Real-Time Arena
A User-level Operating System for Co-operating Robots

Sam Kingsbury, Ken Mayes and Brian Warboys.
Centre for Novel Computing, Department of Computer Science,
University of Manchester, Oxford Road, Manchester, UK
sam.kingsbury@cs.man.ac.uk Tel: +44 0161 275 6135

Abstract

This paper presents the design of a new version of the Arena operating system targeted at real-time applications. Arena is a run-time operating system for parallel applications. The main principle of Arena is to provide predictable and controllable resource management. This is achieved by placing resource management at the user level of the system, thereby allowing any degree of application influence. Extensions to the system for improved real-time performance are discussed in detail. This version of Arena is to provide a research platform into co-operative robotics, and so the interface between the target robot hardware (built especially for this project) and Arena is also examined in detail.

1. Introduction and Background

The Arena operating system has been developed by previous research to investigate how the efficiency of parallel systems may be improved by through operating system provision for *predictable and controllable resource management* by the application (Mayes [8]).

Commonly run-time systems rely on the native operating system to provide resources such as communication primitives or virtual memory. Inefficiencies arise where there are incompatibilities between these two levels of machine, and where compromises have to be made in the run-time system implementation. Furthermore, general purpose operating systems may introduce unpredictability into the execution of an application. The system may determine when housekeeping daemons run, for example, or when deferred I/O processing occurs, without reference to the requirements of the application.

One way around these problems is to link the application with only that resource management it requires, load this object onto the *bare hardware*, and let it run. This is the situation found in many real-time applications. Ghosh et al described such real-time systems using the term 'operating software' in order to emphasise the intimate connection between real-time application and real-time operating system (Ghosh [3]). However, it is unreasonable to expect parallel application developers to write their own low-level, hardware-specific code. Arena provides a solution to this problem - the provision of a generic 'hardware' interface that can reside on the nodes of the parallel machine and that can host the run-time operating system.

This paper describes a new version of Arena developed to support real-time applications, situated in the real-world. The target hardware is a 'network' of autonomous mobile robots intercommunicating using low-bandwidth radio. Each robot has a single 'Amulet' RISC processor (see Furber [2]), 512Kbytes of RAM, and an array of digital and analogue inputs and outputs. The issues surrounding Arena's support for interaction with the real-world on the robot platform (which implies operation in real-time) are described in detail. Also presented are details of some applications based on neural network navigation, that serve to validate the Arena approach by enabling novel research into co-operative robotics.

2. Arena

Moving all policy into user-level resource manager libraries effectively extends the run-time system into the operating system. Figure 1a shows schematic diagrams comparing a conventional monolithic operating system with the approach taken in Arena. All the hardware-dependent code for manipulating the hardware resides in a 'nanokernel', or run-time executive, which is accessed by a generic low-level interface. Having a general low-level interface facilitates system portability.

The structure of Arena is shown in Figure 1b. There are two components. One component consists of *hardware-independent* resource manager libraries for linking to application. These represent a run-time

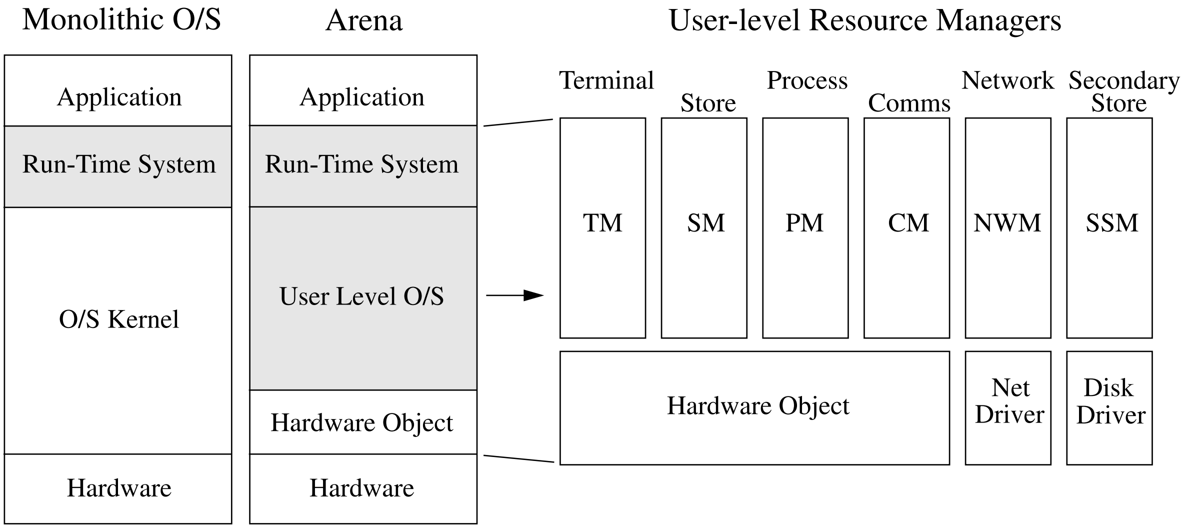

Fig. 1a. Comparison of operating system approaches. (Library components shaded).

