02 December 1992

Debugging an object−oriented system using the Mach interface

Youcef Laribi [(1)]

Xavier Rousset de Pina [(2)]

Abstract:

This paper describes an extension of the Gnu debugger (*gdb*), the debugger provided on OSF/1 MK. The extended *gdb* enables its users to debug both OSF/1 MK processes and Mach tasks not belonging to the OSF/1 environment. This extended *gdb* also has the capability for debugging code dynamically loaded using either the OSF/1 loader or any specific application loader. This new debugger was developed to fulfill the debugging requirements of Guide, an object oriented system implemented on top of Mach 3.0 and which also uses OSF/1 MK services.

1 Introduction

Microkernels provide the operating system designer with a high−level portable virtual machine together with a model of implementing operating systems as one or multiple cooperating servers running on the same or different machines.

In this environment, an application designer may use the basic features of the microkernel as well as services supplied by existing servers (e.g operating system servers, database servers, ...). This is the way the Guide system was designed and is currently being implemented.

Guide is a distributed object oriented platform which aims to provide a generic support for object−oriented languages such as the Guide language [8] and C++. The implementation of the platform uses the features provided by Mach 3.0 such as tasks, threads, ports together with Unix−like services provided by the OSF/1 MK server such as I/O interface or X windowing environment. Furthermore, the object memory management uses dynamic linking and loading of code and data encapsulated in objects.

The standard version of the Gnu debugger, *gdb*, which is available on OSF/1 MK, does not know about Mach 3.0 features (e.g multiple threads) and so cannot control their use. It also cannot debug dynamically loaded code, except for shared libraries. The lack of such a debugger under the OSF/1 MK environment quickly proved an obstacle.

---

(1) OSF RI Grenoble (laribi@gr.osf.org)

(2) Bull−IMAG/Systems lab. (rousset@imag.fr)

This paper reports on the extensions we made to OSF/1 *gdb* to handle the debugging requirements of the Guide project, which are :

- Debugging Guide tasks not belonging to the OSF/1 environment.
- Debugging OSF/1 processes or Guide tasks with dynamically loaded code and data.

Similar work has been done at CMU to achieve the first goal, in the framework of the BSD server project. This work has been of great help. However, we have aimed at structuring the debugging interface in a more generic way (See 4).

Outline of the paper: Section 2 briefly describes the Guide system architecture over OSF/1 MK and Mach 3.0. Section 3 outlines the requirements for *gdb* extensions. Section 4 describes our debugger interface and gives some insight into how it is implemented. Debugging dynamically loaded code is subject of section 5, and we present some conclusions in section 6.

2 The Guide architecture on top of OSF/1 MK and Mach 3.0

The aim of the Guide project (Grenoble Universities Integrated Distributed Environment) is to explore distributed computing in terms of objects for a set of heterogeneous workstations interconnected via a local area network. The system is targeted towards cooperative applications such as document processing and program development.

A first implementation of Guide, called Guide−1, was done over Unix [2] it supports the Guide object−oriented language [8]. We now are in the process of implementing a newly designed version of the Guide system, called Guide−2 on top of OSF/1 MK which uses directly Mach 3.0 features in order to implement the distributed shared object memory and the execution structure of Guide−2. This new version aims to provide a generic support for object−oriented languages; the intended languages are C++ (with extensions for distribution and persistence) and Guide. We limit our presentation of Guide to the part relevant to the debugger. A thorough description of the system can be found in [5] and [6]. We start by a brief presentation of the execution model of Guide then we describe its implementation on top of Mach 3.0 and OSF/1 MK.

2.1 The Guide−2 execution model

The Guide−2 execution model defines two entities : jobs and activities. A job is composed of sequential threads of control called activities, which operate on passive objects. A job may span several physical nodes, it may dynamicaly extend itself or shrink, according to the pattern of object invocation. The execution of an activity consists of successive invocations of methods on objects.The invocations may take place on any node in the system. A program in the Guide−2 system is typically made up of one job .

