Language and Runtime Support for Dynamic Interposition of System Code

Przemysław Pardyak Stefan Savage
Brian N. Bershad

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

November 3, 1995

Abstract

Extensible operating systems require an efficient means to dynamically bind extensions to existing code. The *SPIN* operating system provides this functionality via an event-based invocation mechanism. Events in *SPIN* are well-known typed procedure names that are used by extensions to decouple the provider of a service from potential clients. Extensions communicate with a centralized *dispatcher* service to register their interest in raising or handling such events. To implement this binding, the dispatcher relies on the Modula-3 language and run-time in providing safety and high performance of event operations and it employs run-time code generation techniques to create highly optimized communications paths between raisers and handlers.

1 Introduction

An extensible operating system allows application programs to customize operating system services, thereby improving performance, correctness or simplicity. The *SPIN* operating system supports extensibility by allowing application code to interpose on kernel interfaces [BSP$^+$95]. These *system extensions* are directly linked into the kernel address space, and their memory safety is guaranteed by writing them in a type-safe programming language (Modula-3) [Nel91]. Interposition is implemented using an event-based invocation mechanism that centralizes all communication through the *SPIN dispatcher* service. The dispatcher allows extensions to dynamically change the body of code executed in response to a service request. Because the dispatcher is involved in all communication, it is able to implement access control and provide a consistent semantics of interposition on that communication. The dispatcher is highly integrated with the Modula-3 runtime, which we found essential for providing safety, high performance, flexibility, and a simple representation for events within the Modula-3 language.

Previous work has explored safe interposition [Jon93, Pat93, B$^+$92, VGA94], but the high cost of protecting system code from extension code has generally limited its utility to coarse grain system interfaces, and has rarely been competitive with monolithic implementations. The dynamic linking of kernel objects, as found in modern versions of Unix, provides low overhead communication, but without support for safety. Neither approach is well suited to an environment in which untrusted system extensions are composed at a fine-grain to provide new services.

2 Motivation

Extensible systems require a dynamic binding mechanism to allow the system to be changed in response to changing service demands. To be effective, this mechanism must be efficient, simple to use, and generally applicable. To be used extensively, the mechanism must also be flexible enough to allow the run-time composition and manipulation of independently written extensions.

Common extension schemas, such as stacking [HP94] and filtering [MRA87, YBMM94] can only be decomposed if the binding can be transparently routed to multiple extensions. For example, different extensions may implement different network protocols, but they all must be called from the same network device driver, therefore the decision to execute a particular extension must be a dynamic one, based on the contents of a network packet.

Dynamic linking [OBLM93, BC94] and method lookup are two mechanisms that facilitate late binding. The first enables code to be installed into a running system, but tightly couples clients to a particular service implementation. By itself, dynamic linking does not provide for the transparent routing of requests to alternate or supplementary services. Method lookup, such as Mach's transparent system call emulation facility [Jon93], allows for transparent service interposition, but does not allow multiple modules to service the same request. Using either dynamic linking or method lookup, multiple independent extensions may not provide different parts of a service without explictly cooperating.

An alternative programming methodology that decouples service names from service implementations is based on events, which enables transparent control routing and multicast. In an event based system, one or more modules register interest in handling a particular named event, and one or more modules may raise the event. An underlying event dispatcher routes control (and possibly data) from the raiser to one or more handlers. Because an event simply signals that something has happened and does not specify a direct linkage between caller and callee, multiple extensions are free to interpret and act on the event independently.

Unfortunately, traditional event systems have been less efficient than linked procedure calls, and have usually required an invocation syntax and name space distinct from the native procedure call mechanism. *SPIN* provides the flexibility of an event-based system with the performance and language integration of pure dynamic linking by integrating events into the language syntax, and heavily relying on runtime code generation,

3 Integrating extensions

SPIN and its extensions are written in Modula-3, a modular, type-safe, ALGOL-like language. A Modula-3 program consists of *modules* that contain code and data, and *interfaces*, that control their visibility. Programs in Modula-3 are written as a composition of modules communicating by procedure calls through interfaces. Fundamentally, a procedure call is a signal from one module to another to indicate or request a change in the system state. In effect, a procedure call is an event *raised* by the caller and *handled* by the procedure's implementation. We exploit this relationship to define events in *SPIN*.

3.1 Events

In *SPIN*, an extension is simply a Modula-3 program that executes in the kernel. It requests services by raising events, and provides services by handling events. For example, an extension may service page faults by handling the virtual memory system's *PageFault* event, which is raised whenever a page fault occurs.

Events are defined in terms of Modula-3 procedure signatures, which preserve the natural invocation syntax and "feel" of the language. Consequently, an event is a typed and named message that may carry arguments and return a value. Every procedure is implicitly an event and hence eligible for extension. Events declared in an interface are available for modules that import that interface to either raise the event, or to provide additional handlers for the event. Events not declared in an interface can be explicitly passed between modules as procedure variables.

