Mach Interfaces to Support Guest OS Debugging

Rand A. Hoven

Hewlett Packard, Inc.
Apollo Systems Division
rand@apollo.hp.com

ABSTRACT

Many operating systems that may be implemented as servers on Mach will need to provide specialized debugging features. In general, these features can be provided just by creating the complete interface for the guest OS. However, programs operating under one of these guest OSs may desire to use features that are available only under Mach. In many cases the guest OS's debugging interface is inadequate to deal with programs using these Mach features. Furthermore, it is also desirable to be able to implement one debugger that can debug any program regardless of what guest OS the program and debugger are running under. This paper presents a mechanism by which a guest OS can present debug information to a debugger running under any guest OS.

1. Introduction

Mach is intended to be a µkernel. As such it only provides some basic features such as ports, tasks, threads, and memory objects. It is intended that any other functionality that a program might require be provided by user space servers. The program contacts the server with a request which the server fulfills. [1].

A logical conclusion from this concept is that any operating system can be emulated on top of the µkernel just by creating a server that provides all of the semantics of the desired OS. In this case the µkernel would be the host OS and the emulated operating system would be the guest OS. [2].

The debug interfaces provided by a guest OS will also be presented as part of its overall interface. This is sufficient if a program being debugged stays within the confines of the features of the guest OS. Many new programs will be written with the intention of running under a Mach guest OS and will use Mach features in addition to those traditionally provided by the guest OS. One of these features is the ability to have multiple threads in a single task. Not all guest OS debug interfaces are capable of dealing with this situation. These guest OSs will have to be extended to provide support for debugging these programs. It is also desirable to be able to provide a system with one debugger that can manipulate any program in the system regardless of the guest OS the debugger and the debugged program run under. All of this suggests that a uniform interface for debugging should be developed.

2. The Requirements

A debugger is a application that can control and modify the execution state of one or more targets. A target is a task with one or more threads executing in it. An application is a collection of tasks and threads that are cooperating to accomplish something. Currently, there are a number

of debuggers that are quite good at debugging an application that uses a single thread in a single task. While most of these debuggers often invoke the application themselves, some are able to attach to an application that has been invoked by a third party. Although the set of applications that can be debugged by these debuggers is large, it does not include some of those most difficult to debug. Some applications that cannot be handled by these debuggers include multi-threaded applications and distributed applications. [3]. "Distributed applications" refer to an application that requires more than one task to function. These tasks can be on the same or different machines. A debugger that is capable of debugging all of these applications must be able to do the following:

- Create a debugging relationship with any of the user's tasks with no changes in the task's operating environment.
- Capture the creation of a new task or thread and debug it in addition to the original target.
- Detect that new code has been loaded into the target before any of it is executed.
- Examine a task's state before it is destroyed.
- Control the delivery of asynchronous events.

In order to support the above functionality, the OS must provide the debugger with the following capabilities:

- Read and write the target task's memory.
- Read and write the target thread's registers.
- Suspend and resume execution of the target threads.
- Notify the debugger when the executable image has changed.
- Notify the debugger when a target creates new tasks.
- Notify the debugger when a target creates new threads of execution in the current task or another task.
- Notify the debugger when the last thread of a task dies, before the task is destroyed.
- Notify the debugger when asynchronous events are delivered to the target threads.
- Support a debugging relationship between arbitrary tasks.

3. Viability of the Ptrace Interface

The UNIX† Operating System's *ptrace()* interface was examined to see if it could be extended to fulfill these requirements. One common extension to *ptrace()* is to allow it to attach to a target that is not a direct descendant of the debugger. This requires changes to various system tables so the target will be correctly identified when the debugger performs a *wait()* system call.

Normally, the only information passed to the debugger through *wait()* is the process id of the target and the signal that stopped it. This is sufficient when the only reason to report status to the debugger is the occurrence of a signal. However, additional information needs to be passed when a fork, exec, or other system operation requires the debugger be notified. This information could be passed in the 16 unused bits of the wait structure or obtained by a new *ptrace()* operation.

---

† UNIX is a registered trademark of UNIX System Laboratories, Inc.

The major problem is how to name a particular thread in the target task using the *wait()* and *ptrace()* calls. Using the process id is difficult because the UNIX guest OS associates multiple threads with one process. The naming problem can be avoided by stopping all of the threads when one of them takes a signal. Then any changes to thread state would only modify the thread to which the signal would be delivered. Unfortunately, stopping all of the threads in a task could cause some applications to fail due to timing considerations.

Lastly, using the *ptrace()* interface is only applicable to applications running under the UNIX guest OS. Applications running under other guest OSs cannot be debugged using it as there is no way to name them. These issues prompt consideration of other mechanisms.

4. Debugging the Mach Way

After looking at the *ptrace()* interface, the Mach method of debugging a process was examined. This approach combines Mach primitives with the debug interface provided by the guest OS.

