The Mach Exception Handling Facility

David L. Black, David B. Golub, Karl Hauth
Avadis Tevanian[1] , and Richard Sanzi

Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

1. Introduction

Exceptions are caused by the occurrence of unusual conditions during program execution; raising an exception invokes the operating system to manage recovery from the unusual condition. Although many exceptions can be handled and dismissed transparently to the user (e.g. page faults), some must be exported in some form (invoke handler, core dump, etc.). This paper concerns the design and implementation of exception handling facilities that perform this exporting for general purpose operating systems. We discuss both the overall design principles for these facilities and the specific design and implementation of an exception handling facility for the Mach [1] operating system. We also discuss the support provided to applications that use exceptions (e.g. debuggers) by this facility and other Mach kernel facilities.

Mach supports debuggers via a combination of independent kernel facilities instead of concentrating debugger support in a single kernel component (e.g. ptrace). This approach avoids duplicating functionality within the kernel by designing debugger support facilities to allow use by other applications. This results in increased flexibility and functionality which benefits all applications using these facilities, including debuggers. In addition the kernel is simplified by the corresponding reduction in special purpose debugger support code.

The independence and network transparency properties of Mach's communication facility (Mach ipc) also provide additional debugging functionality. Mach ipc is used to implement not only the exception handling facility, but also the interfaces to all Mach kernel facilities. The independence and network transparency properties separate applications from the servers (including the kernel) that implement facilities. This separation supports parallel and distributed debugging by removing restrictions on relationships between debuggers and other applications, and by allowing components of both debuggers and applications to be distributed over a network without change.

The remainder of this paper has the following structure. We begin with an introduction to the Mach system and a review of exceptions and their uses including previous work in this area. We then describe a general model for exception handling and use it to design the Mach exception handling facility. Additional sections discuss the implementation of applications that use exceptions with particular attention to debuggers. We conclude by describing a debugger that has been implemented using this facility.

2. The Mach System

Mach is a multiprocessor operating system under development at Carnegie-Mellon University. It is based on a simple communication-oriented kernel that exports a small number of abstractions to users via an integrated interface; many traditional operating system functions are implemented by user programs or servers outside the kernel. The kernel itself exports exactly five abstractions to users, the *task*, *thread*, *port*, *message*, and *memory object*. Of these the *memory object* is not relevant to the subject matter of this paper.

Mach splits the traditional notion of a process into the *task* and *thread* abstractions. A *task* is the basic

---

[1]Author's present address is NeXT, Inc., Palo Alto, CA.

unit of resource allocation. It includes a paged virtual address space and protected access to system resources such as communication capabilities and virtual memory. The *thread* is the basic unit of computation. It is a lightweight process executing withing a task, and consists solely of a processing state (i.e. program counter and registers). All threads within a task share the address space and communication rights of that task. A traditional process would consist of a task with a single thread in the Mach system.

Mach's communication facility is implemented by the *port* and *message* abstractions. A *port* is a communication channel – logically a queue for messages protected by the kernel. Ports are the reference objects in Mach and are used in a similar fashion to object references in an object oriented system. Send and Receive primitive operations implement communication by sending and receiving messages to and from ports. A *message* is a typed collection of data objects used in communication; messages may be of any size and can contain pointers and typed capabilities for ports. This communication facility is a basic building block on which more complicated objects may be constructed; it is the underlying communication mechanism for the Mach exception handling facility.

3. Exceptions and their Uses

Exceptions are synchronous interruptions to the normal flow of program control caused by the program itself. This excludes software interrupts and other actions caused by asynchronous external events. Exceptions include illegal accesses (bus errors, segmentation and protection violations), arithmetic errors (overflow, underflow, divide by zero), and hardware instructions intended to support emulation, debugging, error detection, etc. Hardware exceptions cause traps into the operating system; the system handles certain exceptions transparently (e.g. recoverable page faults), but the remaining exceptions are exported to the user by the operating system's exception handling facility.

