Synchronous and Asynchronous Handling of Abnormal Events in the μSystem

Peter A. Buhr*, Hamish I. Macdonald*, C. Robert Zarnke**

* Dept. of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1
(519) 888-4453, Fax: (519) 885-1208, E-mail: pabuhr@watmsg.uwaterloo.ca

** Hayes Canada Inc., 175 Columbia St. W., Waterloo, Ontario, Canada, N2L 5Z5

SUMMARY

This paper presents a general model for dealing with abnormal events during program execution and describes how this model is implemented in the μSystem. (The μSystem is a library of C definitions that provide light-weight concurrency on uniprocessor and multiprocessor computers running the UNIX operating system.) Two different techniques can be used to deal with an abnormal event: an exception, which results in an exceptional change in control flow from the point of the abnormal event; and an intervention, which is a routine call from the point of the abnormal event that performs some corrective action. 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 program. 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.

KEY WORDS Abnormal Event, Exception Handling, Concurrency, Interrupts

Accepted for publication in Software–Practice and Experience, 1992

INTRODUCTION

This paper presents a general model for dealing with abnormal events during program execution and describes how this model is implemented in the μSystem. The μSystem [1] is a library of C [2] definitions that provide light-weight concurrency on uniprocessor and multiprocessor computers running the UNIX* operating system.

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 [3, 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. As above, 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 are discussed in detail. Thus, there are two possible outcomes of an operation: normal completion possibly returning a value, or failure with a change in control flow.

The case where an operation returns a special value to indicate an abnormal event, i.e. a **return code**, is not an actual failure 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 by our definition 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 (e.g. tested for successful completion). 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. In many cases this is not done, resulting in a secondary error far from the actual error.

This paper is divided into two parts. The first part gives an overview of abnormal event facilities and discusses some additional facilities that we feel are necessary. A set of "necessary" facilities are used to compare a number of different abnormal event mechanisms. This part should interest programming language designers as it discusses many design issues for abnormal event facilities, in particular, abnormal event facilities for concurrent systems. The second part describes an almost complete implementation of the ideas presented in the first part. The implementation is written using the C preprocessor and C routines. No compiler support was used so that several trade-offs were necessary. In particular, the syntax for some of the constructs is rather low level and is sometimes baroque. This part should interest programming language implementors as it discusses the issues in the implementation of a sophisticated abnormal event scheme for a highly concurrent environment.

Exceptions

The programmatic mechanism that indicates failure is called an **exception**, and when an exception is **raised**, it causes control to transfer to (not invoke) 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 with which it is associated; this scope might be an expression, a statement or a block.

When an operation fails and control transfers to a particular handler, the failing computation is not completed. For example, if the handler is associated with an expression, the failing operation and any

*UNIX is a registered trademark of AT&T Bell Laboratories 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** [4, 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. Ensuring the handler performs adequate forward recovery could possibly be enforced through post-condition verification by a theorem prover or in an ad hoc manner by a user specified acceptance test at the end of the handler [5].

Interventions

The invocation of a dynamically determined correction routine is called an **intervention**. An intervention routine might be specified by making the routine a parameter (perhaps an implicit one) of the operation that needs it, but this would be inconvenient and is not **extensible**; that is, a user could not add an intervention parameter without changing the original operation definition. Interventions can also be handled by constructs similar to those dealing with exceptions [6], but it is different from an exception because an intervention usually returns to the point of the abnormal event with a value. This is discussed in detail shortly.

It is desirable to allow an intervention to be activated by another task; this is called an **asynchronous intervention**. Such an asynchronous event is like a UNIX signal or system interrupt routine. Normally, an intervention affects only the current task, so no synchronization with other tasks is needed. However, for an asynchronous intervention, some implicit synchronization may occur in the other task before the intervention is activated.

ABNORMAL EVENT MODELS

The seminal work on different forms of abnormal event handling was presented by Goodenough in Reference [6]. A brief analysis of the different forms is presented, with examples given in a C++-like pseudo code.

Exceptions

As stated previously, an exception allows a transfer of control to a dynamically determined location. In essence, the effect of an exception is a goto restricted in the following two ways. First, an exception cannot be used to create a loop (i.e. cause a backward branch in the program). This means that only the looping constructs can be used to create loops. Second, since an exception always transfers out of containing control structures, it cannot be used to branch into a control structure. These restrictions are sufficient to preclude the major problems associated with an unrestricted goto statement [7].

Exception Naming

In general, an exception has a name. In Eiffel, there are only names for the predefined exceptions. Other exceptions arise implicitly through violations of pre-conditions and post-conditions or pre-defined hardware and operating system exceptions. Not having explicit names for exceptions makes it impossible to know which exception was raised, unless it is one of the predefined ones, and hence, why a computation failed, as many operations can fail for a number of reasons.

Exception names can be specified by declarations, as in:

```
exception X, Y
```

which create two named exceptions, X and Y; exceptions might also be associated with routine declarations by listing the exceptions that can be raised in that routine, as in:

```
int f( int x, int y ) exception( X, Y )
```

which states that exceptions X and Y may be raised in routine f.

The scope of an exception name is also important; two approaches are examined. The first approach is a single name space for all exceptions. In this approach, the name of the exception must be unique system-wide. This means that regardless of where a particular exception name is used in the system it always identifies the same exception. The second approach uses the block structure of the programming language in which the exception mechanism is embedded to allow exceptions with the same name to be different. In this approach, the name of the exception is unique only within a particular scope.

We believe a scoped name space is superior to a single name space. A single name space increases the potential for inadvertent collisions of names. For example, if a system routine is updated to raise an exception, that exception might have the same name as a handler in user programs. These programs might inadvertently attempt to handle the new system exception.

One implementation of this approach is to use the character string representation of the exception or a system-wide unique numbering scheme. Unfortunately, constructing system-wide unique names or numbers for each exception is difficult with separate compilation and in distributed environments. An integrated program development environment, where information about all compilations is retained, could provide this capability.

A practical scheme for many existing program development environments is to adopt the same approach used to uniquely identify external names. That is, the compiler assigns addresses to exceptions by generating storage for them and the linker resolve references to exceptions in different compilation units as it does for other such names. These addresses can then be used as unique identifiers in locating handlers. This implementation can be used for both the single and scoped approach for naming exceptions. As well, if storage is allocated for an exception, it might be used to contain other information, such as exception inheritance information (discussed shortly).

Derived Exceptions

Some operations, such as opening a file, can potentially fail for a large number of reasons. This could be dealt with either by having a single generic open-file exception with a parameter giving the precise reason, or by having an exception for each kind of failure. If a generic failure is to be dealt with, the first form is most convenient; whereas if a specific failure is to be dealt with, the second scheme is most convenient. Both can be provided for if one exception can be derived from another [8, 9]. This could be specified as follows:

```
exception jonathan : john           // indicating that exception jonathan is derived from john
```

The effect is that an exception handler for john catches jonathan exceptions but not vice versa. This capability for derivation is used extensively in the definition of μSystem exceptions as will be seen in the second part of the paper. An exception that is not explicitly derived from another is treated as being derived from the special builtin exception any, so that:

```
exception john
```

is treated as:

```
exception john : any
```

where any is essentially defined as:

```
exception any
```

Consequently, all exceptions in a program form a hierarchy whose root node is any. This permits any exception in a program to be caught by an exception handler for any.

Scope of an Exception Handler

There are several different models for associating a handler with the particular computation to which it applies. For example, to associate handlers with a particular expression [6], a construct such as the following might be used:

```
d = (a + b)[ X: ... , Y: ... ] / c
```

where the scope of the handlers for exceptions X and Y in the brackets [] is the expression (a + b). Within the brackets and after each colon is a handler body, which contains appropriate recovery code. In this example, both handlers must return a value for the failing expression so that the expression computation can continue. Alternatively, if handlers can only be associated with a series of statements [8], a construct such as the following might be used:

```
except {
    // block in which the following handlers are active
  handler( X )
    // recovery code for handler X
  handler( Y )
    // recovery code for handler Y
}
```

where the except statement introduces a block which defines the scope for the specified handlers.

