cluster and all of its resources are released. To ensure that multiple tasks are not performing I/O operations simultaneously on the same file descriptor, each μSystem file descriptor has a semaphore that provides mutual exclusion of I/O operations. Figure 5.1 illustrates the runtime structures created for accessing a file (UNIX resources are illustrated with an oval). Depending on the kind of I/O, there may be one or several tasks on the I/O cluster.

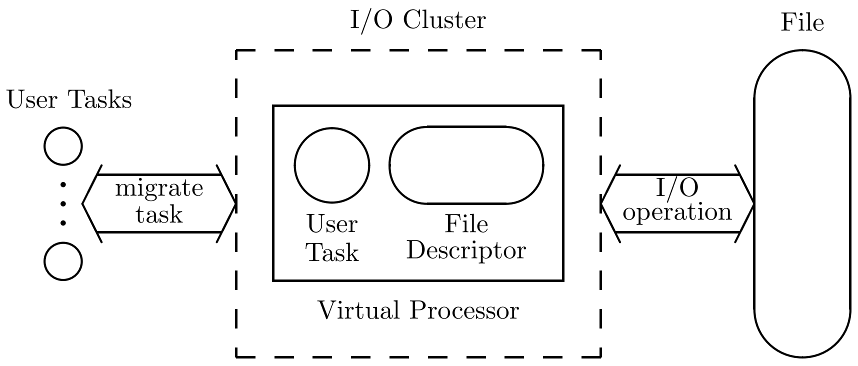

Figure 5.1: UNIX File I/O Cluster

Notice that mutual exclusion only occurs per file descriptor. If a file is opened multiple times, each opening creates a new and independent cluster, virtual processor and file descriptor. Access to these file descriptors on different clusters are not synchronized. This is not a problem if all tasks are reading the file, but will not work, in general, if some tasks read and some write to the same file.

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

5.3 C Standard I/O Routines

To aid the programmer, there are cover routines for the following C standard I/O operations, which either work exactly like their C standard counterpart returning error codes or instead raise exceptions. In addition, their operations are performed on a separate cluster. In the unikernel case, creating a cluster returns a reference to the current cluster so all files are open on the virtual processor for this single cluster. As well, there are three file identifiers `uStdout`, `uStdin`, and `uStderr` which identify the file descriptors managing the corresponding files `stdout`, `stdin`, and `stderr`, respectively. For complete details on each cover routine, first refer to the `man` pages for the corresponding C standard I/O routine. To get the I/O cover routines that raise exceptions instead of returning return-codes, use the compilation flag `-nounixrc`.

5.3.1 Stream Type

`uStream` is the type of a C standard I/O file identifier, as in:

```
uStream input, output;
```

which creates two variables that contain C standard I/O file identifier values.

5.3.2 Opening a File

A file is opened with the `uFopen` routine.

`uStream uFopen( char *` *unix-file-name*`, char *` *open-type* `);`

return-value is a stream identifier to the file (or null if the attempt failed when using UNIX return codes). This value must be retained as it is used to subsequently access the file.

unix-file-name is the address of a string of ASCII characters representing a UNIX path name to the file, terminated by a null character.

open-flag indicates how the file is to be opened for access. See the man entry for `fopen` for the options.

`uFopen` creates a cluster and places one virtual processor on that cluster. Then `uFopen` migrates the calling task to the new cluster, which performs an actual UNIX `open` operation, creating the UNIX file descriptor on the virtual processor for that cluster. The calling task is then migrated back to its original cluster. The following example shows the opening of a file:

```
uStream input;
...
input = uFopen( "test.c", "r" );
```

When file descriptors are in short supply, it is possible to open a new file using an existing file's descriptor with the `uFreopen` routine. The previous file using the descriptor is closed.

`uStream uFreopen( char *` *unix-file-name*`, char *` *open-type*`, uStream` *stream-id* `);`

return-value is *previous-file-id* (or null if the attempt failed when using UNIX return codes). This value must be retained as it is used to subsequently access the file.

unix-file-name is the address of a string of ASCII characters representing a UNIX path name to the file, terminated by a null character.

open-flag indicates how the file is to be opened for access. See the man entry for `fopen` for the options.

previous-file-id is a stream identifier to a previously opened file.

`uFreopen` reuses the existing cluster created for the previous file. The following example shows the opening of a file and closing another:

```
uStream input;
...
input = uFreopen( "test2.c", "r", input );
```

5.3.3 Reading and Writing from a File

The routine `uPrintf` converts, formats, and prints its arguments on the standard output file which is usually the interactive terminal.

`int uPrintf( char *` *format*`, ... );`

return-value is the number of characters transmitted (or `EOF` if the attempt failed when using UNIX return codes).

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

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

The following example shows printing on the standard output file:

```
uPrintf( "Hello World\n" );
```

The routine `uPutc` writes a character on the specified output file (identical to `uFputc` below).

```
int uPutc( char character, uStream file-id );
```

return-value is the character that was written (or EOF if the attempt failed when using UNIX return codes).

character is the character to be appended to the end of the file denoted by the specified *file-id*.

file-id is a stream identifier to the file.

The following example shows printing a character on an arbitrary output file:

```
int ch;
uStream output;
...
ch = uPutc( 'c', output );
```

The routine `uPutchar` writes a character on the standard output file which is usually the interactive terminal).

```
int uPutchar( char character );
```

return-value is the character that was written (or EOF if the attempt failed when using UNIX return codes).

character is the character to be appended to the end of the standard output file.

The following example shows printing a character on the standard output file:

```
uPutchar( 'c' );
```

The routine `uPuts` writes a character string on the standard output file which is usually the interactive terminal.

```
void uPuts( char * character-string );
```

character-string is the string of characters terminated with `'\0'` to be appended to the end of the standard output file.

The following example shows printing a string on the standard output file:

```
uPuts( "abc" );
```

The routine `uFprintf` performs the same operation as `uPrintf` but does not default to printing on standard output. Instead, it can print formatted output on any specified file.

```
int uFprintf( uStream file-id, char * format, ... );
```

return-value is the number of characters transmitted (or EOF if the attempt failed when using UNIX return codes).

file-id is a stream identifier to the file.

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

... is a list of arguments to be formatted and printed on the file denoted by the specified *file-id*. The number of elements in this list must match with the number of format codes.

The following example shows printing on an arbitrary output file:

```
uStream output;
...
uFprintf( output, "This is the number %d\n", 1 );
```

The routine `uFputc` writes a character on the specified output file (identical to `uPutc` above).

`int uFputc( char` *character*`, uStream` *file-id* `);`

return-value is the character that was written (or EOF if the attempt failed when using UNIX return codes).

character is the character to be appended to the end of the file denoted by the specified *file-id*.

file-id is a stream identifier to the file.

The following example shows printing a character on an arbitrary output file:

```
uStream output;
...
uFputc( 'c', output );
```

The routine `uFputs` performs the same operation as `uPuts` but does not default to printing on standard output.

`int uFputs( char *` *character-string*`, uStream` *file-id* `);`

return-value is non-negative (or EOF if the attempt failed when using UNIX return codes).

character-string is the string of characters terminated with `'\0'` to be appended to the end of the file denoted by the specified *file-id*.

file-id is a stream identifier to the file.

The following example shows printing a string on an arbitrary output file:

```
uStream output;
...
uFputs( "abc", output );
```

The routine `uFwrite` performs buffered binary output to the specified output file.

`int uFwrite( char *` *buffer-address*`, int` *object-size*`, int` *number-of-objects*`, uStream` *file-id* `);`

return-value is the number of objects written to the specified file (or zero on EOF or if the attempt failed when using UNIX return codes).

buffer-address is a pointer to a buffer area from which the objects are written.

object-size is the size in bytes of the homogeneous objects.

number-of-objects is the number of objects to be written into the buffer area.

file-id is a stream identifier to the file.

The following example shows writing objects into an arbitrary output file:

```
uStream output;
double buf[10];
...
n = uFwrite( buf, sizeof(double), 10, output );
```

□ In previous versions of the μSystem, there was a problem with the occasional missing carriage return when writing to a terminal. This was dealt with by directing the output into a file (e.g. `>` or `>>`) or through a filer (`|`), for example:

`a.out | more` *output from μSystem program piped into* `more` *ensures no loss of carriage returns*

However, we believe this problem has been fixed. □

The routine `uScanf` reads characters from the standard input file, interprets then according to the specified format codes, and stores the result in the arguments.

`int uScanf( char *` *format*`, ... );`

return-value is the number of successfully matched and assigned input items (or EOF on end-of-file or if the attempt failed when using UNIX return codes).

format is a format string containing text to be matched and format codes which describe how to interpret input text for assignment to the following variable number of arguments.

... is a list of pointers to variables that are assigned the interpreted text values from standard input. The number of elements in this list must match with the number of format codes.

The following shows scanning input from the standard input file:

```
int a, b;
...
uScanf( "%d %d", &a, &b );
```

The routine `uGetc` reads a character from the specified input file (identical to `uFgetc` below).

`int uGetc( uStream` *file-id* `);`

return-value is the next character, returned as an integer, read from the file denoted by *file-id* (or EOF on end-of-file or if the attempt failed when using UNIX return codes).

file-id is a stream identifier to the file.

The following example shows reading a character from an arbitrary input file:

```
int ch;
uStream input;
...
ch = uGetc( input );
```

The routine `uGetchar` reads a character from the standard input file which is usually the interactive terminal.

`int uGetchar( void );`

return-value is the next character, returned as an integer, from the standard input file (or EOF on end-of-file or if the attempt failed when using UNIX return codes).

The following example shows reading a character from the standard input file:

```
int ch;

ch = uGetchar();
```

The routine `uGets` reads n-1 characters, or up to a newline, from the standard input file into a string area.

`char *uGets( char *` *string-pointer*`, int` *number-of-characters* `);`

return-value is the value of the first argument (or `NULL` on end-of-file or if the attempt failed when using UNIX return codes).

string-pointer is a pointer to a string area into which characters are read from the standard input file. The string is terminated by a newline character.

number-of-characters is the maximum number of characters to be read into the string area plus the newline character.

The following example shows reading a string from the standard input file:

```
char *s;
...
s = uGets( s, 21 );
```

This routine is different from the standard UNIX `gets`, which does not have the *number-of-characters* parameter. However, we chose to be consistent with routine `fgets`, which has the *number-of-characters* parameter.