There are four major classes of applications that use exceptions:

- Debugging. Debuggers rely on exceptions generated by hardware trace and breakpoint facilities. Other exceptions that indicate errors must be reported to the debugger; the presence of the debugger indicates the user's interest in any anomalous program behavior.
- Core dumps. In the absence of a debugger, a fatal exception can cause the execution state of a program to be saved (in a file) for later examination. Unix[2] systems refer to these files as 'core dumps' for historical reasons. [3]
- Error handling. Certain applications handle their own exceptions (particularly arithmetic) under some circumstances. For example, a handler could substitute zero for the result of a floating underflow and continue execution. Error handlers are often required by high-level languages (e.g. Ada[3] [2]).
- Emulation. Virtually all modern machines generate exceptions upon encountering operation codes that cannot be executed by the hardware. Emulators can be built to execute the desired operation in software. Such emulators serve to extend the instruction set of the underlying machine by performing instructions that are not present in the hardware (e.g. the Microvax II[4] relies on software "macrocode" to implement infrequently used instructions [5])

Many computers regard system calls as another form of exception at the hardware level; system calls cause the hardware to suspend execution and trap into the operating system (e.g. VAX[5] [4]). The operating system distinguishes system calls from other hardware exceptions; system calls are requests for

[2]Unix is a trademark of AT&T Bell Laboratories

[3]Ada is a registered trademark of the U.S. Government, Ada Joint Program Office

[4]Microvax and Microvax II are trademarks of Digital Equipment Corporation

[5]VAX is a trademark of Digital Equipment Corporation

services from the operating system, and are better viewed as extracodes or meta-instructions implemented by the operating system [8]. This places system calls beyond the scope of any exception handling facility.

4. Existing Approaches

Operating systems designers have approached the area of exceptions in a variety of ways. The Accent operating system [7], a predecessor of Mach, used a message-based facility for debugger invocation upon exception occurrence. This debugger was often the kernel debugger which had limited utility for debugging user programs. In contrast, the Medusa operating system for Cm* [6] provided almost half a dozen different options for handling exceptions. Medusa introduced the idea of stopping a thread/task/process when an exception occurred and invoking a separate "buddy" entity to deal with the exception. Most other research systems have designed exception handling facilities on an ad-hoc basis to provide the minimum functionality necessary for the debugger(s) and/or error handler(s) on that system. Often this involved a simple mechanism to reflect exceptions back to a user program (in a limited fashion) or cause a core dump. The major drawbacks of these approaches are limited functionality (often caused by design solely for use by debuggers), and lack of extensibility to a multi-threaded environment.

Unix generalizes exception handling to the signal facility which handles all interruptions to normal program flow. Differing requirements of the various interruptions (exceptions, timer expiration, control character on terminal, etc.) entail semantics that vary from signal to signal; the default action can be nothing, stop, continue from stop, or terminate (with or without a core dump). Users can change these defaults or specify a handler to be invoked by a signal. The interface to these handlers includes a partial machine context; registers outside this context are not accessible. Debugging support is centralized in the ptrace() system call; it performs all data transfer and process control needed by debuggers, and interacts with the signal facility to make signals visible to debuggers (including signals that would otherwise invoke error handlers or emulators). The occurrence of a signal in a debugged process causes that process to stop in a peculiar manner and notify the debugger that something has happened. This notification is implemented by special treatment of debugged processes in the wait() system call; this call usually detects terminated (zombie) processes, but also detects stopped processes that are being debugged. One consequence of these features and their implementation is that debuggers are restricted to debugging processes that are the immediate children of the debugger.

There are a number of major problems with the Unix signal facility. Two of the most important ones are:

- Executing the signal handler in the same context as the exception makes many registers inaccessible. These registers are often the very registers that an arithmetic error handler needs to modify. (e.g. by substituting 0 for a floating underflow).
- The entire concept of signals is predicated on single-threaded applications. Adapting signals to multi-threaded applications is difficult and complicates the interface to them. At least half a dozen major changes to the Unix signal implementation in the Mach kernel have been required for this reason.