We adopt the second form, even though its granularity of association is not as fine as in the first form. For example, to express the first example using the second construct requires the following:

```
except {
    temp = a + b
  handler( X )
    temp = 0        // recovery action
  handler( Y )
    temp = 1        // recovery action
}
d = temp / c
```

This decision is based on our experience that the need for fine grained handling is rare. As well, having handlers, which may contain arbitrarily complex blocks of code, in the middle of an expression can be difficult to read; we prefer the scope for handlers to be syntactically obvious and not hidden within expressions. Finally, when used in an expression, the handler must have a mechanism to return results to allow execution of the expression to continue; this usually requires additional programming language constructs.

Raising an Exception

All exceptions are raised, some implicitly by the runtime environment and others explicitly by users, through a raise statement, as in:

```
raise X
```

which transfers control to a handler for the exception X terminating all active blocks between the raiser and the handler (supported in Ada [10], ML [11], C++ [12], CLU [13], Mesa [14], Eiffel). Eiffel does not have an explicit raise statement because all exceptions are raised implicitly; users have only indirect access to the exception mechanism through assertions.

Multilevel versus Single-level Model

When an exception is raised it can be propagated to active blocks in one of two ways:

Multilevel model – the exception is propagated through the active blocks in reverse order until it is caught by a handler (as in Ada, Mesa, ML, C++ Eiffel). If there is no handler, the exception is caught by a general exception handler, which is defined implicitly in the first block (preamble) for a program, and which usually terminates the program.

Single-level model – the exception is propagated to the previous active block, and if not handled there, it results in program termination or it is transformed to the "failure" exception, which is then propagated through the active blocks in reverse order until it is caught (as in CLU and proposed in References [5, 6]).

Since CLU is the only programming language that we are aware of that implements the single-level model, we quote their justification for adopting it:

> ... each procedure implements a mapping. The caller of a procedure invokes the procedure to have the mapping performed; the caller need know only what the mapping is, and not how the procedure implements the mapping. Thus, while it is appropriate for the caller to know about the exceptions signaled by the procedure (and these are part of the abstraction implemented by that procedure), the caller should know nothing about the exceptions signaled by procedures used in the implementation of the invoked procedure. [15, p. 547]

To check this, either statically or dynamically, requires that all exceptions that can be raised by a routine must be explicitly specified as part of a routine's header, for example:

```
int f( int x, int y ) exception( X, Y )
```

which states that exceptions X and Y may be raised in routine f. It also requires that the caller handle all exceptions that might be raised; for routine f, each call would have to be embedded in a block containing appropriate handles, as in:

```
except {
    ...
    f( ... );                // block in which handlers are active
    ...
  handler( X ) ...          // handler for exception X
  handler( Y ) ...          // handler for exception Y
}
```

If a caller does not handle an exception, it is considered to be an error or it is transformed to the "failure" exception with a loss of knowledge about the exact exception raised and possibly data passed from the raise. If sufficient information is available, the lack of a particular handler can be detected at compile time; otherwise it requires dynamic checking, for example, in the case of incomplete type information among separate compilation units.

An intermediate model between single-level and multilevel is suggested in Reference [9]. As for the single-level model, an exception can be propagated outside of a terminating block if the the exception is specified as part of the routine's header. However, if the exception is not part of the routine's header it may be transformed from the specific to the general. The transformation makes use of an exception hierarchy to promote the specific exception to one of the exceptions specified in the routine's header that subsumes it. Thus, a specific non-handled exception is not always transformed to the "failure" exception or causes termination.

We reject both the single-level and the intermediate model as being unnecessarily cautious and restrictive, and choose instead the multilevel model because it is simpler to use and more extensible. First, we disagree with the fundamental premise that "each procedure implements a mapping". This is true only when a routine is self-contained. Many routines are used for structuring a program; that is, code is moved into a routine to avoid replication or for organizational reasons. When this is done, there is no longer a guarantee that all exceptions produced by the mapping can be raised in the "root" routine of the mapping; exceptions may now be raised in the structural routines and may be several call levels from the root module. In other words, the mapping is implemented as a set of cooperating routines, any of which might raise an exception to indicate failure of the mapping. For example, the end-of-file exception is often raised several call levels from a user's call to the read routine because the input abstraction is subdivided into a number of submodules with the lowest level routines detecting the end-of-file. Second, we doubt that programmers are willing to code a (potentially large) number of handlers for every routine call when most of them will be empty and when the programmer has decided to handle the exception at a higher level. (The CLU designers agreed with this point and do not require handlers for every exception that might be raised by a call in a routine.) Third, augmenting a routine with new exceptions may require that handlers be added to every routine that contains a call to the augmented routine, to ensure that all exceptions are handled. We believe that exceptions should be like default parameters, which can be added without directly or indirectly invalidating routines that call another routine. Fourth, in an environment where routines are commonly passed as arguments, it is not possible to anticipate all possible exceptions that might be raised when a routine argument is invoked, as in:

```
int h(int (*p)(...)) {          // p is a routine parameter
    ...
    *p( ... )                   // invoke routine parameter which might raise an exception
    ...
}

int g( ... ) exception( X ) { ... }

int f( ... ) {
    except {
        h( g )       // pass routine g as argument which might raise exception X
      handler( X ) ...
    }
}
```

If the definition of h is global, e.g. in a library, it generally cannot know all the exceptions that might be raised by all the different routines passed as arguments. If static checking is used to detect missing handlers, g cannot be passed as an argument to h, and consequently, reusability is restricted. If dynamic checking is used to detect missing handlers, the exception X will generate an error or be transformed into a more general exception, neither of which is useful to the implementor of routines g and f. Also, this usage does not violate the abstraction of h because the user of h is supplying the part of the implementation that raises exception X. We believe that environments where routines are passed as arguments are becoming common. For example, the functional programming style, as in Scheme [16] and ML, results in many routines being passed as arguments. As well, modern type systems may implicitly require routine parameters or variables, as in parametric polymorphic type systems (e.g. implicit routine parameters) [17] and object-oriented type systems (e.g. virtual members).

Therefore, the multilevel model was chosen because it does not transform an exception during the search for the handler, and as with the single-level model, it allows incremental program construction as it is possible to add an exception to a routine without having to change any of the routines that call it. In essence, we believe that exceptions are attributes of a routine but are not part of its type.

Matching an Exception to a Handler

During the raising of an exception, the exception specified in the raise statement has to be matched with a handler. The type equivalence mechanism of a programming language affects the search for a handler.

For the single-level model, all checking is done at compile time and the search can be as simple as an indirect jump to the appropriate handler because the handler can be determined statically at the raise. For the multilevel model, the search is more complex because the handler cannot be determined statically and certain type equivalence mechanisms require runtime type checking during this search.

Fundamentally, the first aspect of the handler search is to locate a list of handlers associated with an enclosing block and compare for equivalence the exception specified in the raise with the ones specified in the handlers. For type systems that use name equivalence, the name of the raised exception must match the name specified at the handler. For type systems that use structural equivalence, the structure of the raised exception must match the structure of that specified at the handler. Unfortunately, this makes all exceptions without parameters equivalent, which is highly undesirable; furthermore, the multilevel model requires dynamic type checking to match exceptions with parameters. These drawbacks explain why there are no programming languages that use structural equivalence for exception matching.

For type systems that support inheritance, the inheritance information is used at runtime to either convert exceptions or affect the handler matching. In the single and intermediate level models, an exception may be transformed from a lower level to a higher level if the caller only has a handler for the higher-level exception. But transformations from the specific to the general lose information. In the multi-level model, the inheritance information is used during matching to determine if the raised exception is the same as or derived from the exception in the handler. However, the raised exception is never transformed during the handler search.

