User-level threads on a general hardware interface

K.R. Mayes, S. Quick and B.C. Warboys

Centre for Novel Computing, Department of Computer Science,
University of Manchester, Oxford Road, Manchester, UK.

1 Introduction

Moving resource management out of the operating system kernel facilitates a high degree of customisation. The lowest layer of the Arena system provides an abstract interface to conventional processor hardware (Mayes, 1993; Quick, 1995). The idea is to encapsulate the hardware behind an interface with certain low-level concepts which are generally applicable to any processor. Localization of hardware-dependency has the effect of increasing modularity and thus portability. This encapsulation, termed the Arena hardware object (HWO) supports portable user-level customizable resource management (Mayes et al., 1994). The aim is to remove resource management policy from the HWO whilst maintaining its integrity. The present paper deals with the Arena approach to the provision of pure user-level threads. Native implementations on Sparc and i486 processors are briefly described and performance figures are given. The compromise between reducing policy and maintaining integrity in the HWO implementations is discussed[1].

2 Features of HWO

Before launching into a more detailed description of the Arena approach, some of the basic features of the HWO can be summarised:

1. An abstract interface to the hardware is provided which deals only with register context blocks, hardware address translation contexts, virtual address ranges and page contexts.

2. The 'trap' interface is internal to the HWO. That is, HWO code will be executed in supervisor mode only when necessary for protection or for access to privileged instructions or data.

3. The HWO device interface is expandable for system-specific devices, e.g. network, hard disk and terminal drivers.

4. There are no kernel threads, and no permanent kernel state for user-level threads.

5. Events may be handled at user-level via an *upcall* mechanism. Upcalls result in the scheduling of a user-level thread.

6. The upcall mechanism depends on a close association between the HWO and the user-level Process Manager. This coupling is in the form of the the Process Manager *switch()* routine and some shared state.

7. The HWO has just one kernel stack. On trapping to privileged mode, thread contexts are saved, but not on the kernel stack.

8. The design includes a *computation* interface for creating new applications. The HWO is responsible for scheduling the computations. Thus virtual processors are provided by the HWO at this coarse-grain, application, level.

3 Threads and the Hardware Object

The full HWO interface is given in the Appendix. This paper is concerned with the thread-related interface primitives. These deal with *thread context* structures. These structures are conventional, and they reside at *user-level* in a thread context block (*tcb*). However, the thread context structures are defined and interpreted by the HWO. That is, HWO

[1] This work is supported by EPSRC grants 91309499 and GR/J84045

interface operations are provided to initialize, reinitialize and terminate thread contexts. These allow user-level Process Managers to create threads, whose execution contexts are processor-dependent, in a hardware-independent manner. In addition, two HWO interface primitives, *hwo_getregctx()* and *hwo_setregctx()* are provided to switch threads. An Arena Process Manager must provide a *switch()* routine, which is the general entry point for all scheduling policy decisions. The *switch()* routine is responsible for calling *hwo_getregctx()* and *hwo_setregctx()*.

Thread switching When switching from user-level $thread_a$ to $thread_b$, the role of the HWO is simply, on *hwo_getregctx()*, to save the current register state to $thread_a$ *tcb* and, on *hwo_setregctx()*, to restore the register state from $thread_b$ *tcb*. The HWO is completely the slave of the user-level Process Manager which determines that $thread_b$ should be scheduled. The HWO contains no concept of 'thread' or of 'scheduling'. The CPU control component of the HWO deals only in register contexts. The *hwo_setregctx()* function does not normally return to the caller because the processor continues execution in the new thread context. Arena threads reside entirely at user-level. There are no kernel threads to provide virtual processors. At the same time, thread scheduling is entirely hardware-independent.

Upcalls and event threads Hardware events are handled in the first instance by the HWO. However, the HWO may *defer* further processing by instigating an *upcall* to a *user-level event handler* which resides in the application domain.

Deferring the processing of a trap has two advantages. Firstly, the time spent at the higher interrupt priority level is limited. Secondly, executing the event handler at user-level allows the application, or its run-time system, to apply policy decisions to its execution. This latter feature is of importance in a customizable system like Arena.