The typical use of signal handlers is to detect and respond to external events; for this they are adequate, but as an exception handling facility they leave much to be desired. The original Unix signal mechanism wasn’t even adequate for detecting and responding to external events; signals were viewed as almost always fatal [3].

5. A Model for Exception Handling

The Mach exception handling facility is based on a general model that describes the use of exceptions. This model is derived from the requirements of applications that use exceptions. The generality of the resulting model is sufficient to describe virtually all uses of exceptions, including those made by the applications discussed in Section 3.

Applications that use exceptions can be divided into two major classes:

1. Error Handlers: These components perform recovery actions in response to an exception and resume execution of the thread involved. This class includes both error or exception handlers and emulators. Error Handlers typically execute in the same address space as that thread for efficiency reasons (access to state).

2. Debuggers: These components examine the state of an entire application to investigate why an exception occurred and/or why the program is misbehaving. This class includes interactive debuggers and servers that produce core dumps; the latter can be viewed as front ends to debuggers that examine core dumps. Debuggers usually execute in address spaces distinct from the application for protection reasons.

This paper uses the terms ‘error handler’ and ‘debugger’ to refer to the entire classes delineated above (e.g. a core dumper is a ‘debugger’). In addition the term ‘handler’ is used to refer to any application that uses exceptions.

Our model is derived by examining the requirements common to error handlers and debuggers. The occurrence of an exception requires suspension of the thread involved and notification of a handler. The handler receives the notification and performs some computation (e.g. error handler fixes the error, debugger decides what to do next), after which the thread is either resumed or terminated. On this basis we propose the following model to cover all uses of exceptions: the occurrence of an exception invokes a four step process involving the thread that caused the exception (Victim) and the entity that handles the exception (Handler, may be the operating system):

1. Victim: **raise** -- cause notification of an exception’s occurrence.

2. Victim: **wait** -- synchronize with completion of exception handling.

3. Handler: **catch** -- receive notification. This notification usually identifies the exception and the victim. Some of this identification may be implicit in where and how the notification is received.

4. Handler: take action. There are two possible actions:
    - **clear** -- clear exception causing victim to return from **wait**.
    - **terminate** -- cause termination of victim thread.

The boldface primitives in this model constitute the high-level model interface to exceptions and can be viewed as operating on "exception objects". The handler will usually perform other functions between the **catch** and **clear** or **terminate** steps; these functions are particular to the handler application itself and are not part of the exception model.

Any exception handling facility must implement these primitives in some form; as an example we consider signal handling in Unix. An exception that invokes a signal handler uses the following implementation of the model:

1. **raise** - Internal kernel code that translates the hardware exception to a signal and sends the signal to the process. The kill() system call can be used to mimic this for user-detected exceptions.

2. **wait** - Implicit because the handler and victim execute in the same process; execution of the victim cannot resume until the handler completes.

3. **catch** - Internal kernel code that invokes the handler and sets up its stack. Both the exception and the entity that caused it are implicit in the choice of the handler and the process context that it executes in.

4. **clear** - sigcleanup() or sigreturn() system call invoked when handler exits. These calls unwind the stack and clear the kernel state associated with a signal handler.

5. **terminate** - exit() system call.

The model also applies to Unix support for debuggers; this is of less interest because the support consists primarily of special-purpose code in the Unix kernel. Much of this code duplicates functionality found elsewhere in the kernel.

This exception handling model also serves as a guide for future exception handling facilities. Many of the problems exhibited by other exception handling facilities are due to their implementation of a restricted version of our model (e.g. Unix signal handler must execute in victim's context). These problems can be avoided by a design that implements the model in full generality; such a design can be more flexible and provide greater functionality without impacting the size of the kernel. The next section provides an example of such a design, the Mach exception handling facility.

6. Design of the Mach Exception Handling Facility

The Mach exception handling facility was designed from scratch as a general implementation of our exception handling model. The major design goals for this new facility were:

- A single facility with consistent semantics for all exceptions.
- Clean and simple interface.
- Full support for debuggers and error handlers.
- No duplication of functionality within kernel.
- Support user-defined exceptions.