In the C++ exception mechanism, there is no special exception type; any type can be used as an exception type. This seems to provide an additional level of generality. However, we doubt that users will create types that are used both in normal computations and for raising exceptions. Users will normally create specific types that describe exceptions, e.g. overflow, underflow. Therefore, having a specific exception type is not really a restriction, and it provides additional documentation and possibly discrimination among conventional and exception types.

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 [18]. 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 routine that is called at the point of the abnormal event, as available in languages like Lisp.

In a statically scoped language, each intervention is implemented using a separate stack. 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 a statically typed programming language, the type of the intervention is fixed when it 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.

Intervention Naming

An intervention must be explicitly named and specify the types of the arguments passed to it and the result returned from it. For example, the intervention declaration:

```
intervention int fred( float, int )
```

declares intervention fred whose handlers take float and int arguments and return an int result.

Calling an Intervention

An intervention is invoked by using the intervention name, as in:

```
i = fred( 3.5, 5 )
```

which invokes the routine at the top of the intervention stack.

Scope of an Intervention Handler

Like exceptions, intervention handlers could be associated with specific operations or blocks containing the operation and we prefer the latter, giving:

```
inter {
    // block in which the following handlers are active
  handler fred( x, y )        // push handler on fred stack, x and y are formal in−parameters
    return x + y              // body of handler fred, whose result type is specified by fred
  handler mary( ... )         // push handler body on mary stack
    ...                       // body of handler mary, whose result type is specified by mary
} // pop handlers off fred and mary stack
```

Each handler defines the body of a routine with the specified parameters in the scope of the handler clause of the inter statement. Pointers to these handler routines are pushed on the stack of the corresponding intervention.

Exceptions versus Interventions

In a programming language that provides no abnormal event constructs, it is possible to simulate both the exception and the intervention models. Simulating the exception model requires the setting of global flag variables and their testing following any call that might set them (likely many calls). Simulating the intervention model requires adding intervention parameters to all routine definitions that require them or that call routines that need them, or by explicitly building the stacks described previously. Both simulations violate encapsulation and extensibility besides being tedious and error-prone to implement. Thus, it is desirable to have at least one of these models provided in the programming language. However, it is not possible to use either model to simulate the other without using the techniques just described. Therefore, both models are desirable in a programming language.

Resumption Model

A combination of exception and intervention mechanisms for dealing with abnormal events is usually called the **resumption model** [6] (supported in Mesa, Eiffel). In the resumption model, the exception mechanism is used to call a handler, and the handler is allowed to decide whether to terminate or resume the caller. This model can be simulated with the intervention and exception models by changing each resumption exception into an intervention and possibly an exception. The intervention is invoked when the abnormal event is detected, and the intervention can deal with the problem or raise an appropriate exception. The resumption model presumes that it is occasionally desirable to have the intervention routine make the decision to terminate or continue. However, we believe that these situations are rare, and hence, having to simulate the resumption model in those situations where it is needed will not be a significant impediment to programmers.

DESIGN REQUIREMENTS

Having selected certain models for dealing with abnormal events, there were still important requirements that directed this work:

Exceptions

Exception Communication

We believe the ability to pass exception-specific data from the raiser to the handler is essential, otherwise the handler cannot analyze why the exception was raised or print an informative error message. Anonymous exceptions, as in Eiffel, cannot support exception-specific data. The design presented thus far is easily extended to allow communication, as in:

```
exception john {
   int x, y                   // data to be communicated
}
```

which defines an exception, john, in the current scope with fields, x and y, that permit data to be communicated from the raise statement to the handler. When the exception john is to be raised, an instance of the type defined by john is created which holds the values of the exception data, x and y; this instance is then passed to the appropriate exception handler, for example:

```
except {
    ...
        raise john( 3, 5 )  // john is both the exception raised and the type for the exception instance
    ...
  handler( john z )         // z is a formal in–parameter of type john
    print( z.x, z.y )
}
```

In this form, the exception instance is an object.

In ML, an exception declaration, such as:

```
exception john of int * int;
```

defines a routine called john that takes a pair of integers as its parameters. The declaration conceptually returns an exception instance containing a unique identifier for john and the two int arguments. The raise statement invokes this routine to get an exception instance, and uses the resulting exception instance to find the handler and deliver the exception's arguments.

Our suggestion for implementing an exception using a storage address is directly analogous to the ML form—the routine is located at a unique address which identifies the exception and the type of the routine defines what values are needed to instantiate the exception. In both cases, there are two essential properties: a unique identification and a type. It is the exception type's storage or ML's exception routine that provides the identity for the exception and possibly other necessary exception information. It is the type part that is used to allow statically type-safe communication between the raise statement and the handler.

A derived exception can have fields in addition to those provided by its parent, as in:

```
exception jonathan : john { float z }

except {
    ...
        raise john( 1, 2 )
        ...
        raise jonathan( 1, 2, 3.6 )
    ...
  handler( john w )
    ...     // w is an instance of john so it only has fields x and y
}
```

The exception jonathan has three fields, the first two inherited from exception john. Although the exception handler for john catches exception jonathan, it would not have access to parameter z of a jonathan exception instance.

Several object-oriented programming languages support multiple inheritance among class definitions. This suggests that multiple inheritance could be applied to exception definitions; however, defining the order of the arguments for an instance of such an exception would be syntactically difficult. While multiple inheritance allows even more complex relations among exceptions, this can substantially increase execution cost. As is being discovered, multiple inheritance is not as useful a mechanism as it initially seemed [19, 20].

Free versus Bound Exception

A **free exception** is one that is not a component of a structure or class, while a **bound exception** is one that is. Each different instance of a structure or class containing a bound exception effectively defines a different exception. A handler for a free exception catches any instance of that exception, while a handler for a bound exception catches an exception associated with a particular instance of the structure or class of which it is a component, for example:

```
exception fred              // free exception

class mary
   exception john           // bound exception
   ...

mary m

except {
   ...
      raise fred            // raise free exception
   ...
      raise m.john          // raise bound exception
   ...
  handler( fred ) ...       // catches free exception
  handler( m.john ) ...     // catches bound exception
}
```

If a user wanted to have handlers for the free exception fred to also catch an associated john exception, the bound exception would have to be derived from the free exception, as in:

```
exception fred

class mary {
   exception john : fred
   ...
```

It is possible to simulate bound exceptions using free exceptions, as shown in Figure 1; however, we believe it makes programs clearer and more extensible if bound exceptions are used. In the left example of Figure 1, the free exception john is caught and the handler must subsequently discriminate among the various data items that could have raised the exception. In the right example, there is a separate handler for each different john. In the left example, the default clause that re-raises the exception is, in general, required. If this convention is not followed, a john exception with an unenumerated data item would be caught and ignored instead of being propagated to the next active block. Following such a convention is unnecessary in the right example because any john exceptions associated with objects other than those listed are automatically propagated because they are not caught. Hence, the coding style on the right is inherently extensible.

Finally, the use of bound exceptions mitigates accidentally catching unanticipated exceptions. For example, assume that a free exception, ioerror, is raised when an I/O operation fails. If a routine is modified to use a file but a handler for ioerror is forgotten, an ioerror exception raised in the use of the new file would be caught accidentally by the caller of the routine if the caller has an ioerror handler. Alternatively, if the ioerror exception is bound to the file descriptor for an open file, the caller catchs only ioerror exceptions for its specific files.

