Implementing Self-Managing Protection Domains in Charm

Alan Dearle and David Hulse

School of Computer Science, University of St Andrews, North Haugh,
St Andrews KY16 9SS, Scotland
email: {al,dave}@dcs.st-and.ac.uk

Abstract

We believe that the engineering of mobile or persistent processes is hindered in many systems by the amount of coupling between user-level and the kernel. This coupling usually takes the form of user level data structures containing opaque references to kernel data structures. In this paper we show how self-managing protection domains may be constructed that support a modern object-oriented application environment that is not closely coupled with the kernel. This is achieved through the introduction of a software artefact called the Umbrella that abstracts over the low-level aspects of self-management and present a clean object-oriented management interface to the upper layers of software. We show how the Umbrella is engineered and some of the problems that must be overcome in order to achieve isolation from the kernel. We conclude by showing how a thread system may be constructed above the Umbrella.

1 Motivation and Approach Taken

We are currently developing a distributed architecture that permits computation to occur safely in arbitrary physical locations. We aim to achieve this through the introduction of security domains hosted on inexpensive network nodes that can be placed within the existing network. We call these machines *Thin Servers*. Thin Servers permit the flexible dynamic placement of code, data and processes by authorised users, in a secure and simple manner. Thin Servers support a model of global computation in which persistent objects have global identity, and the physical domain in which code is executed may be defined by the programmer.

The construction of a distributed architecture such as the one described above requires many research issues to be addressed. However, in this paper we focus on the process support provided by the operating system supporting Thin Servers. The design described here was driven by a number of considerations:

1. The desire to enforce protection between different computations running on a Thin Server.
2. The desire to minimise coupling between user-level and the kernel to make mobile and persistent processes easier to engineer.
3. The desire to eliminate the replication of meta-data between user-level and the kernel.
4. To eliminate policy from the kernel to permit abstractions suitable to the application domain to be constructed efficiently.

These considerations are described in more detail below. Of paramount importance to the Thin Server architecture was the desire to enforce protection domains between different processes running on any particular Thin Server. Consequently, the kernel must provide some mechanism for constructing and identifying a protection domain.

Engineering mobile or persistent processes is hindered in many systems by the amount of coupling between user-level and the kernel. This coupling usually takes the form of user level data structures containing opaque references to kernel data structures. For example, the (user level) open file descriptor table indexes into kernel data structures. Similarly, information about the location of the stack(s) and the heap(s) needs to be both *accessible* (which is sometimes the case) and *definable* (which is less common) from user level. We seek to eliminate this problem by removing all unnecessary coupling and making all kernel data structures (other than those needed to enforce security) directly accessible from user level. This is discussed further in [3], [1] and [7].

The duplication of information between user level and the kernel is a source of inefficiency in many persistent, database and middleware systems and has been reported elsewhere [8] [2]. The most notable example of this involves the virtual address translation tables that are often wholly or partially replicated in user level memory

managers and the kernel. We seek to eliminate the replication of such data structures and wherever possible give the user-level control of them to reduce the kernel's control of policy. Consequently, we seek to minimise the amount of meta-data in the kernel. This has an added benefit of further reducing coupling.

The desire to keep all policy and management out of the operating system results in the concept of a self-managing protection domain. We embrace the ideas set out in the exo-kernel [6] and believe that a self-managing protection domain is all that is required as a building block. Consequently, in the operating system supporting Thin Servers, *Charm* [4, 5] the self-managing protection domain is the only building block supplied to the application programmer. Common abstractions such as processes and threads may be efficiently implemented at user level using this building block.

1.1 Protection Domains

In order to support self-managing protection domains, the kernel needs to provide some mechanisms with which self-management may be implemented. These include communication mechanisms and mechanisms for providing secure access to the hardware resources supported by the machine. Communication mechanisms are required so that:

1. self-managing protection domains may communicate with the kernel,
2. the kernel may communicate with self-managing protection domains, and,
3. self-managing protection domains may communicate with each other.