A consequence of these goals is that we reject the notion of a handler executing in the same context as the exception it is handling. We know of no clean and straightforward way to make a thread's context available to the thread itself; this results in a single thread having multiple contexts (a currently executing context and one or more saved exception contexts). In turn this causes serious naming and functionality problems for operations that access or manipulate thread contexts. Because Mach supports multiple threads within the same task, it is sufficient to stop the thread that caused the exception and execute the handler as another thread in the same task.

The Mach exception handling facility implements the exception handling model via Mach kernel primitives to avoid duplication of kernel functionality. Because the handler never executes in the context of the victim thread, the **raise**, **wait**, **notify**, and **clear** primitives constitute a remote procedure call (rpc). We therefore implement them using a message-based rpc provided by Mach's communication facility. The remaining **terminate** primitive is exactly Mach's **thread_terminate** or **task_terminate** primitive; no special action is required to terminate the thread or task instead of completing the rpc.

The exception rpc consists of two messages; an initial message to invoke the rpc, and a reply message to complete the rpc. The initial message contains the following items:

- Send and reply ports for the rpc.
- The identities of thread that caused the exception and the corresponding task.
- A machine independent exception class. (cf. appendix A)
- Two machine-dependent fields that further identify the exception.

If the rpc is completed, the reply message contains the two rpc ports and a return code from the handler that handled the exception (success in almost all cases). MiG (Mach interface Generator) generated stub routines perform the generation and decoding of the messages; this allows users to avoid dealing directly with the contents of the messages.

An exception rpc corresponds to our exception model as follows:

- **raise -** send initial message.
- **wait -** wait for and receive reply message.
- **catch** - receive initial message.
- **clear** - send reply message.

6.1. Exception Ports

The two messages that constitute the rpc are sent to and received from ports corresponding to the handler (initial message) and victim (reply message). The handler's port is registered as the exception port for either the victim's task or thread; the kernel consults this registration when an exception occurs. The reply port is specified in the initial message; for hardware exceptions the kernel allocates the reply port and caches it for reuse on a per-thread basis. Mach kernel primitives are available to register a port as an exception port for a task or thread, and to return the port currently registered; these and other useful kernel primitives for implementing debuggers and error handlers are described in Appendix B.

Registering exception ports for both tasks and threads effects a separation of concerns between error handlers and debuggers. Error handlers are supported by the thread exception ports because error handlers usually affect only the victim thread; different threads within a task can have different error handlers. The registered exception port for a thread defaults to the null port at thread creation; this defaults the initial error handler to "no handler". Debuggers are supported by the task exception ports because debuggers operate on the application level; this includes at least all the threads in the victim's task, so at most one debugger is ever associated with a single task. The registered exception port for a task is inherited from the parent task at task creation; this supports debuggers that handle trees of tasks (e.g. a multi-tasking parallel program) and inheritance of core-dump servers.

The presence of both task and thread exception ports creates a potential conflict because both are applicable to any exception. This is resolved by examining the differences between error handlers and debuggers. Error handlers use exceptions to implement portions of an application; an error handler is an integral part of the application that generates its exceptions. Exceptions handled by an error handler may be unusual, but they do not indicate anomalous or erroneous behavior. In contrast debuggers use exceptions to investigate anomalous or erroneous application behavior; as a result debuggers have little interest in exceptions successfully handled by error handlers. This implies that exceptions should invoke error handlers in preference to debuggers; this preference is implemented by having thread exception ports take precedence over task exception ports in determining where to direct the rpc invoked by an exception. If neither an error handler nor a debugger can successfully handle an exception, the task is terminated.