The address space of a job is composed of several contexts. A context is an homogeneous address space (virtual memory) localized on a node. Such a decomposition is due to distribution (i.e.,a job has at least one context per visited node) and protection (i.e., only objects having the same owner can be mapped in the same context).

After its creation, the job is made of a unique context (the initial context) with the program mapped in, and an activity (called the main activity) processing the main entry point of the program. This activity may either create other activities in the job or start other jobs.

Activities within the same job or different jobs communicate using shared objects. Guide−2 provides several mechanisms to implement object sharing between jobs (e.g, single copy for object instances, multiple copies for code libraries and classes, synchronized access).

2.2 Implementation

Mach 3.0 abstractions provide a good fit to the Guide−2 requirements. Objects (instance−objets, class−objects, code libraries) are clustered and dynamically associated with Mach 3.0 memory segments. All job contexts (on the same node or on different nodes), except the initial context, are handled by Mach 3.0 tasks. The initial context, in which the program is mapped, is handled by an OSF/1 MK process. The program is an OSF/1 executable binary.

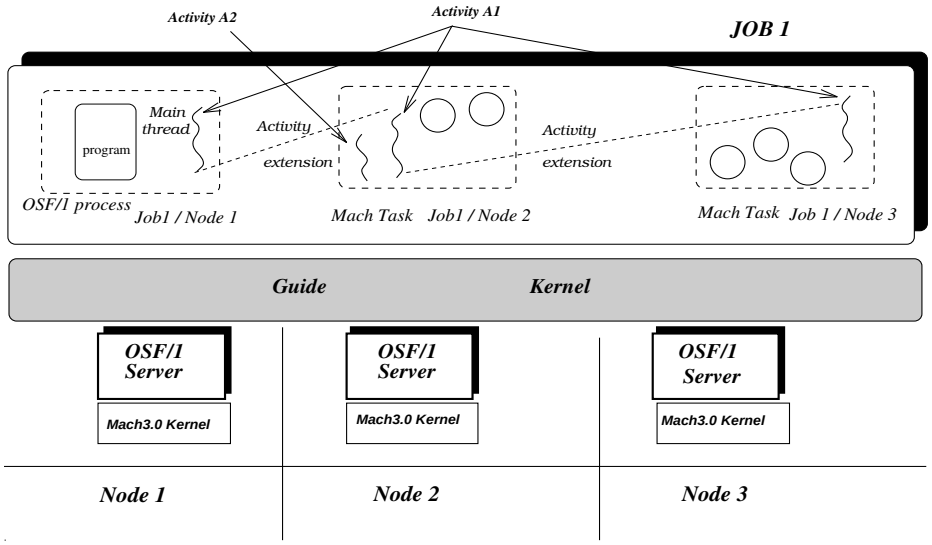

fig.1 job implementation.

The distributed shared object memory is implemented using the Mach 3.0 memory management facility. External pagers cooperate to manage the execution space (the set of objects that are mapped in at least one context) and the secondary storage (backup object storage).

On every Guide−2 node, a daemon (implemented by an OSF1 MK process) deals with the management of the jobs and activities which are present on this node.

3 Requirements for a debugger

We can summarize the Guide−2 debugging requirements as follows:

1. Debugging usual OSF/1 programs.
2. Providing information about threads involved in a computation at any time (number, status, registers, stack, thread ports, active threads).
3. Controlling the execution of individual threads and the overall Mach task.
4. Debugging Mach tasks which are already in progress. This is an important issue, since it allows debugging of tasks that are not created by the debugger.
5. Debugging shared libraries and modules which are dynamically loaded by using the OSF/1 loader.
6. Debugging class code dynamically loaded at run−time by the Guide loader.

Even if some tasks are not implemented as OSF/1 processes, since Guide uses the OSF/1 compilation tools, compiled and linked programs are in the OSF−ROSE binary format. Thus, the debugger has to deal with only one binary format whether the debuggee task is an OSF/1 process or a Mach task belonging to the Guide environment.

The current OSF/1 *gdb* allows Unix−like program to be debugged, but lacks the features needed to debug applications that directly use either Mach 3.0 features or dynamically loaded code. However, OSF/1 *gdb* copes with OSF/1 shared libraries.