Secure access is required to all resources provided by the hardware and is the principal reason for the kernel's existence. The hardware resources multiplexed by the kernel must include: processor time, physical page frames, access to interrupts including device interrupts, clock ticks and protection faults. Depending on the type of hardware, it may also include control over virtual memory. For example, this is necessary on the Pentium implementation of Charm due to the hardware's intimate knowledge of page table layout.

The basic building block provided by the Charm kernel is a self-managing protection domain called an *arena*. An arena is an ephemeral protection domain that has an associated fixed set of *upcall points*. The upcall points are used to communicate events from the kernel to the arenas. Each upcall point specifies the address of a piece of code that is activated when an event occurs. If the arena has control of the CPU when the event occurs, the operating system transfers control to the upcall point in much the same way as hardware interrupts are delivered to the operating system. Consequently, the upcall handler is responsible for the preservation of registers, except for one or two scratch registers whose contents are saved in a globally visible kernel data structure. As we will show in this paper, this permits arbitrary concurrency mechanisms, such as thread models, to be safely and efficiently constructed at user level.

The arena upcall points are managed using system calls all of which are non-blocking in Charm. The set of upcall points currently supported by the implementation includes:

1. *Activate*, which specifies the piece of code to be called when the arena is instantiated.
2. *Time Slice Start* (TSS), which is called whenever the arena subsequently gets control of the CPU.
3. *Time Slice End* (TSE), which is called when the current processor allocation ends.
4. *Tick*, which is used to accept hardware generated clock events.
5. *Exception*, which is called synchronously in response to actions performed by the program running within the arena.
6. *Interrupt* (IRQ), which is called when asynchronous interrupts occur.
7. *Inter-Arena Call* (IAC), which is used for inter-arena communication.

2 Self-Management of Protection Domains

The self-management of protection domains (arenas) is achieved through the implementation of appropriate upcall handlers. Rather than the kernel passing information to an arena as parameters to an upcall, such information is communicated implicitly via a public data area that is visible to all arenas. This data area is maintained in read-only memory by the Charm kernel and is used to publish all of the public kernel meta-data. Part of this data includes an array of structures, one per arena, containing information pertinent to the self-management of that arena. During initialisation, an arena may locate this meta-data using the *get_kernel_data* system call, which returns a pointer to the start of the public data area.

To facilitate the development of self-managing protection domains in Charm, a user-level class library was constructed to abstract over the low-level aspects of self-management and present a clean object-oriented management interface to the upper layers of software. This class library is organised around the concept of the *Umbrella*. The philosophy of the Umbrella is to present an object-oriented framework for self-management to the code hosted by the *arena*. The nature of this framework is described below.

2.1 Upcall Management

At its lowest levels, the Umbrella is responsible for intercepting the seven possible upcall events. To do this, it provides a set of upcall handlers that manage all of the fine details including preserving registers and coping with atomicity problems. The upcall handlers convert each upcall into a software event that can be *hooked* by the layers of software running above the Umbrella. However, the *TSS* and *TSE* upcalls are handled transparently by the Umbrella to create the illusion of a virtual CPU owned exclusively by the arena. That is, the Umbrella abstracts over the context switching between arenas. Since the Umbrella consumes TSS/TSE events, the upper layers are effectively presented with a continuous, if not regular, stream of *tick* events.

2.2 Event Notification

The Umbrella permits the upper layers to *hook* the *Activate*, *Tick*, *IRQ*, *Exception*, and *IAC* events. From the point of view of a program running above the Umbrella, these events are like interrupts that suspend the running program while they are handled. To facilitate the handling of these events, the Umbrella provides an object-oriented interface that interacts with objects supplied by the upper layers. The architecture of the Umbrella interface is shown in Figure 1.