Exception Variables

Currently, all exception instances have been created by the raise statement, but the design does not preclude an exception variable, which might be declared and subsequently used in a raise statement. For example, the exception variable f has different exception instances assigned to it:

```
class mary
    ...

exception john {  // free exception
    mary m
}
mary x, y, z

except {
    ...
        raise john(x)  // x parameter
        ...
        raise john(y)  // y parameter
        ...
        raise john(z)  // z parameter
    ...
  handler( john m )  // catch john
    case m  // discriminate
      x: ...
      y: ...
      z: ...
      default:
         raise john(m) // re−raise
}
```

```
class mary
    exception john  // bound exception
    ...



mary x, y, z

except {
    ...
       raise x.john  // x's john exception
       ...
       raise y.john  // y's john exception
       ...
       raise z.john  // z's john exception
     ...

   handler( x.john ) ...  // discriminate
   handler( y.john ) ...
   handler( z.john ) ...
}
```

Figure 1: Free versus Bound Exception

```
exception fred { int m }

fred f = fred( 3 )            // f is an exception variable of type fred
...
   raise f                    // raise an exception fred with value 3
...
f = fred( 5 )
...
   raise f                    // raise an exception fred with value 5
...
```

Exception variables only seem useful when they there is data to be communicated. Furthermore, we do not recognize many situations where this facility is particularly useful. This might result from our inexperience in writing large complex programs using exceptions.

Interventions

Intervention Definition and Communication

In essence, an intervention is a stack data structure that is polymorphic in the routine pointers that can be pushed onto the stack. Thus, an intervention could be considered an instance of this data structure. Like exceptions, the ability to pass intervention specific data from the call to the handler is essential. Further, the ability to return a result is necessary in most cases so that an expression containing a failing operation can continue. This is handled using the normal routine call mechanism.

Derived Interventions

Interventions are like routines while exceptions are like classes. Deriving interventions from other interventions does not seem desirable nor is routine derivation obvious because of contra-variance [21].

Free versus Bound Intervention

Interventions are simply dynamically bound routines, somewhat like virtual routines in a class-based programming language, but they can change during the life-time of an object. Having interventions as fields in a data structure can be useful [22], as with exceptions. For example, the bound intervention mary in:

```
class fred {
   intervention int mary( float );
   ...
}

fred f;

inter {
     // use f
   handler f.mary( x )          // push handler body on f.mary stack
     // intervention body
}
```

allows the same extensibility as does a bound exception.

Intervention Variables

Because an intervention is a function type, it cannot be used to create variables. At best it could be used to create pointers to interventions.

Asynchronous Events

An asynchronous event is an event activated in one task by a concurrent task or activity (e.g. a hardware device). In analogy to normal (synchronous) exceptions, we define an asynchronous exception to be an exception caused by an asynchronous event; similarly, an asynchronous intervention is an intervention caused by an asynchronous event. Both asynchronous exceptions and interventions are examined.

Asynchronous exceptions and interventions are mechanisms for interacting with a task that might currently not be listening because it is computing or blocked waiting for synchronous communication to complete. For example, pressing the interrupt key might be handled by a keyboard manager task by raising an asynchronous event in the task currently using the terminal screen and/or keyboard. This might cause the task to terminate its output and reset itself for a new input command. Another example is a group of tasks that is collectively solving a problem and the successful completion of one task sensibly causes the other tasks in the group to stop further computation. This would be appropriate, for example, if the tasks are collectively searching a database for some key. In order to provide for this capability, the successful task must have a means of interrupting the other tasks to terminate their computation and reset to some known state in possible preparation for a new search or to shutdown.

Regardless of the approach taken, any asynchronous event mechanism must cope with the following:

- A task needs to be able to control when asynchronous events are handled.

  Asynchronous exceptions or interventions occur at unpredictable times. For example, a task initially may need to establish a handler or intervention routine before it deals with any asynchronous event; otherwise, the event mechanism would be useless because no assumption can be made about the order or speed of execution of tasks, i.e. the asynchronous event could arrive before the code to deal with it is activated. (Requiring an initialization protocol to deal with this problem is not an acceptable or a possible solution in many cases.) This and other situations during the execution of a task require that asynchronous exceptions or interventions be able to be disabled and enabled. Further, a user may disable asynchronous exceptions or interventions during a task's execution to perform an atomic operation and subsequently re-enable them. Disabling and enabling provides the ability to postpone asynchronous events during that period.

- A task needs to be able to completely ignore an asynchronous event.

  This is different from the need to postpone asynchronous events, where the event is eventually delivered. A task may want to receive and ignore an asynchronous exception or intervention.

Asynchronous Exception

Several proposals have been made for delivering exceptions among tasks [10, 23, 24, 25]. In some cases the exception can only be raised in a task blocked doing communication with the task raising the exception, as in Ada. (This simplification makes asynchronous exceptions similar to synchronous exceptions except there are still multiple tasks involved.) In others, the exception can be raised in another task regardless of its execution state. Finally, some schemes convert synchronous and asynchronous interventions into exceptions.

An important question to be asked is whether an asynchronous event should be allowed to cause an exception. We have defined an abnormal event to be the failure of an operation to complete its computation. Since an asynchronous event normally happens during a *successful* computation, we do not believe that asynchronous events should be able to directly cause an exception.

Asynchronous Intervention

Asynchronous interventions exist in almost all systems as interrupt routines. However, this capability is not usually present in the programming languages that run on these systems. Except for Encore EPT [24], whose interventions are restricted to dealing with UNIX signals, we know of no other programming language mechanism that deals with asynchronous interventions. Instead, asynchronous interventions are supplied though the underlying system, for example, UNIX signals. However, UNIX signals are very simple, having no facility to deal with nested handlers and no mechanism to communicate information from the signaller to the signal routine.

The advantages of asynchronous interventions are that the receiver has complete control when an intervention routine is activated. The receiver can install an empty routine to ignore asynchronous intervention activations. The receiver can return to the point of interrupt or raise an exception in the intervention routine. Finally, if the receiving task is blocked awaiting some other task or system operation to complete, an asynchronous intervention will not cause the operation to fail. When a task is blocked, the intervention routine may be called by temporarily awakening it to execute the intervention routine, or the intervention may be postponed until the task is unblocked. (Because of the complexity of implementing the former without compiler support, we have chosen to implement only the latter. Unfortunately, this precludes several useful capabilities, such as pre-empting deadlocked tasks.)

COMPARISON

The following features from the abnormal event models and design requirements can be used to characterize existing abnormal event handling capabilities:

termination – transfer of control to a dynamically determined location.

intervention – invocation of a dynamically determined routine.

multilevel – the use of the multilevel model versus the single-level model.

scoped naming – the use of a scoped name space versus a single name space.

inheritance – the ability to construct hierarchical relationships among exceptions.

parameters – the ability to communicate information from the abnormal event to the handler and possibly back to the activation point for interventions.

	PL/1	CLU	Mesa	Ada	Cui	C++	Eiffel	ML	Modula-3	Argus
termination		●	●	●	●	●	●	●	●	●
intervention	●		●		●		●			
multilevel	N/A		●	●	●	●	●	●	●	
scoped naming	●	●	●	●	●	●	N/A	●	●	●
inheritance						●				
parameters		●	●		†	●		●	●	●
bound					●					
asynchronous										

†Cui's model has data passing only for interventions.

Table 1: Programming Language Comparison

	Encore EPT	Sun LWP	Lee	μSystem
termination	●	●	●	●
intervention	†	●		●
multilevel	●	●	●	●
scoped naming			●	●
inheritance				●
parameters				●
bound				●
asynchronous	†			●

†Encore interventions can only be used with Unix signals.

Table 2: Abnormal Event Library Comparison

bound – exceptions that are fields in data structures, which establishes a relationship between instances of a type and exceptions.

asynchronous – the ability to cause an abnormal event in another task.

Tables 1 and 2 show which characteristics are available in a number of programming languages and abnormal event libraries.