Our goal is to extend the current OSF/1 *gdb* to support the Mach 3.0 features and to allow the debugging of dynamically loaded code either by means of the OSF/1 loader or by using the Guide specific loader.

Previous extension, done on at CMU by the BSD server project team, allows *gdb* to debug applications using Mach features (in the rest of this paper, we call it CMU *gdb*). We chose not to extend the CMU *gdb* because of the differences which exists between BSD and OSF/1 (binary format, shared library, etc.). However, the way the CMU *gdb* uses the Mach interface and the semantics it offers in debugging Mach tasks have greatly influenced our work in designing the extensions of OSF/1 *gdb*.

Because of the modularity it provides, we adopted a layered approach to extending the OSF/1 debugger. In a first layer we added a Mach 3.0 debugging capability. A second layer was built to support the debugging of code dynamically loaded either by the OSF/1 loader or the specific Guide−2 loader.

The implementaion of these layers is discussed in the following sections. We start, by describing the new architecture of the debugging interface.

4 New debugging interfaces

Although, programming a debugger using the Mach 3.0 interface requires more implementation effort than using the Unix *ptrace* interface, it allows a finer control over the debuggee process, in terms of threads, exceptions, etc.

We adopted, for our extension of the OSF/1 *gdb,* the user interface supplied by the CMU *gdb* and we reused as much of its code as possible. However, we split the debugging interface into two parts: a generic interface and an OS related interface [fig 2]. We had two reasons for choosing such an architecture:

- To provide Mach 3.0 with a set of generic powerful debugging primitives which are used by the OS related interface in order to express the specific OS debugging semantics.
- To use directly the generic interface to debug pure Mach 3.0 tasks, assuming that they do not belong to any OS environment.

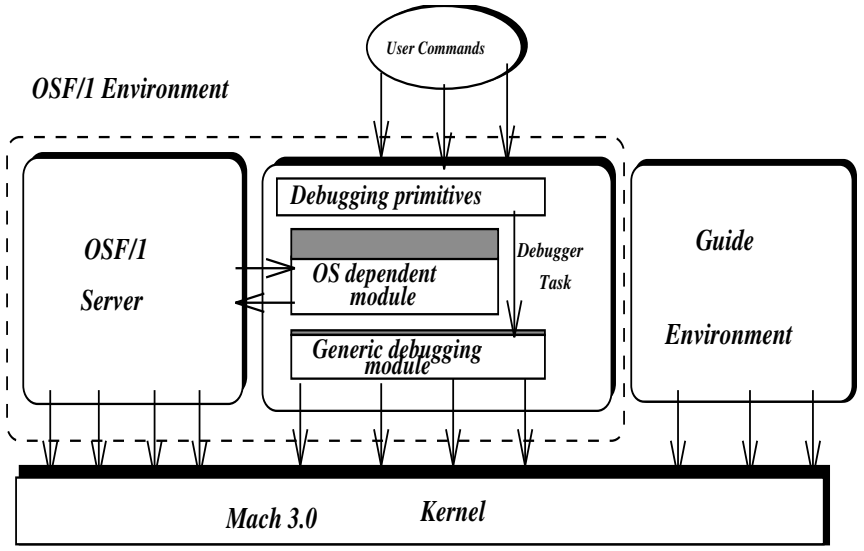

*fig.*2 Architecture of the debugger

4.1 Generic Mach 3.0 debugging module

The module exporting the generic interface insulates low level, machine dependent debugging primitives. It is directly based on the Mach 3.0 interface. We describe here these primitives and how they are implemented using Mach 3.0 kernel primitives together with the Mach 3.0 run−time.

4.1.1 Primitives for controlling task execution

We need a number of primitives in order to control the execution of a task, either thread by thread or to act on a task as a whole. We name the threads by their number (the first thread has number 0) and the primitives translate that number into the thread port.