The routine `uUngetc` pushes the specified character onto the specified input file so that it can be read as if it appeared in the input.

`int uUngetc( int` *character*`, uStream` *file-id* `);`

return-value is the character that is pushed back (or `EOF` if the attempt failed when using UNIX return codes).

character is the character to be pushed back onto the file denoted by the specified *file-id*.

file-id is a stream identifier to the file.

The following example shows pushing a character back onto an arbitrary input file:

```
int ch;
uStream input;
...
ch = uUngetc( ch, input );
```

The routine `uFscanf` performs the same operation as `uScanf` but does not default to reading from standard input. Instead, it can read from any specified file.

`extern int uFscanf( uStream` *file-id*`, char *` *format*`, ... );`

return-value is the number of successfully matched and assigned input items (or EOF on end-of-file or if the attempt failed when using UNIX return codes).

file-id is a stream identifier to the file.

format is a format string containing text to be matched and format codes which describe how to interpret input text for assignment to the following variable number of arguments.

... is a list of pointers to variables that are assigned the interpreted text values from standard input. The number of elements in this list must match with the number of format codes.

The following shows scanning input from an arbitrary input file:

```
int a, b;
uStream input;
...
uFscanf( input, "%d %d", &a, &b );
```

The routine `uFgetc` reads a character from the specified input file (identical to `uGetc` above).

`int uFgetc( uStream` *file-id* `);`

return-value is the next character, returned as an integer, read from the file denoted by *file-id* (or EOF on end-of-file or if the attempt failed when using UNIX return codes).

file-id is a stream identifier to the file.

The following example shows reading a character from an arbitrary input file:

```
int ch;
uStream input;
...
ch = uFgetc( input );
```

The routine `uFgets` performs the same operation as `uGets` but does not default to reading from standard input.

`char *uFgets( char *` *string-pointer*`, int` *number-of-characters*`, uStream` *file-id* `);`

return-value is the value of the first argument (or `NULL` on end-of-file or if the attempt failed when using UNIX return codes).

string-pointer is a pointer to a string area into which characters are read from the file denoted by the specified *file-id*. The string is terminated by a newline character.

number-of-characters is the maximum number of characters to be read into the string area plus the newline character.

file-id is a stream identifier to the file.

The following example shows reading a string from an arbitrary input file:

```
char *s;
uStream input;
...
s = uFgets( s, 21, input );
```

The routine `uFread` performs buffered binary input from the specified input file.

`int uFread( char *` *buffer-address*`, int` *object-size*`, int` *number-of-objects*`, uStream` *file-id* `);`

return-value is the number of objects read from the specified file (or zero on EOF or if the attempt failed when using UNIX return codes).

buffer-address is a pointer to a buffer area into which the objects are read.

object-size is the size in bytes of the homogeneous objects.

number-of-objects is the number of objects to be read into the buffer area.

file-id is a stream identifier to the file.

The following example shows reading objects from an arbitrary input file:

```
uStream input;
double buf[10];
...
n = uFread( buf, sizeof(double), 10, input );
```

5.3.4 File Positioning

To directly seek to a particular location within a file the following routines are used. The routine `uFseek` positions to a particular location in a file for subsequent read or write operations.

`int uFseek( uStream` *file-id*`, long` *offset*`, int` *origin* `);`

return-value is zero (or -1 if the attempt failed when using UNIX return codes).

file-id is a stream identifier to the file.

offset is a signed distance (offset) from *origin*.

origin is a code indicating the beginning of the file, the current position in the file, or the end of the file.

The following shows how to advance 200 bytes forward from the current position in the specified file:

```
uStream input;
...
uFseek( input, 200, SEEK_CUR );
```

The routine `uRewind` positions to the beginning of a file for subsequent read or write operations. It clears the end of file and error indicator for the specified file.

`void uRewind( uStream` *file-id* `);`

file-id is a stream identifier to the file.

The following shows how to position to the beginning of the specified file:

```
uStream input;
...
uRewind( input );
```

5.3.5 Flushing a File

It may be necessary to flush the output buffer to a file to insure that all output is written before the program continues. A file buffer is flushed with the `uFflush` routine.

`int uFflush( uStream` *file-id* `);`

return-value is zero (or EOF if the attempt failed when using UNIX return codes).

file-id is a stream identifier to the file.

The following shows how to flush a file:

```
uStream input;
...
uFflush( input );
```

5.3.6 Closing a File

A file is closed with the `uFclose` routine.

`int uFclose( uStream` *file-id* `);`

return-value is zero (or EOF if the attempt failed when using UNIX return codes).

file-id is a stream identifier to the file.

`uFclose` closes the file. This destroys the virtual processor and the cluster associated with it. The following example shows the closing of a file:

```
uStream input;
...
uFclose( input );
```

5.3.7 File Status Inquiries

The following operations return information about the status of a particular file. The routine `uFeof` indicates if end of file has been reached on the specified input file.

`int uFeof( uStream` *file-id* `);`

return-value is non-zero when end of file is reached and zero otherwise.

file-id is a stream identifier to the file.

The routine `uFerror` indicates if an error has occurred during reading or writing from the specified file.

`int uFerror( uStream` *file-id* `);`

return-value is non-zero when an error has occurred and zero otherwise.

file-id is a stream identifier to the file.

The routine `uClearerr` resets the error indication on the specified file.

`void uClearerr( uStream` *file-id* `);`

file-id is a stream identifier to the file.

The routine `uFileno` returns the integer file descriptor associated with the specified file.

`int uFileno( uStream` *file-id* `);`

return-value is the integer file descriptor associated with the file.

file-id is a stream identifier to the file.

5.4 UNIX I/O

To aid the programmer, there are cover routines for the UNIX I/O operations, which either work exactly like their UNIX counterpart returning error codes or instead raise exceptions in addition to performing their operations on a separate cluster. In the unikernel case, creating a cluster returns a reference to the current and only cluster. For complete details on each cover routine, first refer to the `man` pages for the corresponding routine. To get the I/O cover routines that raise exceptions instead of returning return-codes, use the compilation flag `-nounixrc`.

To use the UNIX I/O facilities in a C program, include the file:

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

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

5.4.1 File Type

`uFile` is the type of a UNIX file identifier, as in:

```
uFile input, output;
```

which creates two variables that contain UNIX file identifier values.

5.4.2 Opening a File

A file is opened with the `uOpen` routine.

```
uFile uOpen( char * unix-file-name, int open-flag, int protection-mode );
```

return-value is a file identifier to the file. This value must be retained as it is used to subsequently access the file.

unix-file-name is the address of a string of ASCII characters representing a UNIX path name to the file, terminated by a null character.

open-flag indicates how the file is to be opened for access. See the man entry for `open` for the options.

protection-mode is the protection mode for a newly created file. See the manual entry for `open` for the protection modes.

`uOpen` creates a cluster and places one virtual processor on that cluster. Then `uOpen` migrates the calling task to the new cluster, which performs an actual UNIX open operation, creating the file descriptor on the virtual processor for that cluster. The calling task is then migrated back to its original cluster. The following example shows the opening of a file:

```
uFile input;
...
input = uOpen( "test.c", O_RDONLY, 0 );
```

5.4.3 Reading and Writing from a File

After opening a file, it is read and/or written with the routines `uRead` and `uWrite`. Both `uRead` and `uWrite` have the same parameters.

```
int uRead( uFile file-id, void * buffer-address, int number-of-bytes );
int uWrite( uFile file-id, void * buffer-address, int number-of-bytes );
```

return-value is the number of bytes actually read from the file by `uRead` or written to the file by `uWrite`. This number may not be the same as the number of bytes requested either because the end of file is reached for a read operation, or no more bytes can be written by the write operation.

file-id is a file identifier to the file.

buffer-address is the address of an area into which the bytes are read into or written from.

number-of-bytes is the number of bytes to be read from or written to the file buffer.

The following example shows reading from and writing to a file:

```
uFile input, output;
char *buf;
int len, count;
...
count = uRead( input, buf, len );
count = uWrite( output, buf, len );
```

5.4.4 Random Access Within a File

The current location of the file pointer associated with an open file may be modified with the `uLseek` routine.

`off_t uLseek( uFile` *file-id*`, off_t` *offset*`, int` *whence* `);`

return-value receives the updated value of the file pointer.

file-id is a file identifier to the file.

offset depending on the value of *whence*, this value sets the file pointer, increments the file pointer, or extends the file.

whence determines how the value of *offset* is interpreted.

The following example shows how to modify a file pointer:

```
uFile direct;
off_t pos;
...
pos = uLseek( direct, 0, L_SET );    /* move pointer to beginning of file */
pos = uLseek( direct, 100, L_INCR ); /* pointer is moved forward 100 bytes */
pos = uLseek( direct, 200, L_XTND ); /* pointer is moved 200 bytes past end of file */
```

5.4.5 Synchronizing a File

The routine `uFsync` causes all modified data and attributes of a file to be saved on permanent storage. This normally results in all modified copies of buffers for the associated file to be written to disk.

`int uFsync( uFile` *file-id* `);`

return-value is a return code of -1 for any error and zero otherwise.

file-id is a file identifier to the file.

5.4.6 Closing a File

A file is closed with the `uClose` routine.

`int uClose( uFile` *file-id* `);`

return-value is a return code of -1 for any error and zero otherwise.

file-id is a file identifier to the file.

`uClose` closes the file, and destroys the virtual processor and the cluster associated with it. The following example shows the closing of a file:

```
uFile input;
...
uClose( input );
```

5.5 Socket I/O

To aid the programmer, there are cover routines for the UNIX socket operations, which either work exactly like their UNIX counterpart returning error codes or instead raise exceptions in addition to performing their operations on a separate cluster. In the unikernel case, creating a cluster returns a reference to the current and only cluster. For complete details on each cover routine, first refer to the `man` pages for the corresponding UNIX routine. To get the I/O cover routines that raise exceptions instead of returning return-codes, use the compilation flag `-nounixrc`.

A client-server model of socket communication is be used, where each client connects with a particular server and each server can connect with multiple clients. Once a connection is established between client and server, communication can be bidirectional between them. After a socket is created, it is specialized as either a server socket or a client socket. A socket can be closed and subsequently re-specialized. The following discussion on socket routines indicates for each routine, whether it is used by a client application, or a server application.

To use socket I/O facilities in a C program, include the file:

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

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

5.5.1 Socket Creation

Both client and server applications must create a socket with the `uSocket` routine.