4.1. Using Mach Primitives

The following ports are used by a debugger to debug a Mach program. The port representing the target's task port is used to manipulate the state of the target task. The ports representing the target's thread ports are used to manipulate the state of the target threads. And the ports representing the target's task exception port is used to detect faults that occur while the target is running. To debug a target process the debugger obtains the task port of the target. This is currently done by calling *task_by_unix_pid()*. The task port is then used to obtain the thread ports of the threads executing in the target task. Once this is done, the debugger creates a port and calls *task_set_exception_port()* to register send rights to the port as the target's task exception port.

The exception port is used to send messages to the debugger concerning any machine or software exceptions that occur in the target task. These exceptions include but are not limited to: breakpoints, bad memory accesses, invalid instructions, and other synchronous faults. An exception message has a fixed shape containing the the type of exception as well as a code and a subcode that further define the exception. The thread that generated the exception will remain blocked until a reply message is received. Once a reply is received, the thread will continue execution.

To manipulate the state of the target, a debugger may use the following Mach primitives: the *vm_read()* call is used to read pages in the target task; the *vm_write()* call is used to write pages in the target task; the *vm_protect()* call is used to allow breakpoints to be written to read only pages; the *thread_get_state()* call is used to read the registers and other state information concerning a thread; the *thread_set_state()* call is used to write the above information. The thread state will also include elements that control single step execution and other hardware/OS assistance useful for debugging programs. In addition *task_suspend()*, *task_resume()*, *thread_suspend()*, and *thread_resume()* may be used by the debugger to control the target. [4].

4.2. Combining Mach Primitives with the Guest OS's Primitives

To debug a program, the Mach primitives are used in conjunction with the guest OS's debugging mechanism. When using the guest OS's debugging mechanism, the debugger will receive debug information from two sources: from the source supported by the guest OS, and from the target's exception port.

Each guest OS's interface and semantics will produce its own set of problems. In the UNIX guest OS for example, the *ptrace()* mechanism can be used to manipulate only the immediate child of the debugger. [5].

5. Filling in the Holes

The Mach primitives provide a good mechanism for debugging a program. However, they have no concept of what the guest OS may do that will affect a debug session. For example, the first thing that a guest OS will generally do in a debugging session is load the program into the target's memory. A new mechanism is needed that will provide a general way for a guest OS to notify the debugger that a guest OS specific event has occurred. The approach used is to have the guest OS generate a Mach remote procedure (RPC) call on behalf of the target thread to the debugger when ever one of these events occurs. This is similar to the existing exception mechanism except that the guest OS generates the messages instead of the µkernel.

5.1. The Trace Event

The basic component of this new mechanism is the trace event. A trace event is a guest OS defined action that the target performs or is subjected to, which may be of interest to a debugger. When a trace event occurs, a trace event message containing the details of the trace event is sent to the debugger via an RPC. The shape and contents of a trace event message and the reply to each message are defined by the guest OS generating the message. A guest OS may use a different shape message for each event it generates. A trace event will include at least a reply port as well as the thread and task ports of the thread that generated the event. In addition, a trace event may carry other information relevant to the event. This might include the thread port of a newly created thread, the name of the file containing newly loaded code, or any other piece of relevant information. The data in the reply may be used by the guest OS to modify the behavior of the target process. These messages are sent from the target to the debugger on the trace event port. Replies to these messages must be sent on the port contained in the trace event message.

5.2. The Trace Event Port

Each guest OS will maintain a registration of send rights to a port for each task being debugged. This port is called the trace event port and is used to send trace event messages from the target to the debugger. When a target creates a new task, the guest OS will propagate the registration of the trace event port to the newly created task. This task will then be a target of the same debugger as the task that created it.

5.3. The Trace Event Reply Port

Each trace event message will include a reference to a port that is to be used as a reply port for the message. A target will have one reply port for each thread that is executing within it. It may have additional reply ports if system services are being provided by multiple servers. These reply ports may exist for the life of the debug session or they may only be valid long enough to receive the reply message.

5.4. The Trace Event List

For each target thread a guest OS may maintain a trace event list, a list of trace events that will be sent to a debugger. A debugger will be able to enable or disable events in this list. When a debugger attaches to a target, the event lists for the threads in the target will be empty. The debugger then enables the events it wishes to receive. A trace event message will be sent to the debugger only if the event is enabled in the trace event list and the the target has a valid port

registered as its trace event port. A trace event list is needed by an OS that generates a large variety of events that will only be of interest to a debugger under certain conditions. An event list can be used to facilitate backward compatibility. To do so, a debugger should enable only those events it is capable handling. If in the future more trace events are defined by the guest OS, an existing debugger will not be affected because the new events will not be sent to it. If a guest OS does not maintain a trace event list, all events will be sent to the trace event port if it is set. When a target creates a thread the trace event list of the new thread will be empty. Requests to get and set the elements of the trace event list are made by sending a message to the trace control port.