The exported primitives for controlling the task execution in this module are :

- `Suspend` and `Resume,` used to suspend or resume respectively either the whole task or individual threads.
- `get_register` and `set_register` are primitives used to read or alter a machine register of some thread. These primitives translate the names of the registers into their corresponding register numbers depending on the machine architecture.
- `get_thread_list` gives a list of a task threads in an order that can be used to name threads in subsequent calls.
- `set_trace_flag` and `clear_trace_flag` sets and clears respectively the trace flag of the specified thread in a way that depends on the underlying hardware architecture. These calls are used when a stepping execution (instruction by instruction) is desired.

4.1.2 Accessing and changing the address space fo the debuggee

Accessing and changing the address space of the task being debugged are functionalities used for many purposes such as reading program variable values or inserting breakpoint instructions. The generic interface provides two primitives which act on the address space of the debuggee task. The `read_memory()` and `write_memory()` functions take as input the address and size of the read or write region. They recompute the addresses to be on page boundaries and they call `vm_read` and `vm_write` Mach primitives to act on the debuggee task.

4.1.3 Dealing with exceptions

First, the debugger needs to change the exception port of the debuggee task to get a port on which it holds receive rights. So, it creates a new port, and uses the `set_exception_port` primitive exported by the generic interface to replace the exception port of the debuggee task. The exported `Wait` call is used to listen on the exception port until either a message or a timeout occurs. `Wait` may also return when it receives a notification on the debugger notification port.

4.1.4 Attaching and detaching tasks

The `attach_task` and `detach_task` functions are used for debugging tasks that are not created by the debugger. As we said, this can be helpful to debug Guide tasks that are not OSF/1 processes. The `attach_task` primitive caller should supply either the send rights on the task to attach or, if the debugger is running as a privileged task, the *mid* of that task. The `attach_task` primitive suspends the target task and changes its exception port by calling the `set_exception_port` primitive. After being attached, all debugging operations can be applied upon the debuggee task as if it was a task created by the debugger. When detaching, the old exception port is restored and the task is resumed.

4.1.5 Naming

The generic interface provides a function called `right_by_mid` which gives the debugger send rights on the kernel port of the debuggee task. Thereby, the debugger can attach tasks, by specifying their *mid* if it is running as a privileged task. This function uses the *Machid* server to obtain the required rights. This is why the debugger must run as a privileged task in order to debug tasks not belonging to the OSF/1 environment, which is the case of some Guide tasks. Of course it is undesirable, to have a debugger as a privileged task. For this reason, the Guide job manager will be enhanced to manage the exchange of rights on tasks ports between Guide contexts. Thus, *gdb* will be able to obtain rights on a task port using the Guide job manager without being privileged.

4.2 OS dependent interface

The module exporting this interface contains primitives that express the debugging semantics of OSF/1 in our case. They are based on the Mach 3.0 generic debugging interface and translate directly the commands issued by the debugger at user level into primitives of this higher interface. These primitives are tied to OSF/1 concepts and constitute the main primitives of the debugger.

We did not use the existing Unix *ptrace* interface because it has many shortcomings compared to our requirements. For example, we cannot use it for attaching to processes that are already running. We replaced it by Mach equivalent primitives. The user of this interface will not see the Mach 3.0 abstractions. However, the upper layers of the debugger can access directly the Mach 3.0 generic debugging interface to manipulate Mach 3.0 abstractions rather than the OS related ones. This is how, in our case, *gdb* is able to debug normal OSF/1 processes using the OS−related interface, and Mach 3.0 tasks not known by the OSF/1 server, by using the generic Mach 3.0 interface. The different primitives of the OSF/1 interface are described below.

4.2.1 Setting and removing breakpoints

Setting breakpoints allows the user to stop the execution of the program at certain points specified by an address value. This module exports the `set_breakpoint` primitive which takes as argument the address where the program should stop. Likewise, The `remove_breakpoint` primitive removes a previously set breakpoint.

4.2.2 Starting and resuming execution

The `start_exec` primitive is used to start the execution of the debuggee after inserting breakpoints. The `resume_exec` and the `step_exec` primitives are used for resuming a suspended debuggee in a normal or a stepped way.