`uFile uSocket( int` *address-format*`, int` *communication-type*`, int` *protocol* `);`

return-value is a file identifier to the socket. This value must be retained as it is used to subsequently specialize the socket as a client or server.

address-format is an address format for interpreting subsequent addresses in socket operations. See the man entry for `socket` for the formats.

communication-type is a type which indicates the semantics of communication. See the man entry for `socket` for the types.

protocol is a particular protocol to be used with the socket. See the man entry for `socket` for the different protocols.

`uSocket` creates a cluster and places one virtual processor on that cluster. Then `uSocket` migrates the calling task to the new cluster, which performs an actual UNIX socket operation, creating the UNIX socket descriptor on the virtual processor for that cluster. The calling task is then migrated back to its original cluster. The following example shows the creation of a socket:

```
uFile socket;
int af, type, protocol;
...
socket = uSocket( af, type, protocol );
```

5.5.2 Server Socket Routines

The following routines are used by applications using sockets for the server side of the model.

5.5.2.1 Binding a Name to a Socket

A server application will bind a name to a socket with the `uBind` routine.

`int uBind( uFile` *socket-id*`, void *` *socket-name*`, int` *socket-name-length* `);`

return-value is a return code of -1 for any error and zero otherwise.

socket-id is a file identifier of a socket.

socket-name is an address of a structure in which the socket name will be placed.

socket-name-length is a the length of the new socket name.

This socket is now a server. The following example shows the binding of a socket with a socket name making it a server:

```
uFile server;
struct sockaddr *name;
int namelen;
...
uBind( server, name, namelen );
```

5.5.2.2 Listening to a Socket

A server application must set a limit on the number of incoming connections from clients with the `uListen` routine.

`int uListen( uFile` *server-id*`, int` *max-queue-length* `);`

return-value is a return code of -1 for any error and zero otherwise.

server-id is a file identifier of a socket that is now a server from a call to `uBind`.

max-queue-length is the maximum length the queue of pending connections.

The following example shows the setting of a maximum number of connections to a server socket:

```
uFile server;
int logsize;
...
uListen( server, logsize );
```

5.5.2.3 Accepting a Connection

A server can accept multiple connections with the `uAccept` routine.

`uFile uAccept( uFile` *server-id*`, void *` *socket-name*`, int *` *socket-name-length* `);`

return-value is a file identifier to the connection through which communicate can occur with a client.

server-id is a file identifier of a server that is managing connections on a socket.

socket-name is an address of a structure containing the socket name.

socket-name-length is a the length of the socket name.

When a client arrives, a connection is established between client and server and `uAccept` returns (unless the server is marked nonblocking). The *connection-id* is use in transfer data through the connection between the client and the server. After the connection is created, the server is available to establish more connections with other clients. The following example shows how to accept a connection on a socket.

```
uFile server, connection;
struct sockaddr *name;
int namelen;
...
connection = uAccept( server, name, namelen );
```

5.5.3 Client Socket Routines

The following routines are used by applications using sockets for the client side of the model.

5.5.3.1 Making a Connection

A client application makes a connection to a server with the `uConnect` routine.

`int uConnect( uFile` *socket-id*`, void *` *server-socket-name*`, int` *server-socket-name-length* `);`

return-value is a return code of -1 for any error and zero otherwise.

socket-id is a file identifier of a socket.

server-socket-name is an address of a structure containing a server socket name.

server-socket-name-length is a the length of the server socket name.

`uConnect` returns when the socket has connected with a server socket. This socket is now a client and it communicates with the server through the connection that was created by the server's accept. The following example shows the connection of a socket with a server making the socket into a client:

```
uFile client;
struct sockaddr *server_name;
int server_namelen;
...
uConnect( client, server_name, server_namelen );
```

5.5.4 Communicating on a Socket

The following routines are used to communicate among clients and connections. After a connection has been established between a client and a connection for a server, communication between client and connection is performed with the `uRead` and `uWrite` routines or the `uRecv`, `uRecvfrom`, `uRecvmsg` and `uSsend`, `uSendto`, `uSendmsg` routines.

Both `uRead` and `uWrite` have the same parameters.

`int uRead( uFile` {*client,connection*}*-id*`, void *` *buffer-address*`, int` *number-of-bytes* `);`
`int uWrite( uFile` {*client,connection*}*-id*`, void *` *buffer-address*`, int` *number-of-bytes* `);`

return-value is the number of bytes actually read from the socket by `uRead` or written to the socket by `uWrite`. This number may not be the same as the number of bytes requested if the requested amount of bytes had not yet arrived. `uRead` operations do not always block until the socket receives the requested amount of bytes. Rather, when some bytes have arrived, and a significant delay has passed, the `uRead` routine will return with only those bytes. Therefore, an application may have to poll the socket until it receives the requested number of bytes.

{*client,connection*}*-id* is a file identifier to a client or a connection.

buffer-address is the address of an area into which the bytes are read.

number-of-bytes is the number of bytes to be read from the socket into the buffer.

The following example shows bidirectional communication from a client to some connection:

```
uFile client;
char *buf;
int len, count;
...
count = uRead( client, buf, len );
count = uWrite( client, buf, len );
```

`uRecv` may be used only when the socket is in a connected state, while `uRecvfrom` and `uRecvmsg` take an unconnected socket and perform the connection with the given named socket.

```
int uRecv( uFile {client connection}-id, void * msg-address, int number-of-bytes, int flags );
int uRecvfrom( uFile socket-id, void * msg-address, int number-of-bytes, int flags,
               void * from, int * fromlen );
int uRecvmsg( uFile socket-id, void * msg-header, int flags );
```

return-value is the number of bytes actually received from the socket by `uRecv`, `uRecvfrom`, and `uRecvmsg`.

{*client,connection*}*-id* is a file identifier to a client or a connection.

socket-id is a file identifier of a socket.

msg-address/msg-header is the address of an area into which the message is received or the address of a `msghdr` structure, containing information about a connection and series of messages areas.

number-of-bytes is the number of bytes to be received.

flags is an indicator for specializing the behaviour of the receive request.

from is an address of a structure containing a socket name.

fromlen is the length of the socket name.

`uSsend` may be used only when the socket is in a connected state, while `uSendto` and `uSendmsg` take an unconnected socket and perform the connection with the given named socket. (The UNIX routine `send` is named `uSsend` because of the conflict with the μSystem routine `uSend`.)

```
int uSsend( uFile {client connection}-id, void * msg-address, int number-of-bytes, int flags );
int uSendto( uFile socket-id, void * msg-address, int number-of-bytes, int flags,
             void * from, int fromlen );
int uSendmsg( uFile socket-id, void * msg-header, int flags );
```

return-value is the number of bytes actually sent from the socket by `uSsend`, `uSendto`, and `uSendmsg`.

{*client,connection*}*-id* is a file identifier to a client or a connection.

socket-id is a file identifier of a socket.

msg-address/msg-header is the address of the message to be sent or the address of an array of `msghdr` structure, each structure containing information about a connection and series of messages to be sent.

number-of-bytes is the number of bytes to be sent.

flags is an indicator for specializing the behaviour of the send request.

from is an address of a structure containing a socket name.

fromlen is the length of the socket name.

□ The μSystem does *not* support out-of-band data on sockets. Out-of-band data requires the ability to install a signal handler on the I/O cluster. Currently, there is no facility to do this. □

5.5.5 Miscellaneous

`uGetsockname` returns the current name of the specified socket.

`int uGetsockname( uFile` {*client,server,connection*}*-id*`, void *` *name*`, int *` *namelen* `);`

return-value is a return code of -1 for any error and zero otherwise.

{*client,server,connection*}*-id* is a file identifier to a client, server or connection.

name is the area in which the socket name is copied.

namelen is the length of the area for the socket name.

5.5.6 Closing a Socket

A client application can close a socket with the `uClose` routine. A server application can close either a connection or close the socket with the `uClose` routine.

`int uClose( uFile` {*client,server,connection*}*-id* `);`

return-value is a return code of -1 for any error and zero otherwise.

{*client,server,connection*}*-id* is a file identifier to a client, server or connection.

There is a significant difference between closing a server or closing a connection. Closing a server causes the entire socket to be destroyed, and no more communication is possible. Closing a connection causes only that connection to be terminated, and the server remains available for further communications from clients. The following is an example of closing a server.

```
uFile server;
...
uClose( server );
```

Appendix A.4 contains a complete socket program.

Chapter 6

Abnormal Event Handling Facilities

The μSystem presents some general facilities for dealing with abnormal events during program execution. In this discussion, 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 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 contextually determined, as opposed to statically determined; otherwise, the same action and context for that action will be executed for every exceptional change in control flow.

2. The operation can call a correction routine, which either takes some corrective action so that the operation can succeed or determines that a correction is not possible and fails as in case 1. The corrective action is an *intervention* in the normal computation of an operation. To be useful, the correction routine has to be contextually determined, as opposed to statically determined; otherwise, the same action and context for that action will be executed for every intervention of an operation.

Both kinds of abnormal events can be handled in the μSystem. Thus, there are two possible outcomes of an operation: normal completion possibly returning a value, or failure with a change in control flow.

Users can define named exceptions and interventions in conjunction with ones defined by the μSystem. Exception handlers and intervention routines for dealing with abnormal events can be defined/installed at any point in a user's code. An exception or intervention can then be raised or called, passing data about the abnormal event and returning results for interventions. Interventions can also be activated in other tasks, like a UNIX signal. Such asynchronous interventions may interrupt a task's execution and invoke the specified intervention routine. Asynchronous interventions are found to be useful to get another task's attention when it is not listening through the synchronous communication mechanism.

□ The case where an operation returns a special value to indicate an abnormal event, i.e. a **return code**, is not an actual failure of the operation. 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. 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. □

6.1 Exceptions

The programmatic mechanism that indicates failure is called an **exception**, and it causes control to transfer to a block of code called a **handler**. By defining handlers in different scopes, there may be several distinct handlers to which control may be transferred. The exact scope of a handler depends upon the programming unit it is associated with; this scope might be an expression or a statement or a block.