The Arena *upcall mechanism* is based on threads. Each user-level event handler is represented by a thread of control which is made runnable when deferred handling of an interrupt or exception is required. Furthermore, because these handler threads are themselves schedulable entities, the user-level scheduling policy for event-threads can occur in the same code as application-thread scheduling policy. This provides the application with control over the scheduling of event handling.

However, this arrangement does mean that the user-level manager object/HWO interface is asymmetrical. Downcalls from managers to the HWO occur simply as function calls to HWO operations – the HWO is treated as another entity in the system. In contrast, upcalls from HWO to user-level manager code are not function calls, but rather are implemented by scheduling the appropriate thread. This asymmetry does however mean that downcall/upcall cycles are avoided.

The upcall mechanism involves running the *switch()* routine so that an event thread can be scheduled. On receiving an event, the HWO makes the appropriate user-level event thread runnable, loads the *switch()* execution context and returns to user-level. The *switch()* routine applies its event scheduling policy and determines which thread to run. The CPU control primitives of the HWO are then called to save the old, and to restore the new, thread context. Event threads are made runnable by setting an element in state shared between HWO and user-level. Shared counts of events are incremented by the HWO on their occurrence and are decremented by the Process Manager on their being serviced. For efficiency, multiple events can be delivered in a single upcall.

4 HWO context switching implementations

This section essentially deals with the implementations of the *hwo_getregctx()* and *hwo_setregctx()* primitives of the HWO. These issues are discussed in so far as choices impinge on the efficiency, the policy and the integrity of the HWO implementations.

The HWO interface has been implemented native on two processors: the Cypress Sparc, and the Intel 386/486. These processors differ from each other sufficiently to present challenges to the implementation of a single low-level interface. The *switch()* routine is normally invoked in two situations:

1. cooperative switching e.g. by a yieldThread() routine.

2. upcall from the HWO as a result of a trapping event.

In both cases, assuming that *switch()* is switching from $thread_a$ to $thread_b$, the processor state will be saved to the *tcb* of $thread_a$. However, in the upcall case, arising from an event, the processor state will already have been saved by the HWO. The Arena HWO has only one kernel stack, per processor, on which all HWO supervisor-level handling is done. The user-level context is not saved to the kernel stack, but is saved to some *logical* 'event context buffer' (*event_cb*) whose implementation is different on different processors. When *switch()* is invoked with the upcall, the saved context is copied from the *event_cb* to the thread *tcb* and stack.

The behaviour of the *hwo_getregctx()* and *hwo_setregctx()* pair in different HWO implementations depends on:

1. whether *switch()* was invoked by an upcall, and, in the case where it is an upcall, how the *event_cb* is implemented.
2. whether the *hwo_getregctx()/hwo_setregctx()* pair requires a change in privilege level.
3. what facilities the processor provides for dealing with page faults.

4.1 Switch() invoked by upcall

The issue here is how the HWOs implement the *event_cb* on different processors (Table 1).

The Sparc implementation uses a thread-local user-level *tcb* for the major component of the *event_cb*. However, for optimisation in case no thread switch occurs, register windows occupied by the thread are not flushed, and thus the register window file effectively becomes part of the Sparc *event_cb*. The use of the Sparc register window file as part of the event context buffer requires that different contexts can coexist in the register file. User-windows are only flushed when a new thread is scheduled (see next subsection). That is, the Sparc HWO attempts to avoid costly window flushing. The price paid for this optimisation is in the complexity of the code.

The 3/486 implementation uses the privilege-level 0 stack as the *event_cb*. On an event, there is a change to privilege-level 0 and the stack pointer is switched to the privilege-level 0 stack. The small register state of the 3/486 is saved to the level 0 stack and is subsequently copied to the thread *tcb* if a thread switch occurs. The stack pointer of the processor is switched to the distinct HWO kernel stack, leaving the level 0 stack free to be used as the *event_cb*.

processor	event context buffer
Sparc	user-level thread context block + register window file
i3/486	privilege-level 0 stack

Table 1: Implementation of the event context buffer on two processors. The event context block is used to save the user-level thread context when an event occurs.

4.2 Switch() and privilege levels

The issue here is how to avoid unnecessary changes of privilege level. The *hwo_getregctx()* and *hwo_setregctx()* implementations involve a trap only if necessary.