6.2. User Extensibility

Mach's exception handling facility permits users to define and handle their own exceptions in addition to those defined by the system. The software class of exceptions (cf. appendix A) contains a range of codes reserved for user-defined exceptions; this allows the handling of these exceptions to be integrated into the handling of all other (i.e. system-defined) exceptions. The same ports are used in both cases, and the interface to handlers is identical. An advantage of this approach is that user-defined exceptions can immediately be recognized as such, even by debuggers that cannot decode the machine-dependent fields which identify the exact exception. Generation of user-defined exceptions is facilitated by a MiG (Mach interface Generator) stub routine that implements the exception rpc (in turn this routine is generated automatically from an interface description of the exception rpc). User code that detects an exception simply obtains the appropriate exception port from the kernel and calls this stub routine; the stub routine handles the rpc and returns a return code from the handler. Alternatively, users may use the MiG exception interface with their own exceptions and exception ports; this approach may be advantageous for applications that handle only user-defined exceptions.

7. Implementing Error Handlers

Error handlers are supported by thread exception ports and invoked by remote procedure calls on those ports. An error handler is associated with a thread by registering a port on which the error handler receives exception rpcs as the thread's exception port. This registration causes all exceptions occurring in the thread to invoke rpcs to the error handler's port. Since most error handlers cannot handle all possible exceptions that could occur, they must check each exception and forward it to the corresponding task exception port if it cannot be handled. This forwarding can be performed by obtaining the exception port for the task specified in the initial message and sending the initial message there. Alternatively the error handler can return a failure code in the reply message; this causes the sender of the initial message to reinitiate the rpc using the task exception port.

Implementation of error handlers requires additional functionality beyond completing the rpc. This functionality is supported by separate Mach kernel primitives that can also be used by other applications. The most common actions and corresponding primitives are:

- Read/Write register state - **thread_get_state**, **thread_set_state**.
- Read/Write memory state - access memory directly within task, otherwise **vm_read**, **vm_write**.
- Terminate thread - **thread_terminate**.
- Resume thread - send reply message to complete rpc (**msg_send**).

Some applications may require that error handlers execute in the context of (i.e. on the stack of) the thread that caused the exception (e.g. emulation of Unix signal handlers). Although this appears to conflict with our restriction of never executing an error handler in the context of the victim thread, it can be implemented by using a system-invoked error handler to set up the application's handler. Specifically the error handler invoked by the exception rpc modifies the victim thread so that the application's handler is executed when the thread is resumed. Unwinding the stack when the application's error handler finishes is the responsibility of the application implementor.

8. Implementing Debuggers

Debuggers are supported by the task exception ports; exceptions invoke debuggers via remote procedure calls on those ports. A debugger is associated with a task by registering a port on which the debugger receives exception rpcs as the task's exception port. An exception rpc only stops the victim thread pending rpc completion; other threads in the task continue running. This has two consequences:

- If the debugger wishes to stop the entire task, a **task_suspend** must be performed. A straightforward way to accomplish this is to do it inside the exception rpc and then complete the rpc; the victim thread cannot resume execution upon rpc completion because its task has been suspended.
- Multiple exceptions from a multi-threaded task may be outstanding for the debugger on a single debugger invocation. If the debugger does not handle these pending exceptions for the task, some may appear to occur at impossible times (e.g. breakpoint occurs after user removes it).

The Mach exception handling facility is one small component of the kernel that can be used by debuggers. The various actions required to support debuggers are implemented via general purpose primitives that also support other applications. Some of the more important debugger actions and corresponding kernel primitives are:

- Detect event - **msg_receive** - System components that generate or detect external events (e.g. interrupt character on a terminal) signal the events by sending messages.
- Read and write application memory (includes setting breakpoints) - **vm_read**, **vm_write**.
- Read and write application registers (includes setting single-step mode if available) - **thread_get_state**, **thread_set_state**.
- Continue application - Task and thread control primitives.
- End debugging session - **task_terminate**.

More detail on these and other useful kernel primitives can be found in appendix B.

Exceptions that invoke error handlers via thread exception ports are not visible to debuggers. A debugger that wishes to detect error handler invocation can insert one or more breakpoints in the error handler itself; exceptions caused by these breakpoints will be reported to the debugger.