When an operation fails and control transfers to a particular handler, the computation carried out in the handler's scope is not completed. For example, if the handler is associated with an expression, the failing operation and any subsequent operations in the expression are not executed. If the handler is associated with a block, the failing operation and any subsequent operations in the expression and statements in the block are not executed. In this discussion, a handler's function is to adjust the current environment so that execution can continue. This is called **forward error recovery** [LA90, p. 146]. The scope of a handler is chosen so that control is transferred to a point where recovery by the handler is possible. If a handler cannot recover or if there is no appropriate handler, the failure can be propagated to the invoker (caller) of the current operation or can cause an error which terminates execution.

6.1.1 Exception Model

The exception model used is a multilevel-termination model. In this model, the raising of a specific exception causes control flow to transfer to a handler in the current block or some active block above the current one; all intervening blocks between the block in which the exception is raised and the block where it is handled are terminated. This includes blocks created with braces {} and routine calls. Therefore, it is not possible to resume an operation after an exception is raised. The multilevel model was chosen because it provides incremental program construction; it is possible to add the raising of an exception to a routine without having to change any of the routines that call this routine. The termination model was chosen because simulating it violates abstraction; to simulate it using normal C constructs requires setting and testing of global flag variables in multiple locations in a program (see reference [BMZ] for more design details).

As a simple introduction, Figure 6.1 shows an example that defines two exceptions, one raised in routine `f1` and the other in `f2` when an abnormal event is detected.

6.1.2 Exception Definition

An exception definition is divided into two parts: the exception declaration and the type of the exception instance.

6.1.2.1 Exception Declaration

An exception is declared using type `uException`. This allocates storage that provides the unique address for identifying the exception, while the storage itself is used to represent the relationships between exceptions (as described below). When separate compilation units are linked, the linker ensures that externally declared exceptions have the same identity. When an exception is raised, the address of the exception is used as (part of) the search key during the searching for the handlers. In almost all of the subsequent contexts where an exception is used, it is the exception address that is needed. While a compiler could automatically determine these contexts if exceptions were integrated into the language, this is not possible in our implementation. Therefore, in almost all cases, a user must precede an exception name with the address-of operator (`&`). Attempts to automatically insert the `&` failed due to the inadequacies of the C preprocessor and a desire to allow pointers to exceptions.

An exception must be initialized at the point of declaration using the macro `U_EXCEPTION` and cannot change afterwards, i.e. it has type qualifier `const`; the parameter of `U_EXCEPTION` specifies the address of the exception from which the new one is derived. If an exception is not derived from any other exception, then it must be derived from the predefined exception `uAny`. For example,

```
uException except1 = U_EXCEPTION( &uAny ), except2 = U_EXCEPTION( &uAny );
```

declares two named exceptions and derives them from the predefined exception `uAny`. A more extensive example:

```
uException SingularMatrix = U_EXCEPTION( &uAny );
uException DimensionMismatch = U_EXCEPTION( &uAny );

void MatrixInvert( ... ) {
    ...
        if ( check for singular matrix ) {
            char *data = "singular matrix";
            uRaise( NULL, &SingularMatrix, data, strlen( data ) + 1 );
        }
    ...
}

void MatrixMult( ... ) {
    if ( check if row/column dimensions are correct ) {
        char *data = "non-matching row/column dimensions for matrix multiply";
        uRaise( NULL, &DimensionMismatch, data, strlen( data ) + 1 );
    }
    ...
}

void uMain( int argc, char *argv[], char *envp[] ) {
    char *edata;
    int i;
    ...

    for ( i = 0; i < N; i += 1 ) {
        uExcept {
            MatrixInvert( M );  /* block in which exception handlers are active */
            MatrixMult( M, N );
        } uHandlers {
            uWhen( NULL, &SingularMatrix, &edata ) {
                uFprintf( uStderr, "%s\n", edata );
            }
            uWhen( NULL, &DimensionMismatch, &edata ) {
                uFprintf( uStderr, "%s\n", edata );
            }
            uWhen( NULL, &uAny, &edata ) {
                uFprintf( uStderr, "Unknown exception raised\n" );
            }
        } uEndExcept;
    }
}
```

Figure 6.1: Matrix Operation Example

```
uException except3 = U_EXCEPTION( &except1 ), except4 = U_EXCEPTION( &except1 );
uException except5 = U_EXCEPTION( &except4 );
```

creates the following hierarchical relationships among the exceptions (with indentation used to indicate derivation):

```
uAny
    except1
        except3
        except4
            except5
    except2
```

Here, if the exception `except5` is raised, it can be caught by a handler for any of `except5`, `except4`, `except1`, and `uAny`. Because every exception has to be derived from some other exception, all exceptions form a single hierarchy.

6.1.2.2 Exception Type

When an exception is raised, an exception instance, which contains the parameters of the exception, is passed from the raiser to the handler. The parameters allow the exception handler to analyze why the exception was raised or to print an appropriate error message. To simulate the derivation of exception types shown earlier, users must follow these conventions:

1. The name of the exception type must be the name of the exception with the suffix `Msg`. These types are used to create a pointer to or cast into the appropriate type the exception instance passed from the raise statement. If an exception provides no more parameters than its parent, it can use the same exception type as its parent. If an exception provides only a single value and has no derived exceptions, then it is possible to forego creating a special exception type structure. Here, the type of the single value is used directly instead of a structure containing it.
2. The type for a derived exception must contain as its first field an item with the type of its parent so that an instance of the derived exception can be passed to a handler for any parent exception.

For example, assume an exception `john` that has two integer values that are passed from the raiser to the handler. If the the exception `jonathan` inherits from exception `john` and adds an additional integer value, than it must ensure that its exception instance has the two values expected by a `john` exception at the beginning of the instance, as in:

```
uException john = U_EXCEPTION( &uAny ), jonathan = U_EXCEPTION( &john );

typedef struct {                |     typedef struct {
    int x, y;                   |        johnMsg base;   /* exception type of parent */
} johnMsg;                      |        float z;
                                |     } jonathanMsg;
```

Because the type for a parent exception does not include the fields of the type of a derived exception, information associated with the derived exception may be inaccessible in handlers for parents of the exception.

These conventions are used for all the predefined exception types that are discussed shortly and must be adhered to if user-defined exceptions are to work with the predefined exceptions and other user-defined exceptions.

6.1.3 Raising an Exception

The μSystem design supports two kinds of exceptions:

free exception is an exception that is not a component of a structure, as in:

```
uException fred = U_EXCEPTION( &uAny );   /* free exception */
```

associated exception is an exception that is associated with a structure, as in:

```
struct mary {
    uException john;   /* associated exception */
    ...
}
```

Each different instance of a structure containing an associated exception effectively defines a different exception.

A handler for a free exception catches any instance of that exception, while a handler for an associated exception catches an exception associated with a particular instance of the structure of which it is a component.

Associated exceptions affect the way that exceptions are implemented; free exceptions can be dealt with as a special case of associated exceptions. There are basically two approaches to implementing associated exceptions. Since associated exceptions are different for each instance of the data structure (class) they are associated with, the obvious approach is to treat the exception as a field of the data structure with which it is associated. However, this may be quite costly because there may be a substantial amount of storage used for exceptions in each instance and each exception field must may have to be initialized. For example, in our implementation, each file descriptor needs storage for 20 exceptions and these exceptions must be initialized. The other approach separates the data-structure instance and its exceptions. The associated exceptions are essentially defined to have `static` storage class so that they are instantiated only once. A raise statement then specifies two addresses: the address of the data-structure instance and the address of the associated exception. Correspondingly, in the handler, two addresses are specified to compare with the raise statement addresses; however, the exception address in the handler need only be a parent exception of the one specified in the raise statement. In this scheme, no storage is needed in each data-structure instance for associated exceptions. However, each raise statement and handler now has two pointer values. The assumption is that the number of data-structure instances with associated exceptions will be greater than the number of raise statements and handler clauses so that there will be a net saving in the amount of storage used. We have adopted the latter approach to implementing associated exceptions and provide free exceptions by using the null address for the associated data-structure. Hence, associated exceptions are simulated by making them into free exceptions with names that do not conflict with other free exceptions, possibly by prefixing the exception name with the structure name.

When an associated exception has essentially the same function as a free exception, only a single exception needs to be declared. The free and associated exceptions can be distinguished by using `null` or a data structure address in the raise statement and handler. This was done for all the I/O exceptions in the μSystem.

6.1.3.1 Raise Routine

The routine `uRaise` is used to raise an exception that affects flow of control in the current coroutine or task.

```
void uRaise( void * data-item-address, uException * exception-address,
             void * exception-type-instance-address, int exception-type-instance-length );
```

data-item-address is the address of the data structure that contains the associated exception or `NULL` if the exception is a free exception.

exception-address is the address of a `uException` name.

exception-type-instance-address is the address of an exception instance to be passed to the exception handler.

exception-type-instance-length is the size in bytes of the exception type instance to be passed to an exception handler. Usually this is just the `sizeof` the exception type unless the exception type is a variable-sized data structure, such as a character string.

An exception affects the current task's thread, terminating the active blocks that the thread is currently executing. In the μSystem, both coroutines and tasks have independent stacks of active blocks. When an exception terminates the last active block of a coroutine or task, the exception is handled by a default handler for exception `uAny`. This handler prints a specific message for the predefined exceptions, including all data passed from the raise point, and a generic message for other exceptions, including user-defined ones. Control never reaches the statement after a `uRaise`.

6.1.4 Handling an Exception

6.1.4.1 Exception Handler

An exception handler is defined using the macros `uExcept`, `uHandlers`, `uWhen`, and `uEndExcept` in the following manner:

```
uExcept {
      block in which the following exception handlers are active
} uHandlers {
    uWhen( ... )
         first exception handler code
    uWhen( ... )
         subsequent exception handler code
    ...
        ...
} uEndExcept;
```

The macro `uExcept` indicates the beginning of the block of code for which the exception handlers are active. The end of this block is indicated by the `uHandlers` macro. The exception handlers defined for a `uExcept` statement are active for the exception block and any nested blocks or routines called from within the exception block. The `uHandlers` macro is followed by a block containing the exception handlers. Each exception handler is introduced by the macro `uWhen`, which is followed by a (possibly compound) statement which is the body of the exception handler. The current implementation does not allow the use of `continue`, `break`, `goto` or `return` statements that would cause control to transfer out of the `uExcept` statement because the exception state will not be updated correctly. The macro `uEndExcept` terminates the block containing the exception handlers. `uExcept` statements may be nested, in both the exception and the handler blocks.