4.2.3 Handling signals

The OS dependent interface allows the upper layers to control how OSF/1 signals are dealt with by the debugger. Since the OS dependent module receives all the exceptions sent

to the debuggee task, they can be translated into OSF/1 signals. Some of these exceptions are due to the debugging activity, such as those generated by breakpoints or stepping. The others are ordinary signals that can be ignored or passed to the original handler of exceptions (e.g the OSF/1 server) depending of what the user of the debugger has specified.

However, some signals are not issued from exceptions such as software signals. For instance when the user kills a process, it is just an indication to the Unix server to kill the given process and Mach is not aware of it. To deal with this kind of signal, the debugger uses Unix primitives, since signals are a pure Unix concept. A simple trick used in the OS dependent module to deal with these signals, consists of using three Cthreads. One Cthread is waiting for exceptions using the generic interface primitive `Wait()`. The other Cthread uses the Unix *wait* syscall to wait for state changes of the debuggee process. The third Cthread is waiting for a message from the other two Cthreads. If one of the first two Cthreads catches something, it sends a message containing all the information needed to the third Cthread and returns waiting for new events. The third Cthread treats the event accordingly and returns waiting for messages from the other Cthreads.To catch Unix signals via the *wait* syscall, we assume that the debuggee process is a child of the debugger. If it is not the case, the debugger has no way to catch these signals, because OSF/1 assumes that the debugger is always the parent. By simultaneously listening to the exception port and waiting for state changes signalled by the OSF/1 server, the debugger controls completely the signalling behavior of an OSF/1 process.

4.2.4 Attaching and detaching processes

The attaching primitive has as input either the *pid* of the process or the *mid* of the task to be attached. If the program to attach to is running in an OSF/1 process, the debugger calls the `task_by_pid()` system call to get send rights to the kernel port of the process, supplying its *pid*. The OSF/1 server checks the identity of the debugger which must have the same *uid* (user id) as the process it asks rights on, and then gives it send rights on the debuggee process. The debugger can then suspend this process and starts debugging. If the target to attach is a Mach task, the debugger uses its lower interface function `right_by_mid` to get the required rights. Once attached, the task looks like the one created by the debugger except that the debugger cannot use the *wait* system call for catching signals since the target task is not its child. This requires that the OSF/1 server be modified to dissociate the parent of a process from its debugger. So, when attaching a process, the user is not able to handle all the signals posted to the debuggee process; it can detect only those which cause the death of the task using Mach notification. This may cause the debugger to panic, when it encounters inexpected behavior of the debuggee process. This unexpected behaviour may be caused by such unperceived, hence uncontrollable signals. This is why it is inadvisable to attach OSF/1 processes. We hope that future releases of OSF/1 will correct this limitation.

5 Debugging Dynamically loaded code

The appearence of the shared libraries paradigm and the ability to load new modules of code during execution in Unix systems, has brought into question the suitability for current debuggers to handle applications using these features. OSF/1 offers both of them. However, OSF/1 *gdb* supports only the debugging of shared libraries. In our case, Guide uses its own loader to bring objects into the address space of the task when needed. As objects in Guide are accessed by specific operations (object methods) defined in the object class, both code and data are mapped (and paged in on demand) when an object fault occurs. To be able to debug the loaded code, we need our debugger to be informed of every change in the debuggee address space.

We initially completed the OSF/1 debugging interface, by adding the ability to debug dynamically loaded code by means of the OSF/1 loader. After, we added a specific Guide layer in the OSF/1 *gdb* to allow the debugging of objects loaded using the Guide loader. Since OSF/1 is not aware of the existence of some Guide tasks, we could not use OSF/1 features in implementing this capability. The ability to debug the Guide system is a compilation option which can be turned **on/off** when building the debugger. Thus, the same debugger can be built just for the OSF/1 environment, or with a Guide system support. We describe first, how the code loaded at run−time by the OSF/1 loader is supported.

5.1 OSF/1 loaded code

In OSF/1, the loader is located in the address space of each process. When an application is built using shared libraries, OSF/1 exports to that application the APIs(3) of the loader via the C library. The user code can ask the loader to load new modules. The loader resolves any undefined references and loads any other modules needed by the requested module, and then returns the entry points of the new modules to the caller.