INTRODUCING THE NEW CONSTRUCTS INTO C

Unfortunately, our particular abnormal event model cannot be cast into C without changes to the language; therefore, much of the syntactic and semantic eloquence presented in the previous pseudo code must be abandoned. In short, the C implementation is a low-level interpretation of the model's functionality. Therefore, parts of the previous mechanism are not enforced by the compiler and users are required to follow particular conventions for correct and consistent results. Further, because our implementation uses macro definitions, this leads to some other limitations.

This work extends that done by Lee in Reference [25], Rizk and Halsall in Reference [26], and Roberts in Reference [27]. While our syntax is different, the basic concepts are similar. However, our work provides the following additional features: derived exceptions, exception parameters, bound exceptions, interventions and asynchronous interventions. As a simple introduction to the abnormal event syntax, Figure 2 shows an example that defines two exceptions, one raised in routine f1 and the other in f2 when an abnormal event is detected.

The implementation was built as part of the μSystem light-weight tasking library so that ideas about asynchronous interventions could be tested. The only features of the μSystem that are used in this paper are the non-blocking I/O library, task creation, and communication among tasks using send/receive/reply [28]. The non-blocking I/O routines are the same as their UNIX counterparts except that their names are capitalized and prefixed with the letter "u", and semantically they do not block the UNIX process when performing an I/O operation. Task creation is done by routine uEmit, which has as parameters the name of a C routine that starts running concurrently and a list of arguments that are passed to that routine. Tasks communicate using routine uSend, uReceive and uReply, which have obvious parameters.

All of the following abnormal event facilities work on all the platforms on which the μSystem runs. The μSystem runs on the following processors: M68K, NS32K, VAX, MIPS, i386/486, Sparc, and the following UNIX operating systems: Apollo SR10 BSD, Sun OS 4.x, Tahoe BSD 4.3, Ultrix 3.x/4.x, DYNIX, Umax 4.3, IRIX 3.3.

EXCEPTION DEFINITION

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

Exception Declaration

An exception is declared using type uException. This allocates storage that provides the unique address for identifying the exception, while the storage holds a pointer to the exception from which it was derived. 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, it must be derived from the predefined exception uAny. For example,

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

declares two exceptions and derives them from the predefined exception uAny. A more extensive 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. Our current experience indicates that a single hierarchy is sufficient.

```
uException SingularMatrix = U_EXCEPTION( &uAny );
/* no exception type because it is a simple character string */
uException DimensionMismatch = U_EXCEPTION( &uAny );
struct DimensionMismatchMsg {
   char *msg;
   int r1, c1, r2, c2;
};

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

void MatrixMult( ... ) {
   if ( row/column dimensions are not correct ) {
      DimensionMismatchMsg inst = {
         "non-matching row/column dimensions for matrix multiply", r1, c1, r2, c2 };
      uRaise( NULL, &DimensionMismatch, inst, sizeof(inst) );
   }
   ...
}

void uMain( int argc, char *argv[], char *envp[] ) {
   char *singularmsgp;
   DimensionMismatchMsg *dimmismsgp;
   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, &singularmsgp ) {
            uFprintf( stderr, "%s\n", singularmsgp );
         }
         uWhen( NULL, &DimensionMismatch, &dimmismsgp ) {
            uFprintf( stderr, "matrix M's row size %d must equal N's column size %d\n",
                  dimmismsgp->r1, dimmismsgp->c2 );
         }
      } uEndExcept;
   }
}
```

Figure 2: Matrix Operation Example

Exception Type

When an exception is raised, an exception instance, which contains the fields of the exception, is passed from the raiser to the handler. The data 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, 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, the pseudo code:

```
exception john { int x, y }
exception jonathan : john { float z }
```

is translated into C as:

```
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.

RAISING AN EXCEPTION

Bound exceptions affect the way that exceptions are implemented; free exceptions can be dealt with as a special case of bound exceptions. There are basically two approaches to implementing bound exceptions. Since bound 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 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 bound 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 bound 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 bound exceptions. However, each raise statement and handler now has two pointer values. The assumption is that the number of data-structure instances with bound 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 bound exceptions and provide free exceptions by using the null address for the associated data-structure. Hence, bound 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, as in:

```
exception fred
exception mary_john                 /* bound exception moved out of mary and renamed */

class mary
    ...

mary m

except {
    ...
        raise( null, fred )         /* raise free exception */
    ...
        raise( m, mary_john )       /* raise bound exception */
    ...
  handler( null, fred ) ...         /* catches free exception */
  handler( m, mary_john ) ...       /* catches bound exception */
}
```

When a bound exception has essentially the same function as a free exception, only a single exception needs to be declared. The free and bound 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, for example:

```
FILE *f, *g
exception Eof                       /* free exception for any end of file */

except {
    fgets( ..., f )                 /* read from file f */
    fgets( ..., g )                 /* read from file g */
  handler( f, Eof ) ...             /* catch f's end of file */
  handler( null, Eof ) ...          /* catch any file's end of file */
}
```

Raise Routine

The routine uRaise is used to raise a synchronous exception, as in:

```
uRaise( data-item-address, exception-address,
        exception-type-instance-address, exception-type-instance-length )
```

data-item-address is the address of the data structure that contains the bound 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.

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 block */
    uWhen( ... )
        /* subsequent exception handler block */
    ...
        ...
} 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. (Because an exception already performs termination of control structures, it can be used to get the effect of a break statement.) 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(*data-item-pointer, exception-address, exception-type-instance-pointer*)

data-item-pointer is the address of a data item that contains the bound 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* dereferenced.

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 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() );
    ...
}
...
```

This approach is more general than having a special reraise construct because the predefined routines are useful in situations other than re-raising an exception.

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, which was added to ANSI-C "to ensure that a local variable retains the value it had at the time of the call to longjmp" [29, p. 86]. 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. Requiring users to specify which variables must be volatile is error prone, but this is the best that can be done when abnormal event handling is not part of the programming language.

EXCEPTION VARIABLES

As stated previously, we do not recognize a pressing need for exception variables in the programs we are constructing. Implementing exception variables would have required that users include the exception address as the first field of each exception data structure. For example, instead of:

```
raise( &x, &y, &ei, sizeof(ei) );
```

a user could be required to place the exception address in the exception instance, as in:

```
struct ei′ {
   exceptionvalue v;          /* first field is the address of the exception */
   ...                        /* ei fields */
};
...
ei′.v = &y;                   /* assign exception address for use in raise */
raise( &x, &ei′, sizeof(ei′) ); /* exception argument y is removed */
```

However, we did not see any reason to further burden the user with low-level details to supply a facility that currently has no obvious use.

PREDEFINED EXCEPTIONS

The μSystem provides a number of predefined exceptions; they are derived from one another as shown by the hierarchy in Figure 3. These exceptions are available globally and users can extend the hierarchy with their own exceptions by deriving them from the appropriate predefined exception.

Data Items Associated with Predefined Exceptions

The exceptions uActiveTasksEx, uEmitEx, and uCreateProcessorEx are raised with the address of the μSystem cluster on which the operation is being executed (see Reference [1] for an explanation of a cluster). All other μKernel exceptions are raised with no associated data item.

All predefined I/O exceptions, i.e. uIOEx and those below it, 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 is 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.

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.

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.