The Sparc implementation is complicated by the need to save the thread register windows to the stack. This requires a trap, and would mean that *hwo_getregctx()* would always trap. This would be an unnecessary overhead if the same thread is immediately rescheduled. Since it is necessary to trap in order to write the new thread context to the processor, *hwo_setregctx()* must always trap in both cooperative and upcall invocations of *switch()*. Thus, flushing the thread register windows is deferred until *hwo_setregctx()* when it can be determined if a thread switch is actually occurring. That is, saving of the old register context to the stack is deferred until a thread switch is detected in *hwo_setregctx()*.

The 3/486 implementation is more straightforward. Since no change in privilege is required to write to the processor state, *hwo_setregctx()* does not trap. Furthermore, the register component of the processor state can be saved to the thread stack without trapping. This is true of both cooperative and upcall invocations of *switch()* on 3/486.

4.3 Page faults during saving

The issue here is how to handle non-resident context save areas.

Page faults can occur on both the user-level *tcb* and on the user-level stack. The Sparc and 3/486 differ in handling this problem. The 3/486 HWO handles page faults, occurring within a trap, on its kernel stack. The thread context already saved in the *event_cb* can then be copied after resolving the fault. The Sparc HWO however cannot straightforwardly handle traps within traps because only one window is guaranteed to be free for trap handling. The Sparc therefore requires a *window save area* which is always resident. Various thread switch optimisations are made possible when *each thread* has its own kernel level window save area. Arena threads have no kernel state, and so cannot take advantage of kernel-level per-thread window save area optimisations. There is a single window save area per processor in the HWO for use in the case where a thread stack is not resident.

5 Performance results

The target machine for the Sparc port was a 13-node European Declarative System prototype (EDSP) machine (Ward and Townsend, 1990). This has a distributed-store architecture of nodes closely -coupled on an internal network. Each node consists of two 40 MHz Cypress Sparc processors sharing 64 MB of nodal store. The processors each have a 64K external direct-mapped MMU cache, with snoopy coherency between the two processors on each node. The results were obtained using one processor, whilst the other processor was in a monitor-level loop. The 3/486 results were obtained on a conventional 66 MHz 486DX2 with 8K internal and 128K external cache.

Three simple user-level Process Managers, providing cooperative, priority and time-slicing scheduling policies for user-level threads, were implemented to run on the HWO interface. The same Process Manager codes will run with no changes on both Sparc and 3/486 HWO implementations.

The performance figures for context switch and thread creation are given in Table 2.

action	i486	EDSP Sparc
context switch (HWO)	4	17
context switch (PM)	13	18
create thread (PM)	38	33

Table 2: Performance figures (microseconds) for Arena Hardware Object (HWO) and a user-level Process Manager (PM). The PM thread creation times are for threads whose stacks have been obtained from a pre-allocated pool.

The figure for the HWO context switch represents the best performance that the system can exhibit. This is the time taken to switch between two user-level thread *contexts*. There is no scheduler list manipulation and no locking of data structures. The discrepancy between the context switch times for the Sparc and the 486 is explicable as a combination of differences in processor speed, relative cache-miss costs, and Sparc register window save/restore costs. The context switch figures given in Table 2 for Sparc is for one user-level register window. The context switch cost on Sparc was found to rise by 6 microseconds for each extra user-level window. This per-user-level window cost comprises the cost of an overflow fault to the thread stack plus the cost of an underflow fault to restore the window from the thread stack. This time includes testing the thread stack and *tcb* for residency, with a cost of 1 microsecond. This performance, with *lazy restoration* of windows by underflow faults, can be improved by a policy of *eager window restoration*. As an experiment to determine the feasibility of making eager window restoration available to say compilers, a shared variable was provided which could be set to specify the number of windows to be explicitly restored from the stack of the thread being cooperatively switched-in. This eager restore (with overflow fault to stack for saving, as before) reduced the cost per user-level register window from 6 to 4 microseconds.