In the case of dynamically loaded code, two problems arise :

1. The loader should know that it works for a process being debugged in order to inform the debugger each time a new module is loaded or discarded.
2. The debugger should obtain information about the mapped module, such as its starting address, the different regions it contains, etc.

We have seen earlier that when the state of the debuggee process changes, the debugger is informed by the OSF/1 system. It then tries to determine what has happened to the debuggee and deals with each case. To change its state, the loader (and so the whole process) is suspended by sending a SIGTRAP signal to itself. Then, the debugger is informed by the system that its child (the debuggee) is suspended. This method is used by a loader to inform the debugger when loading shared libraries. We reused this trick to inform *gdb* about new loaded modules. We modified the OSF/1 loader, so that each time a new module has been loaded by a loader and before returning control to the user code, the loader sends to

(3) Application Programming Interface

itself a SIGTRAP signal. The loader gets the information that it is being debugged from the OSF/1 system. When it is launched, it receives, with the name of the program it has to load (main program), a flag (the debugging flag) which indicates whether it is running in debugging mode or not.

Once, the debugger is informed that the debuggee state has changed, it investigates the cause of the change. The fact that the PC (program counter) is in the loader region enables *gdb* to find out that it has been called by the loader which has loaded in, or removed from memory, a module.

To obtain information about the new code loaded, *gdb* uses the cross−loading capability of its OSF/1 loader(4), which allows information to be obtained about modules loaded in another process giving some privileges. After finishing all required processing, *gdb* resumes the suspended debuggee. The user can now debug the loaded code since the debugger knows all about it.

5.2 Debugging Guide Class code

A Guide application loads objects in a context the first time they are invoked. As in OSF/1, the Guide loader is part of the Guide run−time mapped in each Guide context. However, unlike the OSF/1 loader, which can rely on the use of signals to warn the debugger, the Guide loader does not have such support since Guide contexts may not be known by the OSF/1 system. For the same reason, the Guide loader cannot rely on OSF/1 to know that a Mach task is being debugged.

We investigate another method of obtain these information. Since Guide applications are handled by Mach tasks, we resort to using Mach mechanisms.

5.2.1 Informing the loader

Since there is no implicit information to indicate whether the current process is debugged or not (see 5.1), as was the case for OSF/1, this information must be explicitly communicated to the Guide loader.

Some Guide contexts are not OSF/1 processes, so the debugger cannot create them by using the *fork* system call. The only way for the debugger to debug them is to attach to them during execution. This is not a big problem, since in Guide jobs always start in an OSF/1 process that can be created by the debugger. So, when a user wants to debug a non OSF/1 task, it inserts a breakpoint just after the piece of code that creates the task in the OSF/1 process . At run time, when the breakpoint is reached, the OSF/1 process is detached from *gdb* and the the newly created Mach 3.0 task is attached to *gdb*. Hence *gdb* gains control over the new Guide context before the execution of any instruction.

---

(4) Gdb can use the loader interface as any OSF/1 process.

When attaching to a Guide context, the debugger searches for a special symbol in the symbol table located in the executable file. This symbol plays the role of the "debugging flag". The debugger gets the flag address in the core image of the task. Before resuming the execution of the already attached task, it sets this flag with an authentication key. Initially, the flag has a Null value which means that the current task is not being debugged.

Every time the loader loads new objects, it checks this flag. If it is set, the loader knows that the debuggee is being debugged and should inform the debugger about the changes in the debuggee address space. The debugger will in turn, check the key value to authenticate the task as the debuggee one.

5.2.2 Acquiring the identity of the debugger

Even if the loader knows that the current task is being debugged by reading the "debugging flag", it doesn't know the identity of the debugger. In OSF/1, the loader doesn't care about this, because the system knows which is the debugger of each debuggee process. Our extension used the Mach naming scheme to identify the debugger. When a debugger starts, it creates a Mach port and checks it into the Mach 3.0 name server with a well known name. In this way, the loader of a currently debugged Guide−2 context gets send rights on the debugger port from the Name server. Then, it can inform the debugger that new objects have been loaded or discarded from the context address space. To avoid name conflicts between debuggers working on the same machine, the name registered in the name server contains the *mid* of the debuggee task. The debugger uses the *Machid* server to obtain the debugge task *mid*.