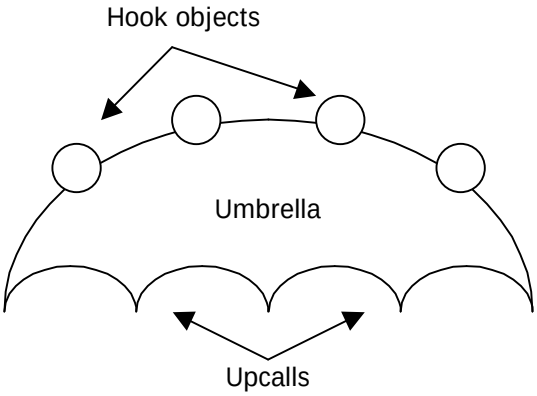

Figure 1: The Umbrella interface

The upper layers of software can hook an event by registering an event-handler object with the Umbrella. Such objects are required to support a particular interface defined by the Umbrella. However, only one such object can be registered against a given event hook. In practice, this restriction is not limiting since it is possible to register broker objects that manage all parties interested in hooking a particular event. To illustrate the event notification mechanism, Figure 2 shows a partial definition of the Umbrella class and some sample hook object interfaces.

Event notification above the Umbrella level operates in a similar manner to a priority-based interrupt system in that code running within the arena is suspended whilst the registered handler object processes each event. Each type of event has a priority level that is used to manage concurrent event notifications. When a higher priority event occurs during the processing of a lower priority event, the low priority event handler is suspended. Thus, event notifications can nest according to priority so that a higher priority event can suspend the handling of a lower priority event but not vice versa. The priority order of events from highest to lowest is *Activate*, *Exception*, *IRQ*, *Tick*, and *IAC*.

An important aspect of the upcall/event handling mechanism is how suspended computations are resumed following an upcall event. According to the Charm philosophy, once an upcall is delivered to an arena, the arena is responsible for resuming the interrupted computation. Thus, it is not necessary (nor possible) to perform any system calls to resume from an upcall. Within an arena, the Umbrella assumes the responsibility for restarting any suspended computation using the *resume* operation which forms part of the Umbrella's interface to the upper layers. *Resume* operates by restoring a saved register context associated with a suspended computation. When a registered hook object is notified of an event, it is passed the saved register context of the interrupted computation. The register values may be manipulated by the hook object as necessary. Once the hook object has finished processing the event, it must call the Umbrella to *resume* a particular computation using a given register

context. Usually, this will be the register context saved prior to the event notification. However, the implementation of a thread scheduler may elect to track multiple sets of registers (one per thread) and resume using a different context.

```
struct RegFrame; // Definition of CPU Register Set.

struct IActivateHandler
{
	virtual void notify() = 0;
};

struct ITickHandler
{
	//  "frame" refers to the saved registers of the suspended code.
	virtual void notify(RegFrame &frame) = 0;
};

class Umbrella
{
public:
	…
	void set_activate_hook(IActivateHandler *pHandler);
	void set_tick_hook(ITickHandler *pHandler);
	…
	void resume(RegFrame &frame);
	…
};
```

Figure 2: Umbrella interface definitions

When the Umbrella is asked to resume a computation following the handling of an event, it first checks to see if there are any other events queued for delivery to the upper layers. If so, the resume is aborted and one of the queued events is delivered instead. Only when there are no queued events does the Umbrella actually resume the specified computation. It is the responsibility of the upper layers to ensure that event handlers do not compete to resume different computations. In practice this is not difficult to achieve. This last point raises one of the key benefits of the self-management concept. The code running within an arena is entirely responsible for its own fortunes. That is, any bugs within the code will only affect *that* code and will never affect the performance of another arena.

3 Implementation Issues

The Umbrella, by its very nature, contains some extremely low-level code to manage the arena. A substantial portion of this code is written in assembly language for efficiency and to enable precise control over the sequence of instructions. However, the upper interface of the Umbrella is expressed entirely in object-oriented C++. The implementation of the Umbrella faces a number of technical challenges that must be dealt with to ensure the correct operation of the arena. This section describes these challenges and how they were met.

The first difficulty to be overcome by the Umbrella is the low-level handling of each of the upcall points. From the arena's point of view, the Charm kernel delivers upcalls like hardware interrupts. This means that the arena's code stream is suddenly suspended and control is transferred to a handler defined by the Umbrella. When the handler receives control, most of the machine registers are live with values from the suspended computation. The Charm kernel saves the PC of the interrupted computation as well as the stack pointer (SP) register, which can be used as scratch storage. These registers are held within the public kernel meta-data for the arena and can easily be directly retrieved by the Umbrella when necessary. The handler must switch to its own stack and save the remaining registers before converting the upcall into a software event and notifying the registered hook object.