The `uWhen` macro specifies the information used in the raise search to find an appropriate handler.

```
uWhen( void * data-item-pointer, uException * exception-address, void * exception-type-instance-pointer )
```

data-item-pointer is the address of a data item that contains the associated exception or `NULL` if the exception is a free exception. The value `NULL` is used as the wildcard value, which matches with any data item. This address is *never* de-referenced.

exception-address is the address of a `uException` name. This may be the address of any user exception or predefined exception (e.g. `uAny`).

exception-type-instance-pointer is the address of a pointer into which the address of the exception type instance from the raiser is copied. This pointer is only valid within the statement after the `uWhen`.

When an exception is propagated to the `uExcept` block, the `uWhen` macros are searched in order from `uHandlers` to `uEndExcept`. Both of the following must be true for a `uWhen` to be selected:

1. The *data-item-pointer* specified in the `uRaise` must match the one in the `uWhen` or else the *data-item-pointer* in the `uWhen` must be `NULL`.

2. The *exception-address* specified in the `uRaise` must match the one in the `uWhen` or one of the exceptions from which it is derived.

Therefore, `uWhen` macros for derived exceptions must precede `uWhen` macros for exceptions from which they are derived, for example:

```
uExcept {
    uFgets( ..., f );           /* uSystem cover routine for fgets */
    uFgets( ..., g );           /* uSystem cover routine for fgets */
    ...
} uHandlers {
    uWhen( f, &uEofEx, ... )    /* catch f's end of file */
    uWhen( g, &uEofEx, ... )    /* catch g's end of file */
    uWhen( NULL, &uEofEx, ... ) /* catch any other end of file */
    uWhen( NULL, &uAny, ... )   /* catch all other exceptions */
} uEndExcept;
```

A `uWhen` macro with a `NULL` *data-item-pointer* and the address of exception `uAny` should be used only as the last exception handler in the block because handlers after it will never be selected. (Appendix A.5 shows a complete file merge program using exceptions.)

Within an exception handler, the predefined routines `uRaisedData()`, `uRaisedException()` and `uRaisedLength()` can be used to access the corresponding argument values used in a `uRaise`. This is useful in a handler, for example a `uAny` handler, to determine exactly which exception was raised and which data item it is associated with. A handler can re-raise an exception by using these routines, as in the following:

```
...
uWhen( NULL, &uAny, &msg ) {
    ...
    if ( uRaisedData() == &f && uRaisedException() == &UserExcept2 )
        uRaise( uRaisedData(), uRaisedException(), msg, uRaisedLength() );
}
...
```

6.1.5 Ensuring Correct Values in a Handler

With an optimizing compiler, there is an unavoidable problem arising from implementing exception handling using macros, i.e. not integrating them into the language. An optimizing compiler may hold the values of certain variables in registers for a series of statements and, in particular, for an entire `uExcept` construct. The problem occurs because it is not possible to locate and restore the registers saved by a call to a routine which raises an exception. Instead, register values are saved at the beginning of a `uExcept` statement and restored if a handler for that block is selected. Thus, if the value of a variable temporarily placed in a register is changed between the `uExcept` statement and the occurrence of an exception, an out of date (stale) value may be restored for it. This can result in unexpected behaviour if the program depends on the values of variables being current after an exception has occurred.

To eliminate this problem, variables that are modified in the exception block should be qualified with `volatile`. A variable with this qualification is assigned to a register only for very short durations, so even when exceptional control flow occurs, the variable always contains its current value. Figure 6.2 illustrates how to ensure that variables in a local scope have correct values after an exception is raised.

6.2 Predefined Exceptions

The μSystem provides a number of predefined exceptions; they are derived from one another as shown by the hierarchy in Figure 6.3. These exceptions are available globally and users can extend the hierarchy with their own exceptions by deriving them from the appropriate predefined exception.

```
uException UserExcept = U_EXCEPTION( &uAny );

void f( void ) {
    if ( error condition exists ) {
        char *data = "exception data";
        uRaise( NULL, &UserExcept, data, strlen( data ) + 1 );
    }
    ...
}

void uMain( int argc, char *argv[], char *envp[] ) {
    char *edata;
    volatile int i;         /* declared volatile */

    uExcept {
        for ( i = 0; i < N; i += 1 ) {
            f();   /* block in which exception handlers are active */
        }
    } uHandlers {
        uWhen( NULL, &UserExcept, &edata ) {
            /* The variable 'i' will have the current loop iteration */
            uFprintf( uStderr, "Exception raised in iteration %d, %s\n", i, edata );
        }
    } uEndExcept;
}
```

Figure 6.2: Obtaining Correct Values in a Handler

6.3 Data Items Associated with Predefined Exceptions

The exceptions `uActiveTasksEx`, `uEmitEx`, and `uCreateProcessorEx` are raised with the address of the cluster on which the operation is being executed. All other μKernel exceptions are raised with no associated data item.

All predefined I/O exceptions, i.e. `uIOEx` and below, are raised using the address of the file in which the exception occurred, except for `uCreateSockEx`, `uNoBufsEx`, and the exceptions under `uOpenEx`. These are raised using `NULL`, except for `uNoBufsEx` which will be raised using `NULL` from `uSocket`, and using the address of the socket from `uAccept` and `uGetsockname`. This allows a handler to catch exceptions specific to a particular file, when a file exists.

6.4 Predefined Exception Types

All μSystem predefined exception types follow the conventions stated earlier, that is, using the exception name and suffix `Msg` for the exception instance type, and including the parent's type, if any, as the first field.

6.4.1 μSystem Exception Type

The exception instance for the `uSystemEx` exception is as follows:

```
typedef char *uSystemExMsg;
```

which is a pointer to a character string that is a brief summary of why the exception was raised. This string can be printed by any exception handler.

```
uAny                         ; Global exception
  uSystemEx                    ; Errors in the uSystem
    uKernelEx                    ; Errors in the kernel
      uDataCommEx                  ; Errors in communication
        uDataCopyEx                  ; Errors in data copying
          uSendMsgTooLongEx            ; Sender's msg too long for receive area
          uReplyAreaTooShortEx         ; Sender's reply area too short for reply msg
          uForwardMsgTooLongEx         ; Forwarder's msg too long for receive area
          uAbsorbAreaTooShortEx        ; Absorber's reply area too short for die msg
          uSuspendMsgTooLongEx         ; Suspender's msg too long for resumed reply area
          uResumeMsgTooLongEx          ; Resumer's msg too long for resumed reply area
        uBadCoroutineEx              ; Resuming non-cocalled coroutine
        uSynchFailEx                 ; Synchronization failure
          uNotReplyBlockedEx           ; Replying to task not reply blocked
          uInvalidForwardEx            ; Cannot forward, not reply blocked or already forwarded
      uCreationEx                  ; Error when creating some uKernel entity.
        uCreateClusterEx             ; No memory or processor creation failure
        uEmitEx                      ; No memory to create task
        uCocallEx                    ; No memory to create coroutine
        uCreateProcessorEx           ; No memory, Bad arguments, or UNIX process creation failure
      uActiveTasksEx               ; Active tasks when destroying cluster
    uOutOfMemoryEx               ; Out of memory
      uNoExtendEx                  ; Cannot extend existing block
    uIOEx                        ; I/O system exceptions
      uEofEx                       ; I/O system end-of-file (uStream and uFile)
      uIOErrorEx                     ; Error exceptions for uFile and uStream
        uSocketErrorEx                 ; Error exceptions for socket operations
          uCreateSockEx                  ; Socket creation error
          uNotSockEx                     ; uFile must be a socket, but isn't
          uNoBufsEx                      ; No buffer space in kernel
          uBadSockAddressEx              ; Bad address for socket
            uConnFailedEx                  ; Socket connection failed due to bad address
        uFileStreamEx                  ; Error exceptions for uFiles and uStreams
          uOpenEx                        ; Error exceptions for the open operation
            uOpenIOFailedEx                ; A UNIX I/O operation failed
            uOpenNoSpaceEx                 ; No Space on the disk
            uBadPathEx                     ; Bad path
            uNoPermsEx                     ; No permissions for operation
            uNoFilesEx                     ; No more file descriptors
            uBadParmEx                     ; Bad parameter
          uReadWriteEx                   ; Error exceptions for read/write operations
            uIOFailedEx                    ; A UNIX I/O operation failed
            uNoSpaceEx                     ; No Space on the disk
            uBadFileEx                     ; Bad file descriptor
```

Figure 6.3: μSystem Predefined Exception Hierarchy

6.4.2 μKernel Exception Types

The exception instances for the μKernel exceptions are as follows:

```
typedef uSystemExMsg uKernelExMsg;

typedef struct {
    uSystemExMsg msg;
    void *sbuf;   /* address of send buffer */
    int  slen;    /* length of send buffer */
    void *rbuf;   /* address of receive buffer */
    int  rlen;    /* length of receive buffer */
} uDataCommExMsg, uDataCopyExMsg;

typedef struct {
    uDataCopyExMsg base;
    uTask sender;   /* task id of sending task */
    uTask receiver; /* task id of receiving task */
} uSendMsgTooLongExMsg;

typedef struct {
    uDataCopyExMsg base;
    uTask replier; /* task id of replying task */
    uTask sender;  /* task id of sending task */
} uReplyAreaTooShortExMsg;

typedef struct {
    uDataCopyExMsg base;
    uTask sender;    /* task id of sending task */
    uTask forwarder; /* task id of forwarding task */
    uTask receiver;  /* task id of receiving task */
} uForwardMsgTooLongExMsg;

typedef struct {
    uDataCopyExMsg base;
    uTask dier;     /* task id of dying task */
    uTask absorber; /* task id of absorbing task */
} uAbsorbAreaTooShortExMsg;

typedef struct {
    uDataCopyExMsg base;
    uCoroutine restarter; /* coroutine id of restarting coroutine */
    uCoroutine resumed;   /* coroutine id of resumed coroutine */
} uSuspendMsgTooLongExMsg, uResumeMsgTooLongExMsg;

typedef struct {
    uDataCommExMsg base;
    uCoroutine restarter; /* coroutine id of restarting coroutine */
    uTask thistask;       /* non-owner task that attempted restart */
} uBadCoroutineExMsg;

typedef struct {
    uDataCommExMsg base;
    uTask sender;   /* task id of sending task */
```

```
        uTask receiver; /* task id of receiving task */
    } uSyncFailExMsg, uNotReplyBlockedExMsg;

    typedef struct {
        uSyncFailExMsg base;
        uTask forwarder; /* task id of forwarding task */
    } uInvalidForwardExMsg;

    typedef uSystemExMsg uCreationExMsg;

    typedef struct {
        uCreationExMsg msg;
        uClusterVars cv; /* copy of the initial cluster defaults */
    } uCreateClusterExMsg;

    typedef struct {
        uCreationExMsg msg;
        int num_proc;  /* number of processors actually existing in cluster */
    } uCreateProcessorExMsg;

    typedef struct {
        uCreationExMsg msg;
        uCluster cluster; /* cluster task is to be created on */
        long space;       /* stack size for task */
        void *begin;      /* address of routine to be emitted */
        long arglen;      /* size in bytes of arguments to task */
    } uEmitExMsg;

    typedef struct {
        uCreationExMsg msg;
        void *rbuf;  /* address of receive buffer */
        int  rlen;   /* length of receive buffer */
        long space;  /* stack size for task */
        void *begin; /* address of routine to be cocalled */
        long arglen; /* size in bytes of arguments to coroutine */
    } uCocallExMsg;

    typedef uSystemExMsg uActiveTasksExMsg;
```