With a user-level Process Manager running on top of the HWO and switching between user-level *thread objects* using a *yieldThread()* routine, including taking locks, the switch time rises. On Sparc, the per-user-level register window cost is unaffected. The user-level Process Manager provides a *createThread()* routine, which is used here to provide figures for thread creation times.

6 Related work

Other systems seek to provide minimal software wrappers for hardware, e.g. Aegis (Engler et al., 1994) and PANDA (Assenmacher et al., 1994).

There are issues relating to the abstractions that should be provided by such systems, and whether user-provided code should be able to be inserted into this hardware-dependent layer. The PANDA system provided two abstractions: protection domain and virtual processor. Aegis allowed applications to run user-code safely in the kernel. The Arena approach is to provide interface primitives to low-level hardware abstractions, but to disallow direct access to the processor hardware. Mukherjee et al. (1994) presented a hardware-independent interface for threads with a *hw-save-context() / hw-restore-context()* pair, though their interface was implemented on top of operating systems, and not, as here, on the hardware itself.

Upcalls in Arena are associated with threads and are thus asynchronous. However, the term 'upcall' was used by Clark (1985) to refer to a synchronous access to services provided by a higher level in a layered system. The 'software interrupt' of Psyche (Marsh et al., 1991) was executed in the context of a kernel thread, after switching the stack and program counter, and entering user-level. A thread-oriented upcall mechanism was described by Anderson et al. (1992) where the kernel created and scheduled a new kernel thread whose entry point is the required upcall code of a user-level threads package. Arena has an upcall mechanism based on dedicated, purely user-level, threads scheduled directly from user-level.

There is a single kernel stack for each processor on which the Arena HWO is running. On this basis the Arena system falls into the 'interrupt kernel' model (Draves et al., 1991). Where a reschedule is required after entry to the 'kernel', thread state is saved in a user-level block. However, there is no explicit 'continuation' code associated with this saved context: it is resumed by the normal user-level scheduler code.

One of the major problems in the implementation of the HWO has been dealing with Sparc register windows. Sparc register window handling has been dealt with quite extensively (e.g. Anderson et al., 1991; Liedtke 1993; Mueller et al., 1994). The Arena HWO uses the register file as a temporary context buffer. It allows the coexistence of multiple contexts in the Sparc register window file. This can be viewed as a variant of the lazy context switch model (Liedtke 1993) where windows from different threads are allowed to coexist.

7 Discussion

Arena is intended to be a system that will operate in both general purpose and embedded application situations. The HWO interface is an attempt to abstract hardware resources and provide mechanisms to access them. The HWO interface primitives are intended to be sufficiently low-level to allow policy to be specified at user-level for application-specific resource management. At the same time, the HWO should be robust and safe. Unfortunately robustness and safety imply that some policy has to be introduced into the HWO. So, for example, the thread switching code can tolerate non-resident context data structures and stacks. This is the more general case, but represents a policy decision fixed into the HWO. Again, the Sparc HWO imposes policy for handling register windows, since it could not be considered to be safe to allow two applications sharing the same processor to apply different policies to register window handling. Fortunately, the provision of robustness and safety tend to converge with hardware-dependency. Hardware-specific detail can be hidden beneath the HWO interface, allowing resource management, exemplified here by Process Managers, to be portable across HWOs on different processors.

8 References

Anderson, T.E., H.M. Levy, B.N. Bershad and E.D. Lazowska (1991) The interaction of architecture and operating system design. Proceedings of 4th International Conference on Architectural Support for Programming Languages and Operating Systems, 108-120.

Anderson, T.E., B.N. Bershad, E.D. Lazowska and H.M. Levy (1992) Scheduler activations: Effective kernel support for the user-level management of parallelism. ACM Transactions on Computer Systems 10(1), 53-79.

Assenmacher, H., T. Breitbach, P. Buhler, V. Hübsch, H. Peine and R. Schwarz (1994) Meeting the application in user space. Proceedings of the 6th ACM SIGOPS European Workshop, September, 82-87.

Clark, D.D. (1985) The structuring of systems using upcalls. Proceedings of 10th ACM Symposium on Operating System Principles, 117-180.