μSystem Kernel Exception Types

Some examples of the exception types for the μSystem kernel exceptions are as follows:

```
typedef uSystemExMsg uKernelExMsg, uActiveTasksExMsg, uCreationExMsg;

typedef struct {
    uCreationExMsg base;
    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;
```

uAny	*Global exception*
uSystemEx	*Errors in the uSystem*
uKernelEx	*Errors in the uKernel*
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 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 is not*
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 3: μSystem Predefined Exception Hierarchy

Figure 4 is a contrived example that attempts 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( ..., stack_size, ... );
            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 */
                }
            }
        } uEndExcept;
    }
} uHandlers {
    uWhen( NULL, &leave, NULL );
}
```

Figure 4: Repeated Attempts to Create a Task

I/O Exception Types

Some examples of the exception types for the I/O exceptions are as follows:

```
typedef struct {
    uSystemExMsg base;
    int errno;              /* UNIX errno */
} uIOErrorExMsg, uSocketErrorExMsg, uFileStreamExMsg;

typedef struct {
    uFileStreamExMsg base;
    char *path;             /* path of file to be opened */
    char *perms;            /* permissions for opening of formatted file (uStream) */
    int  flags;             /* access type for opening of unformatted file (uFile) */
    int  mode;              /* protection mode for unformatted file only (uFile) */
} uNoPermsExMsg;
```

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( NULL, &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 );
      }
      f = uFopen( "/dev/null", ... );       /* corrective action */
   }
} uEndExcept;
```

INTERVENTION DEFINITION

As with exceptions, C's limitations preclude direct implementation of our intervention model. As a result, an intervention definition must be composed of two parts, the intervention name and the type of a routine pointer that can be pushed on the intervention's stack. As well, the intervention handlers must be separate routines instead of part of an intervention statement because there is no nesting of routine definitions in C; therefore, it is difficult to have code fragments, i.e. handlers, in a control structure that are subsequently called. Further, bound interventions must be dealt with in the same way as for exceptions, that is, made into free interventions. Finally, to achieve static type checking when calling an intervention and when installing an intervention routine, the name of the intervention must be used; therefore, pointers to interventions cannot be used in either context.

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 );  /* declare an intervention */
```

Because of C declaration syntax and the implementation, the *pointer-to-routine-type* must be a typedef name. 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 );   /* external intervention declaration */
```

One of the compilation units in a program must make the actual intervention declaration.

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 ); /* declare an asynchronous intervention */
uAsyncInterExtern( intervention-name );  /* external intervention asynchronous declaration */
```

The type of a routine pointer that can be pushed on an 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.

CALLING AN INTERVENTION

Synchronous Intervention

A synchronous intervention is called as follows:

```
uInterCall( intervention-name )( 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.

Asynchronous Intervention

An asynchronous intervention is activated as follows:

```
uAsyncInterCall( task-id, intervention-name, message-ptr, message-length );
```

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.