5.5. The Trace Control Port

Each guest OS will maintain a single trace control port. This port is used by debuggers to send messages to the guest OS. These messages include requests to attach to a task and requests to change the event list of a target thread. Send rights to the trace control port for a guest OS are registered with the net message server.

6. Establishing the Target/Debugger Relationship

Figure 1 describes the messages sent to establish the relationship among the various tasks in a debug session. The following messages will establish a debug relationship.

[1] In order for a guest OS to support trace events, it must register send rights to its trace control port with the naming service of the net message server. If only a single instance of the guest OS is running on the system then a name in the form of TRACE_*guestOS* is sufficient. If more then one instance is running then the name should be in the form TRACE_*guestOS.number*. The name and related port are registered with the *netname_check_in()* call.

[2] To start a debugging session the debugger must determine the name of the guest OS that is running the target. This information may be provided by the user, through the environment, or in a configuration file. Once the name of the guest OS is determined it is used to construct the name of the guest OS's trace control port. The debugger the calls *netname_look_up()* to get the trace control port.

[3] Upon receipt of rights to the trace control port, the debugger calls *trace_request()* to send a message to the guest OS requesting send rights to the task port of the target process. This message will contain the guest OS's identification for the target process, and a reference to the task port of the debugger. The guest OS will either return an error or the rights to the task port of the target. The reference to the debugger's task port is included so that the guest OS can make a decision on whether or not to allow the debug operation to occur. In the case where both the target and debugger are operating under the same guest OS, the guest OS can use the reference to the debugger's task port to determine the debugger's identity. Once the guest OS knows the identity of both the target and debugger, it can determine if the debug relationship is allowable.

[4] Once the rights to the task port are obtained, the debugger will call *trace_set_port()* to send a message to register the trace event port.

[5] The debugger may then call *trace_set_events()* to set the event list for each of the target threads.

[6] The debugger then calls *task_set_exception_port()* to register the task exception port.

Figure 2 shows the tasks and their relationship to each other through the various ports used by the debugger.

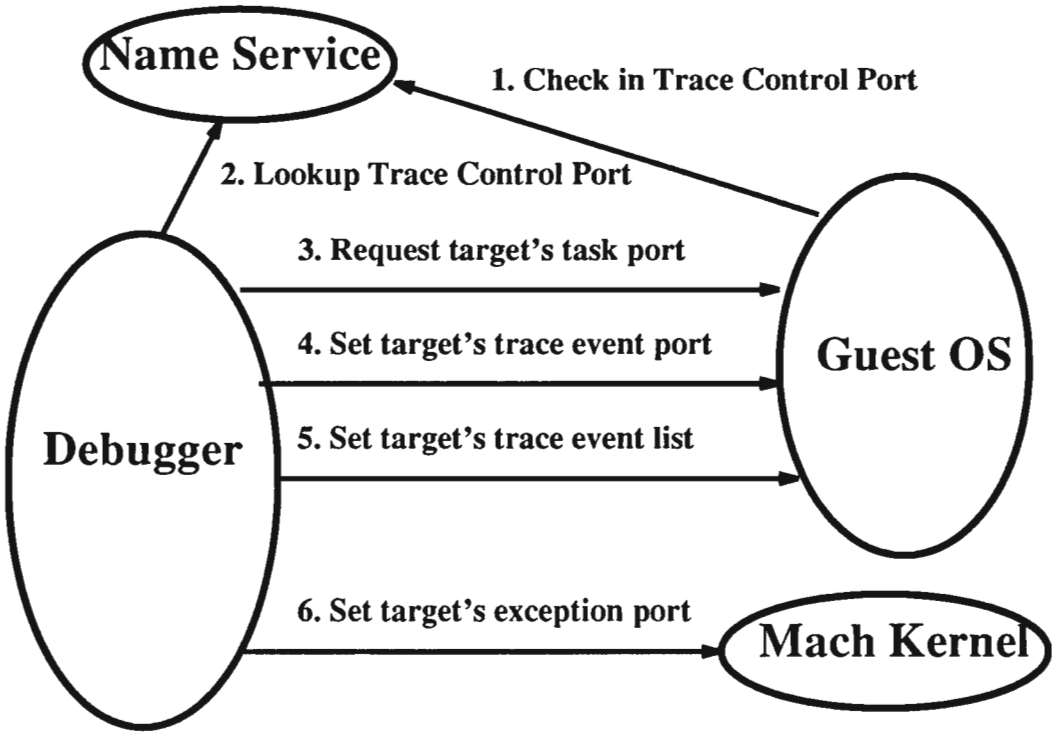

Figure 1.