Draves, R., B. Bershad, R. Rashid and R. Dean(1991) Using continuations to implement

thread management and communication in operating systems. Proc. 13th ACM Symp. on operating systems principles (October), 122-136.
Engler, D., M.F.Kaashoek and J. O'Toole (1994) The operating system kernel as a secure programmable machine. Operating Systems Review 29(1), 78-82
Liedtke (1993) Lazy context switching algorithms for Sparc-like processors. GMD Technical Report No 776.
Marsh, B.D., M.L. Scott, T.J. LeBlanc and E.P. Markatos (1991) First class user-level threads. Operating Systems Review 25(5), 110-121.
Mayes, K.R. (1993) Trends in operating systems towards dynamic user-level policy provision. University of Manchester Computer Science Technical Report UMCS–93–9–1.
Mayes, K., S. Quick, J. Bridgland and A. Nisbet (1994) Language- and application-oriented resource management for parallel architectures. Proceedings of the 6th ACM SIGOPS European Workshop, September, 172-177.
Mueller, F., V. Rustagi and T.P. Baker (1994) Mythos - a micro-kernel threads operating system. Technical report 94-091, Dept of CS, Florida State University.
Mukherjee, B, G. Eisenhauer and K. Ghosh (1994) A machine independent interface for lightweight threads. Operating Systems Review 28(1), 33-47.
S. Quick (1995) A generalised hardware interface for operating systems. In Preparation.
Ward, M. and P. Townsend (1990) EDS hardware architecture. Lecture Notes in Computer Science 457, 816-827.

9 Appendix

The full HWO interface:

errno_t **hwo_getregctx** (thread_ctx *t)
errno_t **hwo_setregctx** (thread_ctx *t)
errno_t **hwo_initthread** (thread_ctx *t, caddr_t entry, caddr_t stack, caddr_t stacksz, hatid_t hatid, eventpri_t eventpriority, caddr_t tidyup_entry, boolean_t runnable, void *arg0)
errno_t **hwo_reinitthread** (thread_ctx *t, caddr_t entry, caddr_t stack, caddr_t stacksz, caddr_t tidyup_entry)
void **hwo_exitthread** (void *tidyuparg)
boolean_t **hwo_isrunnable** (thread_ctx *t)
void **hwo_setrunnable** (thread_ctx *t, boolean_t flag)
boolean_t **hwo_threadavailable** (thread_ctx *t)
eventstate * **hwo_geteventstate** (uint cpuid)
eventpri_t **hwo_currenteventpri** (void)
void **hwo_enter_switchenvironment** (uint)

error_t **hwo_create_hat_ctx** (hatid_t *newhatid)
error_t **hwo_free_hat_ctx** (hatid_t hatid);
error_t **hwo_set_hat_ctx** (hatid_t newhatid);
hatid_t **hwo_current_hat_id** (void);

error_t **hwo_validate_va_range** (hatid_t hatid, caddr_t start, uint sz, uint type, rangeid_t *range)
error_t **hwo_invalidate_va_range** (hatid_t hatid, rangeid_t *rangeid)
error_t **hwo_map_va_range** (hatid_t tohat, caddr_t start, uint type, hatid_t fromhat, rangeid_t fromrange, rangeid_t *newrange)
errno_t **hwo_get_bootinfo** (bootinfo *b)

errno_t **hwo_get_pagecontext** (hatid_t hatid, caddr_t start, uint pages, pagecontext_t *pginfo)
errno_t **hwo_getref_pagecontext** (hatid_t hatid, caddr_t start, uint pages, pagecontext_t *pginfo)
errno_t **hwo_set_pagecontext** (hatid_t hatid, caddr_t start, uint pages, pagecontext_t *pginfo)
void **hwo_flush_cache** (hatid_t hatid, caddr_t start, uint sz)

errno_t **hwo_create_computation** (hatid_t hatid, caddr_t entry, resources *res)
void **hwo_exit_computation** (int status)

uint **hwo_getcpuid** (void)
uint **hwo_getpeid** (void)
int **hwo_testandset** (int x, int *s)
int **hwo_test** (int *s)
uint **hwo_atomicswap** (uint x, uint *s)
uint **hwo_gettime** (void)
uint **hwo_gethighrestime** (void)
uint **hwo_gettimeout** (void)
void **hwo_settimeout** (uint ticks)