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

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 */
```

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 is invoked. Asynchronous interventions are established in the same way, only the routines that can be stacked for asynchronous interventions must have the predefined type for message passing.

ADVANTAGES OF ASYNCHRONOUS INTERVENTION

To show how asynchronous interventions can be used to advantage, a parallel database search was constructed using three different approaches: no abnormal termination, abnormal termination using polling, and abnormal termination using asynchronous interventions. By comparing the results of the first search with the last two, it is possible to see the performance benefit of abnormal termination. By comparing the coding styles of the last two, it is possible to see the benefits of asynchronous interventions.

The parallel search was structured to simulate searching a partitioned text-database for a particular key, for example, to find all documents/papers with a particular keyword. Each document is partitioned across multiple disk-drives so that it can be searched in parallel. The search-controller task selects each document, and for each, starts N searcher tasks, where N is the number of partitions. If any one of the N search tasks finds the keyword, it reports to the search-controller, which includes that document in the list containing the keyword. Once one of the N searcher tasks reports success, the remaining searchers can be stopped. A new search cannot begin until all searchers complete their current search. The search-controller keeps a pool of searcher-tasks available, eliminating the cost of creating the searcher-tasks for the next search. To simplify the experiment, the search-controller searches one document with 100 different keys that all exist in the document, instead of searching 100 different documents. The document is approximately 1.4M and is partitioned into 1 to 8 equal pieces. We had only 4 disk drives on which to partition the document, so access contention begins to reduce the parallelism for more than 4 partitions.

Using synchronous communication methods to solve this problem requires polling using messages or task killing, which is an asynchronous operation; however, killing tasks precludes both programmatic shutdown of the search-tasks and reusing them for subsequent searches. Therefore, asynchronous communications methods are important.

The details of the three different kinds of searches are:

- In the no-abnormal-termination search, each searcher-task completes searching its partition regardless of the results of the other searcher-tasks.
- In the abnormal-termination search using polling, a global flag is used to indicate the status of the search. If a searcher-task finds the key, it sets the flag. A searcher-task can check the flag at any time to determine if it should abort its search. To get a searcher-task to stop as quickly as possible after the flag is set, the flag is checked on every iteration through the search loop.
- In the abnormal-termination search using asynchronous intervention, the controller-task performs an asynchronous intervention activation to each of the searcher tasks after a searcher-task reports success.

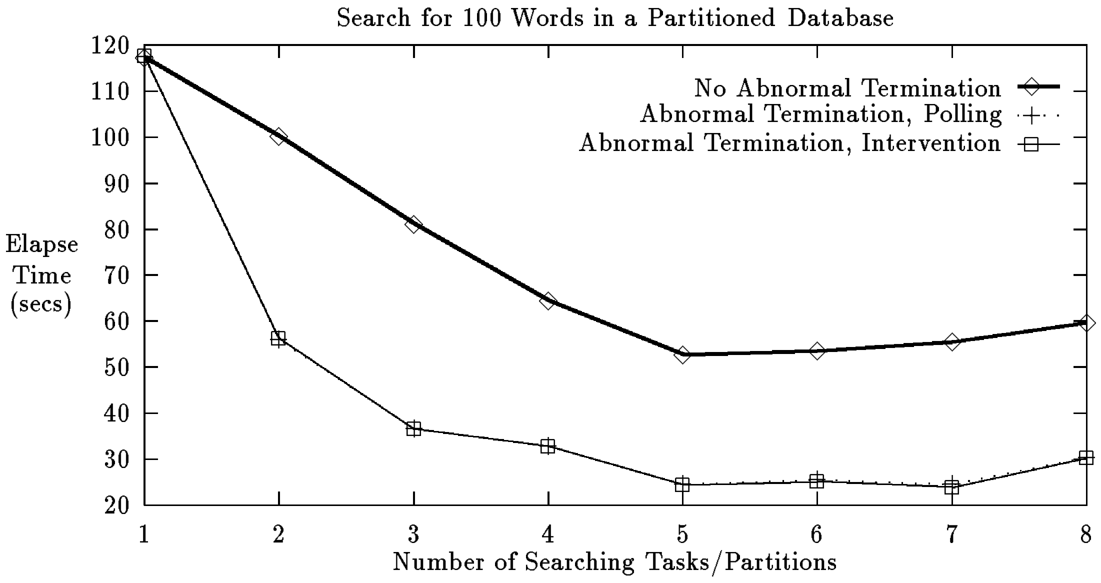

Figure 5: Partitioned-Database Search Results

> The intervention routine raises an exception in the searcher-task so that polling is unnecessary. The searcher-task installs an asynchronous intervention routine for the duration of the search, which raises the abnormal termination exception, and the search code is also inside an exception block.

The abnormal-termination search routines are given in Appendix B.

Interpreting the Results

Figure 5 shows the results of the elapsed times for the three different partitioned-database searches. The searches are equal for one searcher-task because they all stop searching at the same point. In all other cases, the execution time deceases as parallelism is introduced until 5 partitions when bus/controller/disk contention begins to slow the search. When there is parallelism, the no-abnormal-termination search takes about twice as long as the abnormal-termination searches because of the unnecessary searching. The abnormal-termination searches take the same time.

The difference in coding styles between the two abnormal-termination searches is fairly obvious. Using global references is, in general, an undesirable programming technique and polling requires inserting checks at appropriate places to detect the event, which complicates coding and maintenance of the program. A programmer has to place the check(s) so that it is performed often enough but not too often. Finally, using global references is not extensible to a distributed environment without distributed shared-memory, whereas an asynchronous intervention activation may be.

IMPLEMENTATION ISSUES

Interventions and exceptions are related and affect one another. Both create logical stacks of handlers or routines that are added to and removed from as uExcept or uInter blocks are entered and exited. These blocks can be exited in the normal manner or because an exception is raised during the execution of the block. Regardless of how the blocks are exited, the appropriate stack item must be removed at each level during the search for a handler or intervention routine.

In the μSystem, exception handlers and intervention routines are associated with the current coroutine or task in which they are installed. Exceptions and synchronous interventions are raised or called within a coroutine or task, and the appropriate exception handler or intervention routine is invoked, according to the handlers and routines installed. Asynchronous interventions are activated from another task, but the intervention routine activated is the one associated with the called task or the coroutine it is currently executing. Therefore each task and each coroutine has an additional data structure associated with it. This data structure is a list of installed exception handlers and interventions, called the EI list. Both exception handler and intervention nodes appear on the same EI list to make it possible to remove the correct nodes when abnormal flow control (an exception) takes place. Exception handler nodes contain the information necessary to restore context when an exception is raised and some additional housekeeping information used when reraising an exception. The current context could be saved using the UNIX library routines setjmp and longjmp; however, since the μSystem already supplies optimized machine specific context switching code for coroutine and task switching, this was used instead. Intervention nodes contain the information necessary to map an intervention name to the routine to be invoked.

The EI list is searched when an exception is raised or a intervention call made. The fact that there is only one list containing both kinds of nodes means that the cost of a search depends not only on the number of nodes but also the kinds of nodes, as exception nodes take longer to terminate. In general, we do not consider this cost to be significant because exceptions and interventions will be raised rarely, as they indicate an abnormal event, and the number of nodes on the EI list will be small.

The common EI list also makes it possible to clean up allocated data for asynchronous interventions. For the target task of an asynchronous intervention activation, a data structure is allocated containing the message for the intervention and the name of the intervention. This data structure is atomically added to a linked list on the target task descriptor, called the AI list. When the μSystem determines that it is appropriate to call the intervention routine in the target task, it atomically removes the data structure for the intervention from the AI list, pushes it onto the top of the EI list, and then locates and calls the appropriate intervention routine. If an exception is raised during execution of the asynchronous intervention routine, the search for a handler will encounter the AI item on the EI list. This means that control will not return from the intervention routine. At this point, all data associated with the AI item are freed. Note that this cleanup only occurs if the search for a handler encounters the AI item. If an exception handler is found before the search reaches the AI item, it means that control will return from the intervention routine back to the point of interruption, unless another exception is raised. Finally, when a task terminates, any pending intervention calls on the AI list are simply freed.

PERFORMANCE

To give a general idea as to the performance of the Sequent Symmetry, on which the following timings were performed, the following benchmarks are given: a procedure call with no parameters and returning no result takes 3 microseconds, a setjmp followed by a call to a routine that performs a longjmp takes 17 microseconds, a signal (SIGUSR1) with a null handler body takes 434 microseconds.

Exceptions

Due to the lack of compiler support, we could not implement techniques that have little or no execution overhead for executing a uExcept statement [30, p. 455]. This only becomes important for low-level routines which are invoked often and must guarantee execution of termination code if a lower-level exception is raised. For example, I/O routines must ensure clean up code is executed if they are terminated because of a lower-level exception, particularly an exception raised in an asynchronous intervention routine.

The cost of raising a synchronous exception is not fixed and depends on the length of the linear search to locate the handler. There is a fixed cost to start the search, a cost for searching the EI list, a cost for each nested uExcept block (i.e. a longjmp equivalent) on the EI list to get to the exception handlers, a cost for each uWhen during the handler search, and a cost to make the exception instance available for the exception hander if one is selected. Table 3 shows the time to execute a uExcept statement with an empty exception body and to raise an exception where the search traverses 3 exception blocks and each block has 3 handlers; the handler that is found is the third handler in the third exception block and no data is passed from raiser to handler.

Synchronous Interventions

The cost of raising a synchronous interventions is not fixed and depends on the length of the linear search to locate the intervention routine. The variable cost is the search through the EI list to locate the appropriate intervention and a fixed cost for the intervention routine call. Table 3 shows the time to execute a uInter statement with an empty body and to call an intervention where the search traverses 3 intervention blocks and calls the intervention routine associated with the third intervention block; no data is passed from caller to handler and no value is returned.

Asynchronous Interventions

The cost of raising an asynchronous intervention is not fixed and depends on the state of the task that is to be interrupted. Unfortunately, there is a particular scenario in which the intervention will never be delivered. The cost of asynchronous intervention is divided between the activator and the target task. The activator allocates storage for the message as it must be copied, waits until it can safely attach the message to the other task, and links the message to the task; the activator then continues execution. The target task only receives the intervention when it next stacks 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). Fortunately, this situation is rare. Once the decision is made to perform the intervention, the cost is the same as for a synchronous intervention call. After the call, there is the cost of unlinking the message and deleting it. Table 3 shows the time to activate an intervention routine in another task where the search traverses 3 intervention blocks and calls the intervention routine associated with the third intervention block; no data is passed from activator to handler and no value is returned.

computer/CPU	uExcept installation	uRaise 3 levels, 3 handlers	uInter installation	uInterCall 3 levels	uAsyncInterCall 3 levels
Sequent Symmetry S27 Intel 386, 16Mhz	17 μsecs	203 μsecs	15 μsecs	19 μsecs	170 μsecs

Table 3: Executions Timings

CONCLUSIONS

We believe that the exception and intervention models and the selected design requirements provide an excellent set of tools for handling abnormal events in a program. The basic design could be incorporated into any block structured programming language and all constructs proposed can be statically type checked. The differentiation between exceptions and interventions clarifies two quite different mechanisms for handling abnormal events. Each provides a facility that cannot be mimicked by the other or using existing programming language facilities without violating abstraction or losing execution performance. The novel ideas introduced by this work are bound exceptions and combining the general notion of interventions with asynchronous usage.

While we are far from happy with the syntax and lack of programming language support for the C implementation, we are pleased with the functionality that has been provided. When used with a small set of conventions, it is possible to write C programs that have largely all the functionality described in the model and design requirements. Incorporating exceptions into the μSystem and all μSystem I/O cover routines was a significant amount of work, in particular, organizing the hierarchical relationships among the exceptions and the exception data passed from exception to handler. The benefits accrued were a substantial improvement in the readability, reliability and maintainability of programs because it is no longer possible to ignore return codes that indicate abnormal events nor is a program cluttered with numerous checks of return codes.

APPENDIX A Merge Files 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 files */
  uFclose( out );                                      /* close output file */
} /* uMain */
```

APPENDIX B Parallel Searches Using Abnormal Termination

B.1 Abnormal Termination Using Polling

```
int stop;                                              /* global flag for polling for termination */

#define BufferSize ( 8 * 1024 )

void Searcher( uFile searchfile, uTask controller ) {  /* search a file for a key */
  uEofExMsg *eof;
  char *key;
  char text[BufferSize];
  volatile int nbytes, locn;
  int found;

  for ( ;; ) {                                         /* lookup each key */
     uReply( uReceive( &key, sizeof(key) ), NULL, 0 ); /* get key from controller */
   if ( key == NULL ) break;
     uExcept {
        for ( ;; ) {
           nbytes = uRead( searchfile, text, sizeof(text) );     /* read 8K block of text */
           locn = bmhsearch( text, nbytes, key, strlen( key ) ); /* fast search of key in text */
         if ( locn <= nbytes ) {                                 /* key found ? */
              found = 1;
              stop = 1;
              break;
           }; /* exit */
         if ( stop ) {                                           /* someone else found key ? */
              found = 0;
```

```
                break;
            }; /* exit */
        } /* for */
    } uHandlers {
        uWhen( searchfile, &uEofEx, &eof ) {              /* end of file while reading text file */
            found = 0;
        } /* uWhen */
    } uEndExcept;
    uSend( controller, NULL, 0, &found, sizeof(found) );   /* send controller the result of the search */
  } /* for */
} /* Searcher */
```