In the following contrived program example, an attempt is made to create a μSystem task with a specific stack size. If there is insufficient memory to create the task, the exception `uEmitEx` is raised. After catching the `uEmitEx` exception, the handler uses the information passed in the exception instance to determine what to do and possibly retry the operation. The extra exception `leave` and `uExcept` statement are necessary because that is the only possible way to exit the loop from the handler in the loop body, i.e. `break` cannot be used.

```
    uException leave = U_EXCEPTION( &uAny );
    uEmitExMsg *msgp;

    uExcept {
        for ( ;; ) {
            uExcept {
                tid = uEmit( ... );
                uRaise( NULL, &leave, NULL, 0 );    /* terminate loop */
            } uHandlers {
```

```
            uWhen( NULL, &uEmitEx, &msgp ) {
                if ( msgp->space > ... ) {
                    /* free storage or reduce emit stack size */
                } else {
                    uRaise( NULL, &leave, NULL, 0 ); /* terminate loop */
                } /* if */
            } /* uWhen */
        } uEndExcept;
    } /* for */
} uHandlers {
    uWhen( NULL, &leave, NULL );
} uEndExcept;
```

6.4.3 I/O Exception Types

The exception instances for the I/O exceptions are as follows:

```
typedef struct {
    uSystemExMsg msg;
} uIOExMsg;

typedef struct {
    uSystemExMsg msg;
} uEofExMsg;

typedef struct {
    uSystemExMsg msg;
    int errno;    /* UNIX errno */
} uIOErrorExMsg, uSocketErrorExMsg, uFileStreamExMsg,
  uNotSockExMsg, uNoBufsExMsg,
  uReadWriteExMsg, uIOFailedExMsg, uNoSpaceExMsg, uBadFileExMsg;

typedef struct {
    uSocketErrorExMsg base;
    int af;       /* address format for interpreting addresses in socket operations */
    int type;     /* type which indicates the semantics of communication */
    int protocol; /* protocol to be used with the socket */
} uCreateSockExMsg;

typedef struct {
    uSocketErrorExMsg base;
    void *name;   /* address of structure containing a server socket name */
    int  namelen; /* length of server socket name */
} uBadSockAddressExMsg, uConnFailedExMsg;

typedef struct {
    uFileStreamExMsg base;
    char *path;   /* path of file to be opened */
    char *perms;  /* permissions for opening of file (uStream) */
    int  flags;   /* access type for opening of file (uFile) */
    int  mode;    /* protection mode for file only (uFile) */
} uOpenExMsg, uOpenIOFailedExMsg, uOpenNoSpaceExMsg,
  uBadPathExMsg, uNoPermsExMsg, uNoFilesExMsg, uBadParmExMsg;
```

For example, after catching a uNoPermsEx exception, the following code examines one of the fields in the exception instance and possibly prints an error message before continuing execution.

```
uNoPermsExMsg *msgp;

uExcept {
    f = uFopen( ... );                  /* uSystem cover routine for fopen */
} uHandlers {
    uWhen( f, &uNoPermsEx, &msgp ) {  /* open failed because of bad permissions */
        if ( msgp->perms == ... ) {
            uFprintf( uStderr, "%s, errno:%d\n", msgp->base.msg, msgp->base.errno );
            uFprintf( uStderr, "...", msgp->path, msgp->perms, msgp->flags, msgp->mode );
        } /* if */
        f = uFopen( "/dev/null", ... ); /* corrective action */
    } /* uWhen */
} uEndExcept;
```

6.4.4 Memory Allocation Exception Types

The exception instances for the memory allocation exceptions are as follows:

```
typedef struct {
    uSystemExMsg msg;
    int size;     /* bytes of memory to be allocated or re-sized to */
} uOutOfMemoryExMsg;

typedef struct {
    uOutOfMemoryExMsg base;
    void *addr;   /* address of existing allocated area of memory */
} uNoExtendExMsg;
```

6.5 Interventions

Interventions are the common alternative to exceptions for dealing with abnormal events. As stated previously, the intervention model allows a routine to be called to effect a correction. This is like a PL/I on condition [Mac77]. This model provides for extensibility since a user can possibly deal with an abnormal event and continue execution without changing the existing routine in which the abnormal event occurred, e.g. dealing with a zero-divide error by returning zero or a large number. In essence, an intervention handler is a dynamically bound implicit routine argument that is called at the point of the abnormal event, as available in languages like Lisp.

In statically scoped languages, interventions are implemented using a separate stack for each intervention. When an intervention handler is installed, it is pushed on the top of the specified intervention's stack. When the intervention is called, the handler at the top of the stack is invoked. In statically typed programming language, the type of the handler on each stack is fixed when the intervention is created so that static type-checking is possible at the intervention call. The arguments that can be passed to an intervention routine and the result from the handler allow data at the intervention point to be passed to the handler and a corrective result to be returned if applicable.

6.5.1 Intervention Definition

6.5.1.1 Synchronous Intervention

A synchronous intervention is one that is invoked only from within a task. It is declared using the macro uIntervention, which specifies the intervention name and the type of a routine pointer that can be pushed on the intervention's logical stack, as in:

`uIntervention(` *intervention-name*`,` *pointer-to-routine-type* `);`

intervention-name is the name for the intervention. This is used when calling an intervention or installing an intervention routine.

pointer-to-routine-type is the type for the routines for this intervention. This must be a `typedef` for a routine pointer. This type is used for static type checking when calling an intervention or installing an intervention routine.

For example,

```
typedef int (*fredType)( int, float );
uIntervention( fred, fredType );
```

declares an intervention `fred`, which can have pointers to routines of type `fredType` associated with it.

An intervention declaration instantiates a data item, so to use an intervention from another compilation unit requires an `extern` declaration as with normal external data items in C. A special macro is required for an `extern` intervention declaration (unlike a `uException`, which uses the normal C syntax), as in:

`uInterExtern(` *intervention-name*`,` *pointer-to-routine-type* `);`

The parameters have the same meaning as for the `uIntervention` declaration. One of the compilation units in a program must make the actual intervention declaration.

6.5.1.2 Asynchronous Intervention

An asynchronous intervention is one that is activated from another task. It is declared using the macro `uAsyncIntervention`, which specifies only the intervention name, as in:

`uAsyncIntervention(` *intervention-name* `);`
`uAsyncInterExtern(` *intervention-name* `);`

intervention-name is the name for the intervention. This is used when calling an intervention or installing an intervention routine.

The type of a routine pointer that can be pushed on asynchronous intervention's logical stack is assumed to be:

`void (*)( void *` *message*`, int` *message-length* `);`

Therefore, an asynchronous intervention routine is restricted to passing a *message* of size *message-length*. It was not possible to use the argument-parameter mechanism between tasks due to implementation difficulties. However, it is possible to pass multiple values in a message so it is functionally equivalent to argument-parameter passing without the static type-checking.

6.5.2 Calling an Intervention

6.5.2.1 Synchronous Intervention

A synchronous intervention is called as follows:

`uInterCall(` *intervention-name* `)( ... );`

intervention-name is the name of the intervention to be called.

... is the list of arguments to intervention routine.

For example:

```
int f( ... ) {
    ...
    if ( ... ) i = uInterCall( fred )( 3, 4.5 );  /* static type checking */
    ...
}
```

calls the intervention routine at the top of `fred`'s intervention stack with arguments `3, 4.5`.

6.5.2.2 Asynchronous Intervention

An asynchronous intervention is activated as follows:

`uAsyncInterCall( uTask` *task-id*`,` *intervention-name*`, void *` *message-ptr*`, int` *message-length* `);`

task-id is the task indentifier of the task in which the intervention is to activated.

intervention-name is the intervention to be activated.

message-ptr is the address of a message to be sent to the activated intervention.

message-length is the size in bytes of the message to be sent.

For example:

```
int f( ... ) {
    ...
    if ( ... ) uAsyncInterCall( t, mary, msgptr, msglnth );
    ...
}
```

calls the intervention routine at the top of `mary`'s intervention stack in task `t` with message `msgptr`, which has size `msglnth`. Control continues immediately after the message has been delivered to the specified task.

6.5.3 Enabling and Disabling Asynchronous Calls

For a particular task, an asynchronous intervention is disabled until an intervention routine is established for it. Intervention activations remain pending until a routine is established. Once an intervention routine is established, any pending activations or new activations to that intervention are delivered as quickly as possible.

Explicit global disabling and enabling of all asynchronous interventions is provided for critical regions by routines `uAsyncInterEnable` and `uAsyncInterDisable`.

```
void uAsyncInterEnable( void );
void uAsyncInterDisable( void );
```

In general, asynchronous interventions should be disabled for the shortest possible time interval or tasks will not deal with them quickly.