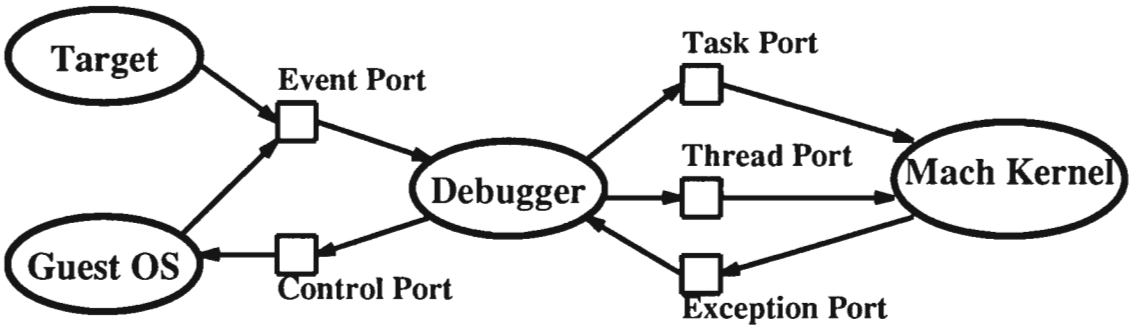

Figure 2.

7. Other Clients of a Debugger

A guest OS is not the only entity that may need to provide information to a debugger. Many languages have special operations that may be of interest. The trace event interface can be used by such a language to communicate with a debugger. For a language to take advantage of this, cooperation between the language's run time module (RTM) and the guest OS will be required. Whenever the the task becomes the target of a debugger the guest OS must pass a reference to the trace event port to the RTM. It must also notify the RTM when it is no longer the target of a debugger. Both of these notification can be delayed until such time as the target generates a language specific trace event. Exactly how the guest OS and RTM communicate will depend on the implementation of the RTM and guest OS.

8. Interface Details

This section describes the ports used by a debugger and the messages that can be sent to them.

8.1. Details on the Trace Control Port

The debugger has send rights to the port registered as the trace control port of the guest OS of the target. Messages sent on the port will be received by the guest OS. These procedures are defined in terms of the Mach Interface Generator (MIG). [6].

8.1.1. Type Definitions

The *trace_target_t* type is the place holder type for guest OS's identifier of a target under its control. Many guest OSs identify a process with a short integer. Some have more complicated schemes. Others only have one program running at a time. By using an array type definition we can accommodate all of these identifiers.

```
type trace_target_t      = array [*:64] of int;
```

The *trace_event_list_t* type is the place holder type for each guest OS's event list. Each guest OS may define a structure that allows the enabling and disabling of each event that the OS can generate. That structure will be passed in place of the *trace_event_list_t* array.

```
type trace_event_list_t = array [*:256] of int;
```

8.1.2. The Trace Request Message

The *trace_request()* RPC is used by the debugger to obtain the task port of the target program. The *trace_control_port* is the port obtained from the naming service of the net message service. The debugger's task port is provided so that the guest OS may make a decision about whether or not the debugger has permission to debug the target. The target identifier is the guest OS's representation of the identity of a process it controls. The *target_task* is returned if the call is successful.

```
routine trace_request(
              trace_control_port     : port_t;
              debugger_task          : task_t;
              target_identifier      : trace_target_t, IsLong;
        out   target_task            : task_t);
```

8.1.3. The Trace Set Port and Trace Get Port Messages

The *trace_set_port()* and *trace_get_port()* RPCs are used to set and inquire about the trace event port of a particular task. The *trace_set_port()* RPC establishes a port created by the debugger as the trace event port in the target_task. If another debugger is currently managing the target the old relationship is broken and this one created. This allows one debugger to pass off a target to another debugger. If the value of trace_event_port is the NULL_PORT then the target/debugger relationship is broken and all events are disabled. The *trace_get_port()* returns the trace event port for the target task.

```
routine trace_set_port(
            trace_control_port      : port_t;
            target_task             : task_t;
            trace_event_port        : port_t);

routine trace_get_port(
            trace_control_port      : port_t;
            target_task             : task_t;
        out trace_event_port        : port_t);
```

8.1.4. The Trace Set Events and Trace Get Events Message

The *trace_set_events()* indicates the events that a guest OS will trace. The *trace_get_events()* call returns the events that the guest OS is currently tracing. The *trace_event_list_t* is a structure of a form defined by the guest OS.

```
routine trace_set_events(
            trace_control_port      : port_t;
            target_thread           : thread_t;
            event_list              : trace_event_list_t, IsLong);

routine trace_get_events(
            trace_control_port      : port_t;
            target_thread           : thread_t;
        out event_list              : trace_event_list_t, IsLong);
```

8.2. Details on the Trace Event Port

The guest OS is responsible for propagating send rights of the trace event port to all of the tasks that will need it. In general this will include the target task and any system servers that will generate events on behalf of the target task. The target or server will keep track of the value of the trace event port. If the value is PORT_NULL, no trace event messages will be sent. If the value is not PORT_NULL, trace event messages will be sent if the event is enabled. At some point the debugger may deallocate the trace event port without notifying the guest OS. In this case the RPC will return an error status of SEND_INVALID_PORT. If this occurs the target or server should set the value of the trace of the trace event port to PORT_NULL and disable all events. The format and names of the event messages are determined by the guest OS.