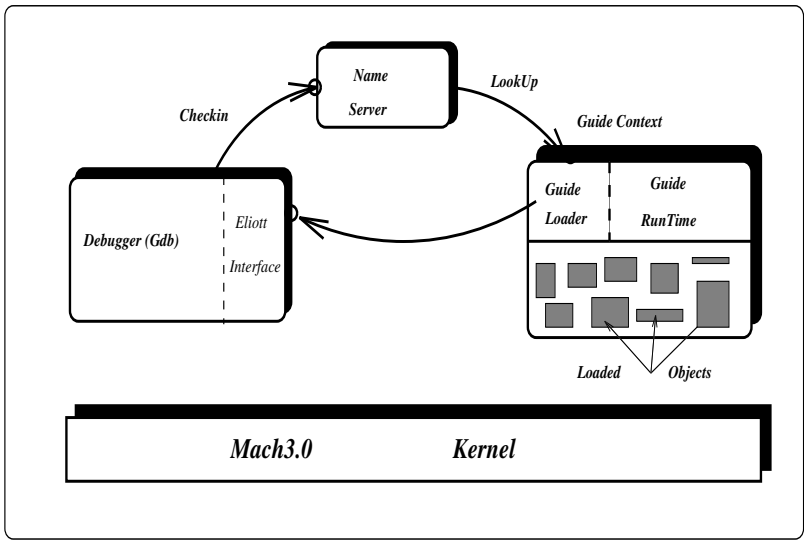

*fig.*3 Debugging Guide class code

5.2.3 Acquiring informations about the loader activity

When the loader has informed *gdb* of the modifications occurred in the Guide context, the debugger gets the information related to the mapping from the loader. This proceeds in

the folowing way. The loader sends a message to *gdb* with the debugging flag value as a data message, and waits for a response (Mach RPC). Then *gdb* authenticates the debuggee task and, if the authentication succeeds, it suspends the task. After that, *gdb* searches in the symbol table for a symbol known as "mapping table" in order to obtain its address in the task context. This address gives access to the data structure containing all the information the loader maintains about the objects present in the debuggee address space; comparing this data to the structure it holds, *gdb* can deduce what objects have been added or removed.

6 Conclusions

This paper has described the customization of the OSF/1 *gdb* debugger to support:

- Debugging of OSF/1 MK processes together with Mach 3.0 tasks and threads.
- Debugging code dynamically loaded by either the OSF/1 loader or a specific application loader.

To build the new *gdb* interface, we have defined a layered architecture based on the Mach 3.0 interface. The lower layer provides a generic interface (i.e. a BSD or an OSF/1 debugger can be built on top of it), which provides Mach 3.0 with a powerful debugging interface. The second layer provides an interface tied to OSF/1 debugging semantics. This new OSF/1 *gdb* is able to debug either OSF/1 processes or Mach tasks belonging to other environments. Furthermore, this new debugger debugs dynamically loaded code.

This debugger was developed in the framework of the Guide project, and is currently being used by our team. Guide−2 provides its own loader, but uses the OSF/1 loader as well. The Guide−2 loader cooperates explicitly with *gdb;* it provides *gdb* with the required information.

Even though this work was mainly motivated by providing Guide developers with a suitable tool for debugging Guide−2 system, the resultant debugger can be used for debugging any software developed under OSF/1 which uses the Mach 3.0 interface directly. This encourages the research community to use the OSF/1 MK architecture, since it allows the use of either the OSF/1 interface or the microkernel services leading to greater freedom in design.

We have also seen that some features cannot be implemented without cooperation with the host OS. The problem of signals sent to attached processes, described in sections 4.2.4 and 5.1, is a good example. This necessitates a deep study of the relationship between the OS servers and the debuggers running on microkernels [7] which is not well understood yet.