9. Communication Properties and Extended Functionality

The Mach communication facility has two important properties that facilitate extended functionality of systems based on it:

- Independence - A port is an independent entity from the tasks which use it to communicate. Port rights can be exchanged in messages, and are tracked by the kernel to maintain protection.

- Network Transparency - User-mode network message servers transparently extend the Mach communication facility over the network. The server on the remote machine creates proxy ports to represent the ports exported to it over the network. A message destined for a remote port is sent to the local proxy port; the local network message server receives the message and forwards it over the net to the remote remote network message server which sends it on to the actual destination port. This forwarding is invisible to the senders and receivers of messages.

Extended functionality can be obtained from the Mach exception handling facility and kernel primitives because the Mach communication facility is used as an implementation base. In addition to the use of messages in the exception handling facility, the kernel interface itself is message based; most system calls are actually invoked by a message rpc to the kernel.

9.1. Debugger Attachment

The independence property described above allows Mach to support debugger attachment and detachment without change to the kernel. Traditional Unix systems require that the debugged process be the child of the debugger; this makes it impossible to debug a process that was not started by the debugger. Subsequent implementors have expended considerable effort to implement an attach primitive that allows a debugger to attach to a previously started process and debug it; this allows analysis of failures that may not be repeatable. Similarly these systems allow a debugger to detach from a running process and exit without affecting the process. No change is required in our design to support this functionality; the debugger need only obtain the port representing the task to be debugged, and may then use all of the primitives previously discussed to debug that task. A debugger can detach from a task by resetting the task's exception port to its former value; there is no other connection between the debugger and task being debugged.

10. Parallel and Distributed Debugging

Our design also supports parallel and distributed debugging without change. There are several cases to be considered based on the structure of the debugger and the application being debugged. In all of these cases the debugger itself may be a parallel and/or distributed application consisting of multiple tasks and/or threads.

For parallel applications composed of multiple threads within a single task, a debugger need only register its exception rpc port as that task's exception port. Multiple concurrent exceptions result in multiple rpc invocations being queued to that port; each invocation identifies the thread involved. Mach's communication facility allows the debugger to accept all of these rpcs before responding to any of them, and to respond to them in any order. (Of course the debugger must keep track of the rpcs and make sure they are all responded to when continuing the application.) A straightforward implementation is to suspend the task in response to the first rpc, and then complete all pending exception rpcs recording the threads and exceptions involved. The exceptions can then be reported to the user en masse.

For parallel applications composed of multiple tasks within a single machine, only minor changes to the above debugger logic are required. The debugger must now register its exception rpc port as the task exception port for each task, and may choose to identify components of the parallel application by tasks instead of threads. Suspending or resuming the entire application now requires an operation on each task. If the application dynamically creates tasks, an additional interface to report these new tasks to the debugger may be required so that the new tasks can be suspended and resumed by the debugger.

Network transparency allows the components of a debugger and the debugged application to be spread throughout a network; all required operations extend transparently across the network. This supports a number of possible debugging scenarios:

- Application and debugger on separate hosts.
- Debugging of a distributed application. The debugger does not require modifications beyond those needed to deal with applications composed of multiple tasks.
- The debugger itself can be distributed over the network.

The last scenario is useful for implementing fast exception response in a debugger for applications that runs in parallel on several distributed hosts; if the exception rpc stays within the host, suspending of all application components on that host can be done faster.

11. Gdb Enhancements

As part of this work we have used the Mach exception handling facility and other Mach kernel primitives to enhance Gdb (the Gnu Debugger) [9] for debugging multi-threaded tasks. This enhanced version of gdb operates at the task level (i.e. any exception causes it to suspend the entire task). A notion of the "current thread" has been added to the user interface; this thread is used by any thread-specific command that does not specify a thread. New commands are provided to list the threads in the task, change the "current thread", and examine or control individual threads. Thread-specific breakpoints are supported by logic that transparently continues the application from the breakpoint until the desired thread hits it. Implementation of attachment to running tasks as described in section 9.1 is in progress, as are changes to deal with multiple concurrent breakpoints.