9. Specifics for the UNIX Guest OS

The initial implementation of this mechanism was for the UNIX guest OS. The following are the type definitions and trace event RPCs. Most of the definitions are made in terms of MIG. Routine names will be proceeded with *report_* when called from the guest OS (the client), and with *catch_unix_* when called from the debugger (the server).

9.1. UNIX Trace Type Definitions

The following type definitions are used by the UNIX guest OS interface.

The *trace_target_t* type is replaced by the *pid_t* type as defined by the UNIX guest OS. The *target_identifier* argument of the *trace_request* call will have an argument of a pointer to a

pid_t passed instead of an argument of *trace_target_t* type. The UNIX_TARGET_COUNT is passed as the length of the target_identifier argument.

```
#define UNIX_TARGET_COUNT (sizeof(pid_t)/sizeof(int))
```

The *unix_event_list* structure represents the event list used by the UNIX guest OS. An argument of *unix_event_list_t* is passed as the *event_list* argument to the *trace_set_events()* and *trace_get_events()* call. This type is used instead of the *trace_event_list_t* place holder. The length of the event_list to be passed is UNIX_EVENT_LIST_COUNT. If any boolean member of the *unix_event_list* is set to TRUE the event messages will be sent. In the case of signals, if the signal is a member of the *signal_events* set a trace event will be sent.

```
typedef struct unix_event_list {
        sigset_t        signal_events;
        boolean_t       fork_event;
        boolean_t       exit_event;
        boolean_t       exec_event;
        boolean_t       load_event;
        boolean_t       unload_event;
        boolean_t       system_call_event;
        boolean_t       pthread_create_event;
        boolean_t       pthread_exit_event;
        boolean_t       pthread_context_assignment;
};
#define UNIX_EVENT_LIST_COUNT \
    (sizeof(struct unix_event_list)/sizeof(int))
typedef struct unix_event_list *unix_event_list_t;
typedef struct unix_event_list unix_event_list_data_t;
```

The following MIG structures are used to report the arguments to a UNIX system call and modify its return values. Upon surveying all UNIX system calls it was determined that no system call has more than six arguments. Some UNIX system calls (such as *fork()* and *getpid()*) have two return values. Unfortunately the only documentation on such calls is the UNIX kernel source itself.

```
type syscall_arguments_t        = array [6] of int;
type syscall_return_values_t    = array [2] of int;
```

9.2. The Signal Event

The delivery of a signal implies the termination of a task or the asynchronous calling of a signal handler in a thread. By reporting the delivery of a signal, the debugger gets the opportunity to correct the cause of the signal and allow the target to continue. Or it may allow the signal to be delivered and debugging of the signal handler to commence. Another possibility would be to ignore the signal or deliver it later with the *kill()* call.

This message is sent any time a signal is delivered that is a member of the *signal_events* field of the *unix_event_list*. *Signalin* is the signal that was generated. *Signalout* is the signal that will be delivered. If *signalout* has a value of 0, no signal will be delivered. The thread is blocked until the reply is received. If the reply message has an error status or *signalout* is invalid then

signalin will be delivered.

```
routine signal(
        requestport   trace_event_port   : port_t;
        replyport     clear_port         : port_t;
                      thread             : thread_t;
                      task               : task_t;
                      signalin           : int;
                out   signalout          : int);
```

9.3. The Fork Event

When a fork occurs, the debugger inherits another target program that it may debug. Reporting a fork event allows the debugger to clean up any breakpoints in the parent and child tasks. It also allows the debugger to gain control of the child before it executes any user space code.

Any time a fork occurs and the *fork_event* field of the *unix_event_list* in non-zero, this message is sent. *Child_thread* is the thread port of the newly created thread. *Child_task* is the task port of the newly created task. Both the parent thread and child thread are blocked until the reply is received.

```
routine fork(
        requestport   trace_event_port   : port_t;
        replyport     clear_port         : port_t;
                      thread             : thread_t;
                      task               : task_t;
                      child_thread       : thread_t;
                      child_task         : task_t);
```

9.4. The Exit Event

The exit event allows the debugger to examine the target's state before it is destroyed. This is useful for obtaining a final back trace.