Managing the saving of registers in response to an upcall is theoretically a straightforward task. However, the arena has no control over atomicity of operations, which creates problems during the critical period of an upcall handler when it is trying to preserve the state of the interrupted computation. This problem arises because the Charm kernel delivers upcalls to the arena whenever events occur, regardless of what the arena is currently doing. Therefore, an upcall can conceivably occur between any two instructions. To overcome this difficulty, it is necessary to employ some low-level coding tricks to properly guarantee atomicity. Therefore, the following design features were introduced:

1. The public meta-data for each arena contains one register save area for each possible upcall. Thus, there are seven separate save areas per arena. Each save area holds the PC and SP for the interrupted computation. This was necessary to circumvent atomicity problems arising when the kernel delivers

multiple events to the arena in quick succession. For example, an IRQ followed immediately by a TSE would cause the saved registers from the IRQ to be overwritten before the arena handler could preserve them. Therefore, this feature allows one instance of each type of upcall to be active concurrently.

2. Once an upcall has been dispatched to an arena, other upcalls can interrupt at any time, even before a single instruction of the handler has been executed. We were therefore forced to define physical regions of code that are deemed *critical* in terms of behaving atomically. The Umbrella code is organised so that all of the *critical* regions are physically adjacent in memory. This allows the value of the PC to be tested to see whether the interrupted code is *critical* or not. We believe that a similar technique is used in the exo-kernel [6]. Using this tactic, an upcall handler can check to see exactly what kind of code it interrupted. If the code is in the *critical* region, it is possible to determine exactly which piece of *critical* code was interrupted, which is important for managing the priority ordering of events. Thus, a typical upcall handler will begin by saving a minimum number of registers. It will then check to see what it interrupted and act appropriately. For example, if an IRQ event interrupts the handling of a TSE event, the IRQ handler will simply flag a pending IRQ event and resume the TSE handler as soon as possible. This occurs in a manner so as not to disturb the state of the TSE handler, thereby preserving its notion of atomicity.

Using the above tactics, it is possible to build upcall handlers that execute atomically and correctly deliver events to the C++ hook objects. The next issue to resolve is how to deal with *Time Slice End* events in a timely fashion. This is an important issue due to the fact that the Charm kernel will forcibly terminate any arena that fails to yield promptly following a TSE upcall. Unfortunately, handling a TSE upcall is relatively difficult since it may interrupt any of a number of pieces of *critical* code and must be allowed to proceed without delay. The TSE handler operates by preserving the state of the interrupted code stream and then examining what it was. Based on this, a status code is set for the corresponding TSS handler to pick up. The status code directs the TSS handler to where the state was preserved and what action to take to resume the interrupted computation. For example, if the TSE event interrupted the *critical* portion of the *Exception* handler, the state of the handler is preserved in a special save area and the TSS handler is directed to resume from this save area. In some cases, it is not necessary for the TSE handler to preserve any state due to the idempotent nature of the interrupted computation. For example, if the TSE event interrupted the Umbrella's *resume* sequence, the TSE handler will not bother to preserve any state and will indicate that the TSS handler should simply restart the *resume* sequence. Thus, all upcall handers are required to co-operate to ensure the correct operation of the arena.

4 Case Study: A Simple Thread Scheduler

In this section we show how the Umbrella abstractions may be built upon to construct a simple thread management system. The significance of this example is that the entire thread management system may be constructed entirely at user level using only the abstractions provided by the Umbrella, which is also completely implemented at user level. Thus, a component of the operating system that is traditionally intimately related to kernel data structures has been completely de-coupled from the kernel thereby easing the implementation of mobile and persistent threads.

The class *SimpleThreadManager* shown in Figure 3 implements two interfaces – one high level and one low level. The *IThreadManager* interface (not shown) provides the application with traditional thread facilities. The low-level interface implements the *ITickHandler* interface described earlier permitting it to plug into the Umbrella using its *set_tick_hook* method. Notice that the constructor for *SimpleThreadManager* takes a pointer to the Umbrella as a parameter. This permits it to make use of the Umbrella's *resume* method when threads need resuming.