The existence of multiple threads within a debugged task complicates gdb's execution control logic. In addition to the **task_suspend** required upon exception detection, resuming from a breakpoint becomes somewhat intricate. Vanilla gdb removes the breakpoint, single steps the process, puts back the breakpoint and continues. The enhanced version must ensure that only the thread at the breakpoint executes while performing the single step; this requires switching from task suspension to suspension of all of the threads except one and then back again before resuming the application.

The Mach exception handling facility is an important implementation base for the enhancements to gdb. Identification of the victim thread in the initial message avoids confusion over which thread in the process is being manipulated by ptrace(); without this identification it is necessary to compare the context accessed by ptrace() to all other thread contexts in the task to determine this. This identification also make it possible to handle multiple concurrent exceptions; all of the Unix primitives are restricted to one current signal per task, and hence preclude handling of multiple concurrent exceptions. Finally the independence of the debugger from the debugged application makes it possible to implement debugger attachment without kernel modifications; Unix requires extensive kernel modifications to achieve similar functionality [10]

12. Conclusion

This paper has discussed the design and implementation of exception handling facilities for general purpose operating systems. In the past these facilities have often been designed as afterthoughts to support a few specific applications. In contrast to this approach we have set forth a general model for the use of exceptions and designed an exception handling facility that implements the model in full generality. The resulting facility is both simpler and more powerful than its predecessors; among its major advantages is a clean separation between debuggers and error handlers.

We have also shown that significant benefits can be derived from the properties of the communication facility used for exception handling. The independence of the Mach communication facility from the applications using it to communicate provides direct support for debugger attachment without the kernel modifications required by previous implementations. The combination of this independence property with network transparency enables Mach to support parallel and distributed debugging without change to the operating system kernel.

Appendix A. Exception Classification

The Mach exception handling facility employs a new hardware-independent classification of exceptions. This is in contrast to previous systems (e.g. Unix), whose exception classifications are closely wedded to the hardware on which they were originally developed. Our new classification divides all exceptions into six classes based on the causes and uses of the exceptions; further hardware and software specific distinctions can be made within these classes as needed. The six classes are:

1. Bad Access - A user access to memory failed for some reason and the operating system was unable to recover. (e.g. invalid memory, protection violation)

2. Bad Instruction - A user executed an illegitimate instruction. (e.g. undefined instruction, reserved operand, privileged instruction)

3. Arithmetic - A user arithmetic instruction failed for an arithmetic reason. (e.g. overflow, underflow, divide by zero).

4. Emulation - A user executed an instruction requiring software emulation.

5. Software - A broad class including all exceptions intended to support software. These fall into three subclasses

    a. Hardware - Hardware instructions to support error detection. (e.g. trap on overflow, trap on subscript out of range)

    b. Operating System - Exceptions detected by operating system during system call execution. (e.g. no receiver on pipe) These are for operating system emulation (e.g. Unix). Mach does not use exceptions for system call errors.

    c. User - Exceptions defined and caused by user software for its own purposes.

6. Debugger - Hardware Exceptions to support debuggers. (e.g. breakpoint instruction and trace trap).

In cases of potential confusion (e.g. is a bad opcode a bad instruction or an instruction requiring emulation?) the correct classification is always clear from the intended uses of the instruction as determined by the hardware and system designers.

Two machine dependent fields are used to identify the precise exception within a class for flexibility in encoding exception numbers. Two fields are needed for emulation instructions containing a single argument (one for the instruction, one for the argument), but we have also found them useful for constructing machine-dependent exception classifications (e.g. use one field to hold the trap number or vector, and the other to distinguish this trap from the others that use this number or vector). Cases in which two fields do not suffice require a separate interface to extract the additional machine-dependent status.

Appendix B. Kernel Interface

This appendix lists the Mach system calls that relate directly to the exception handling facility and other calls of particular interest to users of the facility. This is a partial listing of system calls with brief descriptions; a complete listing with full documentation can be found in the Mach Kernel Interface Manual.

The following four operations are used to read and modify exception ports:

- **task_get_exception_port**(*task*, *exception_port*) - obtain task's exception port.