Any time a task exits and the *exit_event* field of the *unix_event_list* is non-zero, this message is sent. The *exit_value* field contains the exit value that will be reported by *wait()* to the parent of the task. The thread and task still retain their states as they were just before the exit occurred. The task will exit when the reply message is received.

```
routine exit(
        requestport   trace_event_port   : port_t;
        replyport     clear_port         : port_t;
                      thread             : thread_t;
                      task               : task_t;
                      exit_value         : int);
```

9.5. The Exec Event

When a target execs a new executable image it, discards the old executable image. This includes any breakpoints that were set in the old image. The exec event allows the debugger to set new breakpoints in the target and complete preparation for debugging a new target.

Any time a task execs and the *exec_event* field of the *unix_event_list* is non-zero, the exec event message is sent. The state of the task and thread will be set to their initial state at the beginning of the new program. If the program was loaded by the OSF user space loader, [7]. the thread's state will be that of the first instruction of the loader. If the program was loaded entirely by the UNIX guest OS, the thread's state will be that of the first instruction of the program. *Argp* is the address in the target task that contains the characters of the arguments array. *Arg_size* is the total size of the array. *Envp* and *env_size* point to and give the length of the environment array in the target's address space. The format of these arrays is a set of characters terminated by a 0 byte followed immediately by the next set of characters, etc. The new image will execute when the reply is received.

```
routine exec(
        requestport   trace_event_port   : port_t;
        replyport     clear_port         : port_t;
                      thread             : thread_t;
                      task               : task_t;
                      argp               : vm_offset_t;
                      arg_size           : vm_size_t;
                      envp               : vm_offset_t;
                      env_size           : vm_size_t);
```

9.6. The Load Event

When the guest OS reports a load event, the debugger will have a chance to load the symbol table of the freshly loaded code. The debugger will also be able to set breakpoints before the new code is executed.

This event is generated by the OSF user space loader after it has loaded one or more related executable images and before the initialization routines of the images are called. The load event message will be sent if the *load_event* field of the *unix_event_list* is non-zero. Execution will continue when the reply is received.

```
routine load(
        requestport   trace_event_port   : port_t;
        replyport     clear_port         : port_t;
                      thread             : thread_t;
                      task               : task_t);
```

9.7. The Unload Event

The unload event provides symmetry with the load event. When the guest OS reports an unload event, a debugger can purge the symbols of the unloaded image from its symbol table. It will also be able to determine which breakpoints will be lost when the code is removed from memory.

This event is generated by the OSF user space loader after it has removed a module from the loaded package table list and before it has removed it from memory. The unload event message will be sent if the *unload_event* field of the *unix_event_list* is non-zero. Execution will continue when the reply is received.

```
routine unload(
        requestport  trace_event_port   : port_t;
        replyport    clear_port         : port_t;
                     thread             : thread_t;
                     task               : task_t;
                     module_id          : int);
```

9.8. The System Call Event

Intercepting system calls allows the debugger to emulate the behavior of the OS. This allows a debugger to run a program that should be run as a setuid program. The debugger intercepts any calls that are affected by the identity of the target and adjusts their results so that the target believes that it is actually a setuid program. In addition, by intercepting system calls, the debugger could create an activity log of an executing program and then allow the user to rerun the program having the system calls return the same results as in a previous run.

This event message is sent before a system call is executed whenever the *system_call_event* field of the *unix_event_list* is non-zero. The *call_number* is defined in <sys/syscall.h>. The *arguments* parameter is an array containing the arguments to the system call. If *fake_call* is returned with a value of false, the system call will execute normally. If *fake_call* is returned with a value of true, the system call will not be performed. In this case *error_value* and *return_values* will be returned to the target program. An *error_value* of zero will cause the *return_values* to be returned to the caller. Any other *error_value* will cause that error to be returned.

```
routine system_call(
        requestport  trace_event_port   : port_t;
        replyport    clear_port         : port_t;
                     thread             : thread_t;
                     task               : task_t;
                     call_number        : int;
                     arguments          : syscall_arguments_t;
                out  return_values      : syscall_return_values_t;
                out  error_value        : int;
                out  fake_call          : boolean_t);
```

9.9. The Pthread Create Event

This event allows the debugger to obtain control of a newly created thread before it executes any code, and gives the debugger the opportunity to set breakpoints or change the initial state of the new thread.

This event will be sent whenever a new pthread is created and the *pthread_create_event* field in the *unix_event_list* is non-zero. *New_thread* is the thread port of the newly create thread. *Thread_name* is the pthread identifier for the thread. There is no guarantee that a particular pthread will always use the same kernel thread to execute its instructions. *Thread_argument* is

the argument passed to the first procedure of the new thread. Both the current thread and the new thread will be blocked until a reply is sent.

```
routine pthread_create(
        requestport  trace_event_port   : port_t;
        replyport    clear_port         : port_t;
                     thread             : thread_t;
                     task               : task_t;
                     new_thread         : thread_t;
                     thread_name        : pthread_t;
                     thread_argument    : vm_offset_t);
```

9.10. The Pthread Exit Event

This event will give the debugger access to the state of a thread just before it exits. This is useful for obtaining a final back trace.