6.5.4 Handling an Intervention

An intervention handler is established using the macros `uInter` and `uEndInter` in the following manner:

```
uInter( intervention-name, routine-pointer ) { /* push new routine on intervention stack */
                    /* block in which intervention may be called */
} uEndInter;        /* pop routine from the top of the intervention stack */
```

intervention-name is the name of the intervention on whose logical stack *routine-pointer* is pushed.

routine-pointer is the address of a routine that has the same type as that associated with *intervention-name.*

Intervention calls to *intervention-name* from within the block established by the `uInter` macro will execute the given routine, assuming another nested `uInter` block for that intervention is not opened. The `uEndInter` macro pops the routine from the top of the intervention stack. For example:

```
int fredCorrection( int x, float y ) ...

uInter( fred, fredCorrection ) {
    f( ... ); /* possible call to fredCorrection if there is an abnormal event in f */
} uEndInter;
```

establishes routine `fredCorrection` at the top of the intervention stack for intervention `fred`. If a synchronous intervention call is made to `fred` within `f`, routine `fredCorrection` will be invoked. Asynchronous interventions are established in the same way, only the routines that can be stacked for asynchronous interventions must have the predefined typed for message passing.

6.5.5 Asynchronous Intervention Activation

The task in which an asynchronous is activated only receives the intervention when it next installs an intervention routine, enables interventions, or is made active; if a task does none of these actions, it will never receive the activation (this requires that time-slicing be turned off for a computationally bound task).

Chapter 7

Miscellaneous

7.1 Profiling

When the `-profile` option is specified on the `concc` command, the user's program is loaded with a version of either the unikernel or multikernel that has been compiled with the `-p` flag. It also compiles the user's program with the `-p` option. This allows a complete profile of both the user and the μKernel code.

The `-p` option causes the C compiler to generate extra code to gather information about a program's execution. This code gathers for each external symbol, the percentage of time spent executing between that symbol and the next (usually a routine), together with the number of times that code fragment is transferred to (called for routines) and the number of milliseconds to execute it. This profile information is produced for each virtual processor (UNIX process) and is written to a file for subsequent analysis by the program `prof`. For multiprocessor programs, which involve multiple virtual processors, multiple profile files are produced. If these files are given as input to `prof`, the output represents the sum of the profiles across all the virtual processors during a program's execution.

The names of the files produced by the virtual processors are controlled by the value of the shell environment variable `PROFDIR`. If the variable `PROFDIR` does not exist, all profiling output is written to file `mon.out`. **So in the absence of this variable, only uniprocessor profiling works correctly.** If `PROFDIR` is set to an empty string, profiling is turned off, and no profile files are written. If `PROFDIR` is set to a non-empty string, the profile data is written to the file *string/pid.progname*, where *string* is the value of `PROFDIR`, *pid* is the UNIX process id of the virtual processor, and *progname* is the name of the program (taken from `argv[0]`). Note that the directory *string* must exist. For example, for a multiprocessor program named `prog` that uses three virtual processors and `PROFDIR=.`, the profile information is placed in files in the current directory whose names are of the form *nnnnn*`.prog`, where *nnnnn* is the UNIX process id of a virtual processor. After the files are produced, they can be analyzed as follows:

```
% prof prog 2345.prog 2346.prog 2348.prog
```

where the UNIX process ids for the three virtual processors were 2345, 2346 and 2348, respectively.

□ In the profile output there will be a routine called `uEndUserCode` that may have a significant amount of execution time associated with it. `uEndUserCode` is the first routine in the μKernel (see Section 7.5) and it is followed by a number of static kernel routines that perform all the scheduling and context switching. The profiler charges the execution of these kernel routines to `uEndUserCode`. If a program is performing a large number of μSystem routine calls, this may result in a large amount of execution time associated with routine `uEndUserCode`. □

7.2 Program Termination

To terminate a program other than returning from `uMain` and to return a status code to the invoking shell, call the free routine `uExit`:

void uExit(int status);

This routine is the same the UNIX routine `exit`, except that it works correctly in both unikernel and multikernel.

□ Currently, `uExit` does not work with profiling. □

7.3 Errors

Errors in the μSystem are divided into three categories:

- Normally, errors should be handled by raising exceptions, however if a catastrophic error is detected, execution must be aborted. The mandatory way to stop all execution while running within the μSystem is to call routine `uAbort`. The UNIX routines `exit` and `abort` are designed for single process programs and will not work as expected in the multikernel.

  The routine `uAbort` prints a user specified string which is presumably a message describing the error, and then prints the identity of the task (or its name, if present) calling the routine and the current value of the UNIX signal number.

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

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

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

- A μSystem exception is raised and no user handler catches it. The default handler prints an error message by calling `uAbort`. Examples of such errors are running out of memory or sending a message that is too long to be received.

- A user task executes some code that causes the virtual processor to fault. The death of the UNIX process will be caught by a task executing on the parent process of the terminating process. In general, this is a task in the system cluster, which calls routine `uAbort`. For example, if a task tries to divide by zero or access memory out of the address space currently available to the application, these errors will be trapped. In such situations, the UNIX signal number of the terminating process is displayed in the error message. Hence, when the μSystem displays a message saying that a UNIX process died, the cause of that UNIX process's death can be determined. The list of UNIX errors that may be reported as the result of processor death may be looked up in `/usr/include/signal.h`.

7.4 Symbolic Debugging

The symbolic debugging tools (e.g. `dbx`, `gdb`) do not necessarily work well with the μSystem. 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 will be understood by the debugger. Further, in the multiprocessor case, there are multiple UNIX processes that are not necessarily handled well by all debuggers. Nevertheless, it is possible to use many debuggers on programs compiled with the unikernel. At the very least, it is usually possible to examine some of the variables, externals and ones local to the current coroutine or task, and to discover the statement where the error occurred. The `gdb` debugger works well in uniprocessor form, but time-slicing must be turned off if breakpoints are to be used.

7.5 Pre-emptive Scheduling and Critical Sections

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

Our solution is to allow pre-emption only in user code. When a pre-emption occurs, the handler for the interrupt 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 the unikernel case, this means that μSystem cover routines like `uOpen` are not necessary; however, in the multikernel case `uOpen` is necessary to deal with the blocking I/O problem. To ensure portability between unikernel and multikernel, μSystem supplied cover routines, like `uOpen`, should always be used.

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

7.6 Monitoring Execution

When executing a multiprocessor μSystem 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 virtual processors and then opens 3 files. During execution of this program, called `a.out`, the following output fragment 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 wrap around to zero) is always the virtual processor for the system cluster. This virtual processor has an execution time 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 `uStdin` or `uStdout` or `uStderr`, then the system cluster will have 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), then the system may be deadlocked (however, they might also be blocked waiting for a terminal I/O operation to complete).

7.7 Installation Requirements

The μSystem runs on the following processors: M68000 series, NS32000 series, VAX, MIPS, Intel 386, Sparc, and the following UNIX operating systems:

- BSD 4.{2,3}
- UNIX System V that has BSD system calls `setitimer` and a `sigcontext` passed to signal handlers which contains the location of the interrupted program
- Apollo SR10 BSD
- Sun OS 4.x
- Tahoe BSD 4.3
- Ultrix 3.x/4.x
- DYNIX
- Umax 4.3
- IRIX 3.3

The uniprocessor μSystem runs on the following vendor's computers: DEC, Apollo, Sun, MIPS, Sequent and SGI. The multiprocessor μSystem runs on the following vendor's computers: Sequent Symmetry and Balance, Encore Multimax and SGI.

The μSystem requires at least GNU C 1.37.1 [Sta89]. For the Encore, GNU C version 1.37.1 requires patching *before* the uSystem can be compiled. All patches are supplied. This compiler supports both K&R C and ANSI C [KR88] (see `man gcc` for information) and can be obtained free of charge. The μSystem will NOT compile using other compilers due to the inline assembler statements that appear in the C machine dependent files and the use of structure constructors for initialization. The Sequent and Encore versions are setup so that GNU C always uses the vendors assembler because the GNU assembler does not handle the assembler directives generated from GNU C when the `-fshared-data` flag is used. This allows the uSystem to function even when GNU C is installed using the GNU assembler.

7.8 Reporting Problems

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

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

Appendix A

Example Programs

A.1 Coroutine Example

```
/* Producer−consumer problem, full coroutines */

#include <uSystem.h>

long random( void );

void Producer( uCoroutine *Cons, uCoroutine creator, int NoOfItems ) {
    int i, product;

    uSuspend( NULL, 0, NULL, 0 );                         /* wait for consumer to be created */
    uPrintf( "Producer will produce %d items for the consumer\n", NoOfItems );
    for ( i = 1; i <= NoOfItems; i += 1 ) {
        product = random() % 100 + 1;
        uPrintf( "Producer:  %d\n", product );
        uResume( *Cons, NULL, 0, &product, sizeof(product) );
    } /* for */
    product = −1;                                         /* terminal value */
    uResume( *Cons, NULL, 0, &product, sizeof(product) );  /* terminate consumer */
    uResumeDie( creator, NULL, 0 );                       /* restart creator */
} /* Producer */

void Consumer( uCoroutine *Prod ) {
    int product;

    uSuspend( &product, sizeof(product), NULL, 0 );       /* wait for producer */
    while ( product != −1 ) {
            uPrintf( "Consumer:  %2d\n", product );
            uResume( *Prod, &product, sizeof(product), NULL, 0 );
    } /* while */
} /* Consumer */

void uMain() {
    uCoroutine prod, cons;

    prod = uCocall( NULL, 0, Producer, &cons, uThisCoroutine(), 10 ); /* create producer */
    cons = uCocall( NULL, 0, Consumer, &prod );           /* create consumer */

    uResume( prod, NULL, 0, NULL, 0 );                    /* start producer */

    uPrintf( "successful completion\n" );
} /* uMain */
```