3.2 Event handlers

An event handler may be called when its associated event is raised. Any arguments specified when an event is raised are provided to the handler. Each event handler may be associated with a set of *guard* predicates, which are used to filter out unwanted event raises. For example, an extension may only be interested in handling page fault events for its data segment, and can define a guard that checks whether the particular virtual address where the fault occurred is in that segment. A handler is executed only if all of its guards evaluate to *TRUE*.

We allow the predicate to be specified outside of the handler for two reasons. First, predicates common to multiple handlers on the same event can be evaluated only once. Second, it allows a service to impose an arbitrary predicate on a handler which can be used to restrict when that handler may execute. For example, the virtual memory system may impose a guard that restricts an extension to handling page fault events only within its own address space. A predicate applied for the purpose of limiting access to an event is called an *imposed* guard.

3.3 The Dispatcher

A centralized dispatcher implements the event machinery, which includes binding events to handlers, evaluating guards, and invoking handlers. Here is the signature of the primary function exported by the dispatcher interface:

```
PROCEDURE InstallHandler(event: PROCANY;
                         handler: PROCANY;
                         guard: PROCANY);
```

The first argument is the name of the event to be handled, expressed as a procedure (a `PROCANY` is a general type representing the class of all procedures [?]). The second argument, `handler`, is the specific handler procedure that should run when the named event is raised. Lastly, the third argument is the guard predicate that must evaluate to `TRUE` at the time the event is raised in order for the handler to run. At installation time, the dispatcher ensures that the type signature of the event, the handler, and the guard are compatible. It is a checked error if they are not.

3.4 An example

Figure 1 demonstrates the use of the dispatcher interface by a TCP module. The module installs a handler and guard on IP's "PacketArrived" event. When an IP packet arrives, the IP module will raise that event. The dispatcher will fire TCP's guard predicate, which will check if the incoming packet is indeed destined for TCP. If it is, then the dispatcher will then fire the handler, which will do the actual processing for the packet.

3.5 Protection

As mentioned, events provide a name space for communication among extensions. There are three aspects to protecting events:

```
INTERFACE IP;
(* Event raised on each incoming
   IP Packet *)
PROCEDURE PacketArrived(
                packet: Net.T;
                payload: T);

(* Return the contents of the
   packet's IP protocol field *)
PROCEDURE GetProtocolField(
                 packet: Net.T) :
                        Byte;

END IP.
```

```
(* A module implementing the
   TCP protocol *)
MODULE TCP;
IMPORT IP;

CONST TCPProtocol = 6;

(* A guard procedure *)
PROCEDURE IsTCPPacket(packet: Net.T;
                      payload: T): BOOLEAN =
  BEGIN
    RETURN IP.GetProtocolField(packet)
                = TCPProtocol;
  END IsTCPPacket;

(* A handler procedure *)
PROCEDURE Input(packet: Net.T;
                payload: T) =
  BEGIN
    (* Process incoming packet as
       a TCP packet *)
  END Input;

(* Initialize the TCP module *)
PROCEDURE Init() =
  BEGIN
    Dispatcher.InstallHandler(
        IP.PacketArrived,
        Input,
        IsTCPPacket);
  END Init;
...
END TCP.
```

Figure 1: The TCP module installing itself as a handler on the IP module's "PacketArrived" event.

- *Static access control.* How to control which modules are allowed to raise an event and install a handler on an event?
- *Dynamic access control.* How to control which handlers are allowed to handle a particular instance of an event raise?
- *Authorization.* How to control which module can control static and dynamic acccess control?

We rely on a safe dynamic linker [SFPB96] for static access control of events. This is an issue of event name visibility. An event name is simply a linker symbol. If the extension is not linked against the protected context in which the symbol is defined, then the event is not visible.

The right to handle an event may not imply the right to handle all occurrences of that event. Dynamic access control ensures that only legitimate handlers receive an instance of an event raise. As previously mentioned, imposed guards enable an event occurence to be dynamically checked before it is delivered to a handler.

The module that implements the procedure that defines an event is responsible for managing access control for that event. Specifically, if some module `M` implements a procedure `M.P`, then `M` it is authorized to manage access control to the event `M.P`. When a client requests that the dispatcher install a handler on an event, the dispatcher invokes the event's authorizer procedure. The authorizer procedure can return TRUE which allows full accesss to the event, FALSE which denies the installation, or may impose a guard on the specified handler. The imposed guard can be used to dynamically control access to the event each time it is raised.

4 Overhead of the dispatcher

The performance of a system's interposition mechanism determines the granularity with which it can be used. Messages and IPC, for example, are suitable for infrequently called extensions that execute at the level of processes, but cannot be used for extending fine-grained and frequently used services such as TLB miss handlers.

The interposition mechanism described in this paper enables extensibility at the level of a procedure call. The overhead of event dispatching is comparable to the cost of a procedure call through the use of run-time code generation. Dispatching an event requires iterating over all handlers installed on that event. For each handler, the dispatcher evaluates any guards and calls the handler if they all return `TRUE`.