Whenever a *tick* event occurs, the currently running thread is suspended by the Umbrella as described above and the *notify* method in the registered *SimpleThreadManager* is called. The thread manager is free to implement whatever policy is appropriate to the application and either saves the suspended thread's register set in a data structure and calls resume with a previously saved register set, or calls resume directly with the same register set.

```
class SimpleThreadManager :
       public ITickHandler,      // lower interface
       public IThreadManager     // upper interface
{
public:
       SimpleThreadManager( Umbrella *pu );
       virtual ~SimpleThreadManager();

public:  // IThreadManager interface
       virtual void       add_thread(Thread *t);
       virtual void       remove_thread(Thread *t);
       virtual Thread *   get_current_thread();
```

```
        virtual void        kill_thread(Thread *t);

public:  // ITickhandler interface
        virtual void        notify(RegFrame &f);

private:
        // data structures elided
private:
        // instance data elided
};
```

Figure 3: Thread Manager definition

5 Conclusions

The de-coupling of user level software from the kernel is motivated by the desire to implement mobile and persistent processes. We believe that the approach described in this paper has many potential benefits including: the ability to efficiently construct application systems in whatever manner is appropriate, low kernel/user-level coupling, ease of constructing synchronisation mechanisms and the ability to implement in an environment that is free from operating system imposed policy. The approach is in line with the ideas first espoused by Engler [6] that operating systems should not provide any abstractions. Instead, operating systems should do little more than multiplex the hardware and provide some basic functionality from which higher level abstractions may be constructed.

Despite the deliberate lack of abstractions provided by the kernel, we have shown how the self-managing protection domains provided by the kernel enable an object-oriented threading system to be constructed that is isolated from the kernel. We believe that this approach makes concurrency control mechanisms both easier to engineer and potentially more efficient, thus easing the creation of mobile and persistent processes.

The Charm kernel has been implemented and runs on the Pentium architecture. It consists of 5000 lines of C++ and 800 lines of assembler and compiles into an executable image 90KB in size. We believe that the kernel in its current form contains enough functionality to efficiently implement a wide variety of application systems.

6 Acknowledgements

This work is partially supported by ESRC Integrated Operating System Support for Large Heterogeneous Archives Project HS 519 25 5043 which is part of the Analysis of Large and Complex Datasets initiative and by EPSRC grant number GR/M78403 under the Distributed Information Management initiative.

7 References

[[1] L. Cardelli and A.D. Gordon, "Mobile Ambients", *Lecture Notes in Computer Science*, **1378**, pp. 140-155, http://www.luca.demon.co.uk/Bibliography.html#Abstractions for Mobile Computation, 1998.

[[2] D. Chamberlin, M.M. Astrahan, *et al.*, "A History and Evaluation of System R", *Communications of the ACM*, **24**(10), pp. 632-646, 1981.

[[3] A. Dearle, "Towards Ubiquitous Environments for Mobile Users", *Internet Computing*, **2**(1), pp. 22-32, ftp://jeroboam.cs.stir.ac.uk/pub/papers/mobileic.pdf, 1998.

[[4] A. Dearle and D. Hulse, "The Charm Operating System Web Pages", http://persistence.cs.stir.ac.uk/Charm/, 1999.

[[5] A. Dearle and D. Hulse, "Operating System Support for Persistence: Past, Present and Future", *Software Practice and Experience*, **Special Issue on Persistent Object Systems 30**(4), pp. 295-324, 2000.

[[6] D.R. Engler, M.F. Kaashoek, and J.W.O.T. Jr. "Exokernel: An Operating System Architecture for Application-Level Resource Management", in *Proceedings of Fifteenth ACM Symposium on Operating System Principles*, Copper Mountain, Colorado, pp. 251-266, 1995.

[[7] J. Rosenberg, A. Dearle, D. Hulse, A. Lindström, and S. Norris, "Operating System Support for Persistent and Recoverable Computations", *CACM*, **39**(9), pp. 62-69, 1996.

[[8] M. Stonebraker, "Operating System Support for Database Management", *Communications of the ACM*, **24**(7), pp. 412-418, 1981.