- **task_set_exception_port**(*task*, *exception_port*) - set task's exception port.

- **thread_get_exception_port**(*thread*, *exception_port*) - obtain thread's exception port.

- **thread_set_exception_port**(*thread*, *exception_port*) - set thread's exception port.

The above operations are implemented as macros that expand to the exception port version of the corresponding set_special_port or get_special_port calls.

The following additional kernel operations may be of interest to programmers writing error handlers and debuggers:

- Communication Operations
  - **msg_send**(*msg_header*, *option*, *timeout*) - send a message to a port.
  - **msg_receive**(*msg_header*, *option*, *timeout*) - receive a message from a port.
  - **msg_rpc**(*msg_header*, *option*, *reply_size*, *send_timeout*, *reply_timeout*) - send a message to a port and receive the reply.
  - **port_allocate**(*task*, *port*) - allocate new port in task.
  - **port_deallocate**(*task*, *port*) - deallocate port from task.
- Task Operations
  - **task_create**(*parent_task*, *inherit_memory*, *child_task*) - Create new child task from parent task.
  - **task_terminate**(*task*) - Terminate execution of the task.
  - **task_suspend**(*task*) - Suspend execution of the entire task.
  - **task_resume**(*task*) - Resume execution of the entire task.
  - **task_threads**(*task*, *thread_list*, *thread_count*) - Obtain a list of all threads in the task.
- Thread Operations
  - **thread_create**(*task*, *thread*) - Create a new thread in the specified task. The new thread is created in a suspended state.
  - **thread_terminate**(*thread*) - Terminate execution of the thread.
  - **thread_suspend**(*thread*) - Suspend execution of the thread.
  - **thread_resume**(*thread*) - Resume execution of the thread.
  - **thread_get_state**(*thread*, *flavor*, *state*, *count*) - Read registers in thread's context. Flavor distinguishes between multiple register sets (e.g. general and floating-point) in hardware.
  - **thread_set_state**(*thread*, *flavor*, *state*, *count*) - Write registers in thread's context.
- Virtual Memory Operations
  - **vm_read**(*task*, *address*, *size*, *data*, *count*) - Read another task's (virtual) memory.
  - **vm_write**(*task*, *address*, *data*, *count*) - Write another task's (virtual) memory.

References

[1] Accetta, M.J., Baron, R.V., Bolosky, W., Golub, D.B., Rashid, R.F., Tevanian, A., and Young, M.W.
Mach: A New Kernel Foundation for UNIX Development.
In *Proceedings of Summer Usenix*. July, 1986.

[2] *American National Standard Reference Manual for the Ada Programming Language*
ANSI, New York, NY, 1983.
ANSI/MIL-STD 1815A-1983.

[3] Bach, M. J.
The Design of the Unix Operating System.
Prentice-Hall, Inc., Englewood Cliffs, NJ, 1986.

[4] *VAX Architecture Handbook*
DEC, Marlboro, MA, 1981.

[5] Dobberpuhl, D.W., Supnik, R.M., and Witek, R.T.
The MicroVAX 78032 Chip, A 32-Bit Microprocessor.
Digital Technical Journal (2):12-23, March, 1986.

[6] Ousterhout, J.K.
Partitioning and cooperation in a Distributed Multiprocessor Operating System: Medusa.
PhD thesis, Carnegie Mellon University, 1980.

[7] Rashid, R.F., and Robertson, G.
Accent: A Communication Oriented Network Operating System Kernel.
In *Proceedings of the Eighth Symposium on Operating System Principles*, pages 64-75. December, 1981.

[8] Siewiorek, D.P., Bell, C. G., and Newell, A.
Computer Structures: Principles and Examples.
McGraw-Hill Book Company, New York, NY, 1982.

[9] Richard M. Stallman.
GDB Manual: The GNU source-Level Debugger
First edition, Free Software Foundation, 1987.

[10]
Sun Operating System Version 3.2.
Operating System Kernel Source Code.

Table of Contents