This event message will be sent whenever a pthread exits and the *pthread_exit_event* field in the *unix_event_list* is non-zero. The *thread_status* is the status passed to *pthread_exit()*. The thread will exit when the reply is received.

```
routine pthread_exit(
        requestport  trace_event_port   : port_t;
        replyport    clear_port         : port_t;
                     thread             : thread_t;
                     task               : task_t;
                     thread_name        : pthread_t;
                     thread_status      : vm_offset_t);
```

9.11. The Pthread Context Assignment Event

As mentioned while discussing the pthread create event, there is no guarantee of a one to one correspondence between a pthread and a kernel thread. In a single thread implementation of pthreads this message will be sent for every context switch. In an implementation that does guarantee a one to one correspondence between pthreads and kernel threads, this message will never be sent. Other implementations will send this message whenever the pthread that is executing in a kernel thread changes. This message will only be sent if the *pthread_context_assignment* field in the *unix_event_list* is non-zero. *Old_pthread* is the pthread that is losing the use of the *thread*. *New_pthread* is the pthread that is gaining the use of *thread*. If either of these pthreads is the value NO_PTHREAD then the parameter does not refer to any pthread.

```
routine pthread_context_assignment(
        requestport  trace_event_port   : port_t;
        replyport    clear_port         : port_t;
                     thread             : thread_t;
                     task               : task_t;
                     old_pthread        : pthread_t;
                     new_pthread        : pthread_t);
```

10. Example of a Unix Debugger

The following routine, *unix_target()*, will establish a debug relationship with a target application running under the UNIX guest OS. The *unix_target()* routine will then dispatch events to the appropriate MIG generated routines, *exc_server()* and *unix_trace_server()*. These routines will then make call backs to the *catch_exception_raise()*, *unix_catch_signal()*, *unix_catch_fork()*, and *unix_catch_exec()* routines.

```
unix_target(
    pid_t                   pid
) {
    port_t                  control_port;
    port_t                  event_port;
    port_t                  exception_port;
    port_t                  client_ports;
    port_t                  target_task;
    kern_return_t           ret;
    thread_array_t          thread_list;
    unsigned int            count;
    unsigned int            ii;
    unix_event_list_data_t  event_list;
    struct dummy_message    request;
    struct dummy_message    reply;

    /* allocate ports needed to receive events and exceptions */
    ret = port_allocate(task_self(), &event_port);
    ret = port_allocate(task_self(), &exception_port);
    ret = port_set_allocate(task_self(), &client_ports);
    ret = port_set_add(task_self(), client_ports, event_port);
    ret = port_set_add(task_self(), client_ports,
        exception_port);

    /* initialize the event list structure */
    bzero(&event_list, sizeof(event_list));
    sigfillset(&event_list.signal_events);
    event_list.fork_event = TRUE;
    event_list.exec_event = TRUE;

    /* get the task port of the target task */
    ret = netname_look_up(name_server_port, "",
            "TRACE_UNIX", control_port);
    ret = trace_request(control_port, task_self(),
            &pid, UNIX_TARGET_COUNT, target_task);

    /* setup the debugger/target relationship */
    ret = task_suspend(target_task);
    ret = trace_set_port(control_port, target_task, event_port);
    ret = task_threads(target_task, &thread_list, &count);
    for (ii = 0; ii < count; ii++)
        ret = trace_set_events(control_port, thread_list[ii],
```

```
                &event_list, UNIX_EVENT_LIST_COUNT);
    ret = task_set_exception_port(target_task, exception_port);
    ret = task_resume(target_task);

    /* dispatch events from the target to the MIG servers */
    for(;;) {

        /* wait for and event message or exception message */
        request.Head.msg_local_port = client_ports;
        request.Head.msg_size = sizeof(request);
        ret = msg_receive(&request.Head, MSG_OPTION_NONE, 0);

        /* pass the message to the correct MIG server routine */
        if (request.Head.msg_local_port == exception_port)
                exc_server(&request, &reply);
        else if (request.Head.msg_local_port == event_port)
                unix_trace_server(&request, &reply);

        /* send the reply back to the target */
        reply.Head.msg_local_port = request.Head.msg_local_port;
        reply.Head.msg_remote_port =
                request.Head.msg_remote_port;
        ret = msg_send(&reply.Head, MSG_OPTION_NONE, 0);
    }
}

/* exc_server's call back for exceptions */
catch_exception_raise(
    port_t          exception_port,
    port_t          clear_port,
    thread_t        thread,
    task_t          task,
    int             exception,
    int             code,
    int             subcode
) {
    /* code to handle exceptions */
}

/* unix_trace_server's call back for signal events */
unix_catch_signal(
    port_t          event_port,
    port_t          clear_port,
    thread_t        thread,
    task_t          task,
    int             signalin,
    int             *signalout
) {
    /* code to handle signals */
```

```
}

/* unix_trace_server's call back for fork events */
unix_catch_fork(
    port_t          event_port,
    port_t          clear_port,
    thread_t        thread,
    task_t          task,
    thread_t        child_thread,
    task_t          child_task
) {
    /* code to handle forks */
}

/* unix_trace_server's call back for exec events */
unix_catch_exec(
    port_t          event_port,
    port_t          clear_port,
    thread_t        thread,
    task_t          task,
    vm_offset_t     argp,
    vm_size_t       arg_size,
    vm_offset_t     envp,
    vm_size_t       env_size
) {
    /* code to handle execs */
}
```