We use run-time code generation to generate a specialized and optimized version of the dispatch routine. The code is specialized for the number of arguments to the event. The dispatch loop is unrolled to transform handler invocations from indirect procedure calls into the dispatcher to direct procedure calls through to the guards and handlers. We also use inlining to embed the code of small guards and handlers directly into the dispatch routine. Finally, we use peephole optimizations to improve the quality of the generated code.

Table 1 shows the performance of our dispatcher under a variety of conditions. We show the performance of a dispatched event to between 1 and 10 handlers, with and without installed guards, and varying numbers of arguments. Guards always return TRUE, and handlers return without performing any work. In the simple case, where an event has only one potential handler with no guard, performance is identical to straight procedure call from the raiser to the handler. Indeed, the implementation in this case is straight procedure call. Only when additional guards or handlers are installed does the dispatcher introduce new code beyond that required to perform a single procedure call. Overhead grows linearly with the number of guards and handlers, with each incurring an overhead of roughly one procedure call.

number of arguments	Modula-3 procedure call	dispatched events					
		1 handler		5 handlers		10 handlers	
		w/o guard	w/ guard	w/o guard	w/ guard	w/o guard	w/ guard
0	0.12	0.12	0.37	0.76	1.22	1.36	2.26
4	0.18	0.18	0.49	0.85	1.46	1.45	2.65

Table 1: The overhead of event dispatching in μsecs. We use runtime code generation to implement a fast path to invoke guards and handlers from the dispatcher. Events are dispatched to 1, 5 and 10 handlers with and without guards. The first column shows the overhad to directly invoke a Modula-3 procedure call.

We find that our runtime code generation and optimizations improve the performance of event dispatching by roughly a factor of four when compared to iterating over a table of function pointers. Even with optimizations, though, dispatch overhead may be significant if many handlers are installed, but few actually execute when an event is raised. This overhead comes from evaluating guards that return `FALSE`.

We have planned a set of opimizations that will transform the set of guards for an event into a decision tree which will serve as an optimized dispatch routine. By eliminating common subexpressions and transforming similar comparisions into hash table lookups we expect to achieve dispatch overhead which is linear with the number of executed rather than potential handlers. Furthermore, we intend to take advantage of on-going research into automatic runtime code generation [CEA+96] to automate our optimizations.

5 Conclusions

SPIN's event handling services allow extensions to interact with systems services in a decoupled manner. Our event interface relies on the language to define events in terms of procedures, the language's compiler and runtime to ensure that events and handlers are properly typed, and runtime code generation to ensure that events can be dispatched quickly. The result is a mechanism that enables low-overhead, decoupled, flexible communication within a large, dynamically extensible operating system.

References

[B+92] D. L. Black et al. Microkernel Operating System Architecture and Mach. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 11–30, Seattle, WA, April 1992.

[BC94] Arindam Banerji and David L. Cohn. Protected Shared Libraries. Technical Report 37, University of Notre Dame, 1994.

[BSP+95] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin G n Sirer, Marc Fiuczynski, David Becker, Susan Eggers, and Craig Chambers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, Copper Mountain, CO, December 1995.

[CEA+96] C. Chambers, S. Eggers, J. Auslander, M. Philipose, M. Mock, and P. Pardyak. Automatic dynamic compilation support for event dispatching in extensible systems. Submitted to the First Workshop on Compiler Support for Systems Software, November 1996.

[HP94] J.S. Heidemann and G.J. Popek. File-System Development with Stackable Layers. *Communications of the ACM*, 12(1):58–89, February 1994.

[Jon93] Michael B. Jones. Interposition Agents: Transparently Interposing User Code at the System Call. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 80–93, Asheville, NC, December 1993.

[MRA87] J. Mogul, R. Rashid, and M. Accetta. The Packet Filter: An Efficient Mechanism for User-level Network Code. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 39–51, Austin, TX, November 1987.

[Nel91] Greg Nelson, editor. *System Programming in Modula-3*. Prentice Hall, 1991.

[OBLM93] Doug Orr, John Bonn, Jay Lepreau, and Robert Mecklenburg. Fast and Flexible Shared Libraries. In *Proceedings of the 1993 Winter USENIX Conference*, June 1993.

[Pat93] Simon Patience. Redirecting Systems Calls in Mach 3.0, An Alternative to the Emulator. In *Proceedings of the Third USENIX Mach Symposium*, pages 57–73, Santa Fe, NM, April 1993.

[SFPB96] E.G. Sirer, M. Fiuczynski, P. Pardyak, and B.N. Bershad. Safe Dynamic Linking in an Extensible Operating System. Submitted to the First Workshop on Compiler Support for Systems Software, November 1996.

[VGA94] A. Vahdat, P. Ghormley, and T.E. Anderson. Efficient, Portable and Robust Extension of Operating System Functionality. Technical Report UCB CS-94-842, University of California, Berkeley, December 1994.

[YBMM94] Masanobu Yuhara, Brian N. Bershad, Chris Maeda, and J. Eliot B. Moss. Efficient Packet Demultiplexing for Multiple Endpoints and Large Messages. In *Proceedings of the 1994 Winter USENIX Conference*, pages 153–165, San Francisco, CA, January 1994.