B.2 Abnormal Termination Using Asynchronous Interventions

```
uAsyncIntervention( SearchInterrupt );
uException SearchStopped = U_EXCEPTION( &uAny );

void DummyInter( void *msg, int msglen ) {                /* invoked if asynchronous intervention is */
} /* DummyInter */                                        /* delivered after the search is complete */

void TerminateSearch( void *msg, int msglen ) {           /* invoked if asynchronous intervention is */
  uRaise( NULL, &SearchStopped, NULL, 0 );                /* delivered during the search */
} /* TerminateSearch */

#define BufferSize ( 8 * 1024 )

void Searcher( uFile searchfile, uTask controller ) {     /* search a file for a key */
  uEofExMsg *eof;
  char *key;
  char text[BufferSize];
  volatile int nbytes, locn;
  int found;

  uInter( SearchInterrupt, DummyInter ) {
    for ( ;; ) {                                          /* lookup each key */
      uExcept {
        uReply( uReceive( &key, sizeof(key) ), NULL, 0 ); /* get key from controller */
       if ( key == NULL ) break;
        uExcept {
          uInter( SearchInterrupt, TerminateSearch ) {
            for ( ;; ) {
              nbytes = uRead( searchfile, text, sizeof(text) ); /* read 8K block of text */
              locn = bmhsearch( text, nbytes, key, strlen( key ) ); /* fast search of key in text */
             if ( locn <= nbytes ) {                      /* key found */
                found = 1;
                break;
              }; /* exit */
            } /* for */
          } uEndInter;
        } uHandlers {
          uWhen( searchfile, &uEofEx, &eof ) {            /* end of file while reading text file */
            found = 0;
          } /* uWhen */
        } uEndExcept;
      } uHandlers {
        uWhen( NULL, &SearchStopped, NULL ) {             /* someone else found key */
          found = 0;
        } /* uWhen */
```

```
            } uEndExcept;
            uSend( controller, NULL, 0, &found, sizeof(found) );     /* send controller the result of the search */
        } /* for */
    } uEndInter;
} /* Searcher */
```

REFERENCES

1. P. A. Buhr and R. A. Stroobosscher, 'The μSystem: Providing light-weight concurrency on shared-memory multiprocessor computers running unix', *Software–Practice and Experience*, 20, (9), 929–963, Sept. 1990.

2. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice Hall Software Series. Prentice Hall, second edn, 1988.

3. B. Meyer, *Object-oriented Software Construction*, Prentice Hall International Series in Computer Science. Prentice-Hall, 1988.

4. P. A. Lee and T. Anderson, *Fault Tolerance - Principles and Practice*, volume 3 of *Dependable Computing and Fault-Tolerant Systems*, Springer-Verlag, 2nd revised edn, 1990.

5. B. Randell, 'System structure for software fault tolerance', *IEEE Trans. Softw. Eng.*, SE-1, (2), 220–232, June 1975.

6. J. B. Goodenough, 'Exception handling: Issues and a proposed notation', *Commun. ACM*, 18, (2), 683–696, Feb. 1975.

7. P. A. Buhr, 'A case for teaching multi-exit loops to beginning programmers', *SIGPLAN Notices*, 20, (11), 14–22, Nov. 1985.

8. A. Koenig and B. Stroustrup, 'Exception handling in C++', *Journal of Object-Oriented Programming*, 3, (2), 16–33, July/August 1990.

9. T. N. Turba, 'The Pascal exception handling proposal', *SIGPLAN Notices*, 20, (8), 93–98, Aug. 1985.

10. United States Department of Defense, *The Programming Language Ada: Reference Manual*, ANSI/MIL-STD-1815A-1983 edn, Feb. 1983, Published by Springer-Verlag.

11. R. Milner, M. Tofte, and R. Harper, *The Definition of Standard ML*, MIT Press, Cambridge, Massachusetts, U. S. A., 1990.

12. M. A. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*, Addison Wesley, first edn, 1990.

13. B. Liskov, R. Atkinson, T. Bloom, E. Moss, J. C. Schaffert, R. Scheifler, and A. Snyder, *CLU Reference Manual*, volume 114 of *Lecture Notes in Computer Science*, Springer-Verlag, 1981.

14. J. G. Mitchell, W. Maybury, and R. Sweet, 'Mesa language manual', Technical Report CSL–79–3, Xerox Palo Alto Research Center, Apr. 1979.

15. B. H. Liskov and A. Snyder, 'Exception handling in CLU', *IEEE Trans. Softw. Eng.*, SE-5, (6), 546–558, Nov. 1979.

16. J. Rees and W. Clinger, 'Revised^3 report on the algorithmic language Scheme', *SIGPLAN Notices*, 21, (12), 37–79, Dec. 1986.

17. G. V. Cormack and A. K. Wright, 'Type-dependent parameter inference', *SIGPLAN Notices*, 25, (6), 127–136, June 1990, Proceedings of the ACM Sigplan'90 Conference on Programming Language Design and Implementation June 20-22, 1990, White Plains, New York, U.S.A.

18. M. D. MacLaren, 'Exception handling in PL/I', *SIGPLAN Notices*, 12, (3), 101–104, Mar. 1977, Proceedings of an ACM Conference on Language Design for Reliable Software, March 28–30, 1977, Raleigh, North Carolina, U.S.A.

19. H. Bretthauer, T. Christaller, and J. Kopp, 'Multiple vs. single inheritance in object-oriented programming languages. what do we really want?', Technical Report Arbeitspapiere der GMD 415, Gesellschaft Für Mathematik und Datenverarbeitung mbH, Schlo*β* Birlinghoven, Postfach 12 40, D-5205 Sankt Augustin 1, Deutschland, Nov. 1989.

20. T. A. Cargill, 'Does C++ really need multiple inheritance?', In *USENIX C++ Conference Proceedings*, pages 315–323, San Francisco, California, U.S.A, Apr. 1990. USENIX Association.

21. I. Choi and M. V. Mannino, 'Graph interpretation of methods: A unifying framework for polymorphism in object-oriented programming', *OOPS Messenger*, 2, (1), 38–54, Jan. 1991.

22. Q. Cui, 'Data-oriented exception handling', Technical Report CS-TR-2384, Department of Computer Science, University of Maryland, College Park, Maryland, U. S. A., 20742, Jan. 1990.

23. A. Szalas and D. Szczepanska, 'Exception handling in parallel computations', *SIGPLAN Notices*, 20, (10), 95–104, Oct. 1985.

24. Encore Computer Corporation, *Encore Parallel Thread Manual, 724-06210*, May 1988.

25. P. A. Lee, 'Exception handling in C programs', *Software–Practice and Experience*, 13, (5), 389–405, May 1983.

26. A. Rizk and F. Halsall, 'Design and implementation of a C-based language for distributed real-time systems', *SIGPLAN Notices*, 22, (6), 83–100, June 1987.

27. E. S. Roberts, 'Implementing exceptions in c', Technical Report 40, Digital Systems Research Center, 130 Lytton Avenue, Palo Alto, California, 94301, Mar. 1989.

28. W. M. Gentleman, 'Message passing between sequential processes: the reply primitive and the administrator concept', *Software–Practice and Experience*, 11, (5), 435–466, May 1981.

29. American National Standards Institute, 1430 Broadway, New York, New York 10018, *American National Standard for Information Systems – Programming Language – C*, Dec. 1989, X3.159-1989.

30. C. N. Fischer and R. J. LeBlanc, Jr., *Crafting a Compiler*, Benjamin Cummings, 1988.