11. Limitations

When using this interface there are several issues to keep in mind. These include process identity and UNIX signal delivery.

11.1. Identity

One such issue is that of process identity. The µkernel has no concept of identity. Only the guest OS controlling a process knows the identity of the process. However, the guest OS does not control all access to a task. If another process has send rights to the task kernel port of a process, the guest OS cannot change the identity of the process. Doing so would grant access to the new identity to the other process holding rights to the target's task kernel port. Furthermore, the guest OS cannot determine which task has send rights to the port and what its identity is. For the UNIX guest OS, if a debugger is attached to a process, then the process cannot be allowed to change its identity by executing a setuid program. This is true even if the other task holding send rights to the target's task kernel port has appropriate privileges.

11.2. UNIX Signal Delivery

In the past, UNIX debuggers received notification of a signal near the time it was raised. The debugger was notified of the signal regardless of whether it was ignored, held, handled or left at the default action. This allowed the debugger to share the same controlling terminal as the

target. When the user hit the interrupt key, the debugger noticed because the target stopped. This stop occurred regardless of what the target wanted to do with the signal.

The trace mechanism only notifies the debugger when a signal is delivered. This allows the target to run more closely to the way it would when running without a debugger attached. However, if the target is ignoring the interrupt signal, the debugger will not notice when the user hits the interrupt key. To solve this problem, the debugger will have to arrange another avenue of input from the user for itself.

12. Conclusions

This interface provides a mechanism for debugging programs that can be easily extended in the future. Currently, only the UNIX OS interface has been designed. It was recently extended to deal with the OSF user space loader and pthreads. Other new functionality in the guest OS may be added with equal ease. This can be done in a manner compatible with existing debuggers through careful use of the event lists. By having the debugger clear the data of an event list and then turning on the events it desires, it will never receive events it does not understand.

The behavior of target programs can be kept very close to their behavior when running without a debugger. Events are reported at as late a moment as possible. Timing problems are reduced by enabling or disabling individual events. The costs will only be paid for events that the debugger has actually enabled. A program that has a debugger attached to it but with no events enabled should run with no changes in behavior.

By involving the net message server in the arrangement of debugger and target relationships it becomes possible for a program running on one machine to debug a program on another machine. In the future, if the problem of verifying identity between different guest OSs is solved, it will be possible for a debugger running under one guest OS to debug a target running under a different guest OS.

Acknowledgements This work benefited from the conversations I have had with Nawaf Bitar, Tracy Hoover, Dave Leblang and Dan McCue.

References

1. Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young, ''Mach: A New Kernel Foundation for UNIX Development,'' *USENIX Conference Proceedings*, July 1986.
2. David Golub, Randall Dean, Alessandro Forin, and Richard Rashid, ''Unix as an Application Program,'' *USENIX Conference Proceedings*, Anaheim, CA, June 1990.
3. Hector Garcia-Molina, Frank Germano, Jr., and Walter H. Kohler, ''Debugging a Distrbuted Computing System,'' *IEEE Transactions on Software Engineering*, vol. SE-10, No. 2, March, 1984.
4. David L. Black, David B. Golub, Karl Hauth, Avadis Tevanian, and Richard Sanzi, ''The Mach Exception Handling Facility,'' *CMU Technical Papers*, April 1988.
5. Deborah Caswell and David Black, ''Implementing a Mach Debugger For Multithreaded Applications,'' *USENIX Conference Proceedings*, Washington, D. C., January 1990.
6. Richard R. Draves, Michael B. Jones, and Mary R. Thompson, ''MIG - The MACH Interface Generator,'' *CMU Technical Papers*, August 1987.
7. Larry W. Allen, Harminder G. Singh, Kevin G. Wallace, and Melanie B. Weaver, ''Program Loading in OSF/1,'' *USENIX Conference Proceedings*, Dallas, TX, January 1991.