Acknowledgments

We would like to aknowledge all the helpful feedback we have received especially from James Loveluck, Sacha Krakowiak, Serge Lacourte and Éamonn McManus. This work was partly supported by the commission of European Communities under the Comandos ESPRIT project (no 2071).

Bibliography

[1] M.J. Bach, *The design of the Unix operating system*, Prentice–Hall, Englewood cliffs, New Jersey, 1986.

[2] R. Balter, J. Bernadat, D. Decouchant, A. Duda, A. Freyssinet, S. Krakowiak, M. Meysembourg, P. Le Dot, H. Nguyen Van, E. Paire, C. Roisin, M. Riveill, X. Rousset de Pina, R. Scioville, G. Vandôme, Architecture and implemetation of Guide, an object–oriented distributed system, *Computing Systems*, vol 4 (1), pp. 31–68, 1991.

[3] F. Barbou Des Places, P. Bernadat, M. Condict, S. Empereur, J. Febvre, D. George, J. Loveluck, É. McManus, S. Patience, J. Rogado, P. Roudaud, Architecture and Benefits of a Multithreaded OSF/1 Server, February 1992.

[4] F. Boyer, J. Cayuela, P. Y. Chevalier, D. Hagimont, A.Freyssinet, Supporting an object–oriented distributed system: experience with Unix, Mach and Chorus, *Proc. Symposium on Experience with Distributed and Multiprocessor Systems*, Atlanta, Mar. 1991.

[5] A.Freyssinet, S.Krakowiak, S.Lacourte, A generic object–oriented virtual machine, *Proc. 3rd Int. Workshop on object–Orientation in Operating Systems*, pp. 73–77, Palo Alto, Oct. 1991.

[6] D.Hagimont, S. Krakowiak, X. Rousset de Pina, Protection in an Object–Oriented Distributed System, *Proc. of the 4th Int. Workshop on Object Orientation in Operating System*, Dourdon pp. 273–279, Sep. 1992.

[7] Rand A. Hoven, Mach interfaces to support Guest OS Debugging, *Usenix Conference Proceedings*, pp. 131–147, Monterey, November 1991.

[8] S. Krakowiak, M. Meysembourg, H. Nguyen Van, M. Riveill, C. Roisin, X. Rousset de Pina, Design and implementation of an object–oriented, strongly typed language for distributed applications, *Journal of Object–Oriented Programming*, 3 , 3 (Sep–Oct 90), pp. 11, 21.

[9] H. Nguyen Van, M. Riveill, C. Roisin, *Manuel du langage Guide*, Bull–IMAG, 2, rue de Vignate 38610 Gières France , Dec 1990.

[10] R. Stallman, R. Pesch, *Using GDB* , Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139, USA , July 1991.

Debugging an object−oriented system using the Mach interface

1 Introduction . . . 1

2 The Guide architecture on top of OSF/1 MK and Mach 3.0 . . . 2

2.1 The Guide−2 execution model . . . 2

2.2 Implementation . . . 3

3 Requirements for a debugger . . . 4

4 New debugging interfaces . . . 5

4.1 Generic Mach 3.0 debugging module . . . 5

4.1.1 Primitives for controlling task execution . . . 5

4.1.2 Accessing and changing the address space fo the debuggee . . . 6

4.1.3 Dealing with exceptions . . . 6

4.1.4 Attaching and detaching tasks . . . 6

4.1.5 Naming . . . 7

4.2 OS dependent interface . . . 7

4.2.1 Setting and removing breakpoints . . . 7

4.2.2 Starting and resuming execution . . . 7

4.2.3 Handling signals . . . 7

4.2.4 Attaching and detaching processes . . . 8

5 Debugging Dynamically loaded code . . . 9

5.1 OSF/1 loaded code . . . 9

5.2 Debugging Guide Class code . . . 10

5.2.1 Informing the loader . . . 10

5.2.2 Acquiring the identity of the debugger . . . 11

5.2.3 Acquiring informations about the loader activity . . . 11

6 Conclusions . . . 12