A.2 P/V Example

```
/* Producer and Consumer Problem using P/V with a Bounded Buffer */

#include <uSystem.h>
#define QueueSize 10

extern long int random( void );

struct shrqueue {
    int front, back;                                    /* position of front and back of queue */
    uSemaphore full, empty;                             /* synchronize for full and empty buffer */
    int queue[QueueSize];                               /* queue of integers */
}; /* shrqueue */

void Producer( struct shrqueue *q, int NoOfItems ) {
    int i, product;

    uPrintf( "Producer will produce %d items for the consumer\n", NoOfItems );
    for ( i = 1; i <= NoOfItems; i += 1 ) {             /* produce a number of items */
        product = random() % 100 + 1;                   /* generate random product */
        uPrintf( " Producer:  %2d\n", product );
        uP( &(q->empty) );                              /* wait if queue is full */
        q->queue[q->back] = product;                    /* insert element in queue */
        q->back = ( q->back + 1 ) % QueueSize;          /* increment back index */
        uV( &(q->full) );                               /* signal consumer */
    } /* for */
    product = -1;                                       /* terminal value */
    uP( &(q->empty) );                                  /* wait if queue is full */
    q->queue[q->back] = product;                        /* insert element in queue */
    q->back = ( q->back + 1 ) % QueueSize;              /* increment back index */
    uV( &(q->full) );                                   /* signal consumer */
    uDie( NULL, 0 );
} /* Producer */

void Consumer( struct shrqueue *q ) {
    int product;

    for ( ;; ) {
        uP( &(q->full) );                               /* wait for producer */
        product = q->queue[q->front];                   /* remove element from queue */
        q->front = ( q->front + 1 ) % QueueSize;        /* increment the front index */
        uV( &(q->empty) );                              /* signal empty queue space */
      if ( product < 0 ) break;
        uPrintf( "Consumer :  %2d\n", product );
    } /* for */
    uDie( NULL, 0 );
} /* Consumer */

void uMain( ) {
    struct shrqueue queue = { 0, 0, U_SEMAPHORE( 0 ), U_SEMAPHORE( QueueSize ) };

    uTask Prod = uEmit( Producer, &queue, 10 );         /* create producer */
    uTask Cons = uEmit( Consumer, &queue );             /* create consumer */

    uAbsorb( Prod, NULL, 0 );                           /* wait for completion */
    uAbsorb( Cons, NULL, 0 );
    uPrintf( "successful completion\n" );
} /* uMain */
```

A.3 Message Passing Example

```
/* Producer−consumer problem with send/receive/reply communication. */

#include <uSystem.h>

long random( void );

void Producer( uTask Cons, int NoOfItems ) {
    int i, product;

    uPrintf( "Producer will produce %d items for the consumer\n", NoOfItems );

    for ( i = 1; i <= NoOfItems; i += 1 ) {
        product = random() % 100 + 1;
        uPrintf( " Producer:  %2d\n", product );
        uSend( Cons, NULL, 0, &product, sizeof(product) );
    } /* for */
    product = −1;                                          /* terminal value */
    uSend( Cons, NULL, 0, &product, sizeof(product) );     /* terminate consumer */
    uDie( NULL, 0 );
} /* Producer */

void Consumer( void ) {
    int product;

    for ( ;; ) {
        uReply( uReceive( &product, sizeof(product) ), NULL, 0 );
      if ( product < 0 ) break;
        uPrintf( "Consumer :  %2d\n", product );
    } /* for */
    uDie( NULL, 0 );
} /* Consumer */

void uMain( ) {
    uTask Prod, Cons;

    Cons = uEmit( Consumer );                              /* create consumer */
    Prod = uEmit( Producer, Cons, 10 );                    /* create producer */

    uAbsorb( Prod, NULL, 0 );                              /* wait for completion */
    uAbsorb( Cons, NULL, 0 );

    uPrintf( "successful completion\n" );
} /* uMain */
```

A.4 Socket Example

A.4.1 Client Socket

```
#include <uSystem.h>
#include <uFile.h>
#include <sys/socket.h>
#include <sys/un.h>

void reader( uFile sd ) {
    int c;

    for ( ;; ) {
            uRead( sd, &c, sizeof( c ) );
      if ( c == EOF ) break;
            uPutc( c, uStdout );
    } /* for */
} /* reader */

void writer( uFile sd ) {
    int c;

    for ( ;; ) {
            c = uGetc( uStdin );
            uWrite( sd, &c, sizeof( c ) );
      if ( c == EOF ) break;
    } /* for */
} /* writer */

void uMain( int argc, char *argv[] ) {
    uFile sd;
    struct sockaddr_un server;
    void strcpy( char *, char * );
    uTask wr;
    uTask rd;

    switch ( argc ) {
      case 2:
        break;
      default:
        uAbort( "usage:  %s socket-name\n", argv[0] );
    } /* switch */

    sd = uSocket( AF_UNIX, SOCK_STREAM, 0 );             /* create a socket */
    if ( sd == NULL ) {
            uAbort( "Error while opening socket!\n" );
    } /* if */

    server.sun_family = AF_UNIX;                          /* specify socket domain */
    strcpy( server.sun_path, argv[1] );                   /* specify destination socket name */

    uConnect( sd, &server, sizeof( server ) );            /* connection to destination socket */

    wr = uEmit( writer, sd );                             /* emit a worker to read from file and write to socket */
    rd = uEmit( reader, sd );                             /* emit a worker to read from socket and write to file */
    uAbsorb( wr, NULL, 0 );                               /* absorb worker */
    uAbsorb( rd, NULL, 0 );                               /* absorb worker */

    uClose( sd );                                         /* close socket */
} /* uMain */

/* Local Variables: */
/* compile-command: "concc -quiet -O -o Client SocketClient.c" */
/* End: */
```

A.4.2 Server Socket

```
#include <uSystem.h>
#include <uFile.h>
#include <sys/socket.h>
#include <sys/un.h>

void Worker( uFile fd ) {
    int c;                                          /* used to store byte read from socket */

    for ( ;; ) {
            uRead( fd, &c, sizeof( c ) );           /* read byte from socket */
            uWrite( fd, &c, sizeof( c ) );          /* write byte to socket */
      if ( c == EOF ) break;                        /* no bytes left? */
    } /* for */
} /* Worker */

void uMain( int argc, char *argv[] ) {
    int c;
    uFile sd;
    uFile fd;
    uTask worker;
    struct sockaddr_un server;
    void strcpy( char *, char * );

    switch ( argc ) {
      case 2:
        break;
      default:
        uAbort( "usage:  %s socket-name\n", argv[0] );
        break;
    } /* switch */

    sd = uSocket( AF_UNIX, SOCK_STREAM, 0 );

    server.sun_family = AF_UNIX;
    strcpy( server.sun_path, argv[1] );
    uBind( sd, &server, sizeof( server ) );

    uListen( sd, 5 );

    for ( ;; ) {
        fd = uAccept( sd, NULL, NULL );             /* accept a connection */
            worker = uEmit( Worker, fd );           /* emit a worker to deal with the connection */
            uAbsorb( worker, NULL, 0 );
            uClose( fd );                           /* close the connection */
    } /* for */

    uClose( sd );                                   /* close socket */
} /* uMain */

/* Local Variables: */
/* compile-command: "concc -quiet -O -o Server SocketServer.c" */
/* End: */
```

A.5 File Example Using Exceptions

```
#include <uSystem.h>

/* Merge 2 files into a third file (assuming a high-key merge is not possible).
   End of file is detected using exceptions */

void uMain( int argc, char *argv[] ) {
    uStream in1, in2, out;
    int ch;
    uOpenExMsg *msg;
    uEofExMsg *eof;
    int len;
    char line1[256] = "", line2[256] = "";

    switch ( argc ) {
      case 3:
      case 4:
        uExcept {
            in1 = uFopen( argv[1], "r" );                    /* open input file */
            in2 = uFopen( argv[2], "r" );                    /* open input file */
        } uHandlers {
            uWhen( NULL, &uOpenEx, &msg ) {
                uAbort( "%sInput open error for file '%s'.\n", msg->base.msg, msg->path );
            } /* uWhen */
        } uEndExcept;
        if ( argc == 3 ) {                                   /* default output file */
            out = uStdout;
        } else {
            uExcept {
                out = uFopen( argv[3], "w" );                /* open output file */
            } uHandlers {
                uWhen( NULL, &uOpenEx, &msg ) {
                    uAbort( "%sOutput open error for file '%s'.\n", msg->base.msg, msg->path );
                } /* uWhen */
            } uEndExcept;
        } /* if */
        break;
      case 1:
      case 2:
      default:
        uAbort( "Usage:  %s infile1 infile2 [outfile]\n", argv[0] );
    } /* switch */

    uExcept {                                                /* merge input files */
        uFgets( line1, sizeof(line1), in1);                  /* try to get a line from each file */
        uFgets( line2, sizeof(line2), in2);

        for ( ;; ) {                                         /* output the smaller and attempt to replace it. */
            if ( strcmp( line1, line2 ) < 0 ) {
                uFprintf( out, "%s", line1 );
                uFgets( line1, sizeof(line1), in1 );
            } else {
                uFprintf( out, "%s", line2 );
                uFgets( line2, sizeof(line2), in2 );
            } /* if */
        } /* for */
    } uHandlers {
        uWhen ( in1, &uEofEx, &eof ) {                       /* dump out the other non-empty file */
            uExcept {
                for ( ;; ) {
                    uFprintf( out, "%s", line2 );
                    uFgets( line2, sizeof(line2), in2 );
                } /* for */
            } uHandlers {
                uWhen ( in2, &uEofEx, &eof );
            } uEndExcept;
        } /* uWhen */
        uWhen ( in2, &uEofEx, &eof ) {
            uExcept {
                for ( ;; ) {
                    uFprintf( out, "%s", line1 );
                    uFgets( line1, sizeof(line1), in1 );
                } /* for */
            } uHandlers {
                uWhen ( in1, &uEofEx, &eof );
            } uEndExcept;
        } /* uWhen */
    } uEndExcept;

    uFclose( in1 ); uFclose( in2 );                     /* close input file */
    uFclose( out );                                     /* close output file */
} /* uMain */

/* Local Variables: */
/* compile-command: "concc -work -nounixrc -O File.c" */
/* End: */
```

Bibliography

[BMZ] Peter A. Buhr, Hamish I. Macdonald, and C. Robert Zarnke. Synchronous and Asynchronous Handling of Abnormal Events in the μSystem. submitted to Software–Practice and Experience.

[BS90] Peter A. Buhr and Richard A. Stroobosscher. The μSystem: Providing Light-Weight Concurrency on Shared-Memory Multiprocessor Computers Running UNIX. *Software–Practice and Experience*, 20(9):929–963, September 1990.

[Che82] D. R. Cheriton. *The Thoth System: Multi-Process Structuring and Portability.* American Elsevier, 1982.

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

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

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall Software Series. Prentice Hall, second edition, 1988.

[LA90] P. A. Lee and T. Anderson. *Fault Tolerance - Principles and Practice*, volume 3 of *Dependable Computing and Fault-Tolerant Systems.* Springer-Verlag, 2nd revised edition, 1990.

[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.

[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.

[Sta89] Richard Stallman. The Free Software Foundation's Gnu C Compiler. Free Software Foundation, 1000 Mass Ave., Cambridge, MA, U. S. A., 02138, 1989.

Index