Fig.1b. Structure of the Arena System.

operating system. The other component is the *hardware-dependent* code. The run-time operating system gains access to hardware-dependent code via an abstract interface. This interface is called the Hardware Object (HWO) interface. *The main principle in this division is to distinguish between policy and mechanism*. The HWO is intended to give access to the mechanisms provided by the hardware. The resource manager libraries determine the policy in the use of these mechanisms.

Arena has been developed to support computation-intensive applications such as parallel relational databases (Mayes [7]), and to give virtual shared memory support for a parallel Fortran compiler (Ford [1]). The target hardware is typically a tightly coupled, highly parallel hardware architecture. Until now, Arena has been implemented on the EDS Sparc architecture (Ward [11]) and PC-AT Intel 3/486 processor based architectures.

2.1. Hardware Object

The HWO is essentially a run-time executive that provides a primitive machine interface to the resource managers. This interface presents a generalised view of processor hardware, so that porting of the system to various hardware platforms is facilitated. The HWO interface provides a set of *downcalls* which is used by the hardware-independent, user-level, manager code. There are, at present, 34 primitives in the HWO interface, not including device-specific interface primitives. This structure represents an instance of the system that would reside on each node of a distributed store multicomputer. The trap interface is internal to the HWO, whose interface is accessed via a library. The HWO code is implemented at user-level whenever possible. However, for protection, and access to supervisor-mode instructions/registers, some library routines consist of a stub which traps to supervisor and accesses the code conventionally via the trap table.

The HWO deals mainly with low-level concerns such as general *register contexts* and *address translation*. Interfaces to devices such as disks and networks are an extension of the HWO interface. The reasoning here is that different systems will have different devices which may, in turn, require different interfaces.

2.2. User-Level Managers

Arena resource management components are decoupled, interacting via well-defined interfaces, and implemented in C or C++. Resource manager base classes are defined for Process Manager (PM), Store Manager (SM), Secondary Store Manager (SSM), Communication Manager(CM), Network Manager (NWM) and Terminal Manager (TM). Only those managers required by an application need be used. For example, a single-node application needs no NWM, whereas only applications doing disk I/O will need an SSM.

The interface provided by each of these managers has three components:

1. A set of operations used by application code. These operations are at the level of a typical microkernel interface. They serve to map conventional concepts on to the low-level HWO interface. The PM, for example, provides operations for manipulating and switching *threads* and *address spaces*. Threads in Arena are entirely user-level constructs. The HWO has no concept of a *thread*. The HWO only provides the concept of a *register context*. The SM operations allow *regions* to be created and mapped. The CM provides message passing via the familiar *port* abstraction.
2. A set of operations used by other managers. These operations allow managers to interact. For example, the PM provides interfaces for the SSM to stop and resume threads which are involved disk I/O. The SM operations allow managers to allocate regions: the PM requires regions for thread stacks; the SSM requires regions for buffer caches. The PM uses an SM operation to obtain a new address space. The NWM provides operations to allow other managers to exchange messages, upwards and downwards, with the network device interface of the HWO.
3. A set of operations used by the HWO. These are termed *upcalls*. Such operations allow the manager to do deferred handling of events at user-level. These routines are executed by distinct *event threads* (see section 3.4.). This area of Arena has been significantly extended for the real-time Amulet implementation.

Arena is a customisable system. Resource manager subclasses allow instantiation of a resource manager for particular policies. An application links to the managers providing the required resource management policies. For example, different PMs exist for cooperative, priority-based and round-robin scheduling of threads. Different SSMs can be selected for particular caching policies.

3. Real-Time Arena

Before presenting the real-time version of Arena, the target hardware requirements will be discussed.

3.1. The Amulet Processor

Also developed at Manchester University, the Amulet processor is not yet a commercial product. It is an *asynchronous* implementation of the ARM RISC processor. An asynchronous processor has no global clock and so each section of the processor automatically assumes a quiescent state whenever there is no work to be done. The main advantage of this approach over a synchronous processor is a significant reduction in power consumption. Furthermore the Amulet operates using only a 3 volt electricity supply, therefore, for embedded systems (such as robots) that run on batteries, the life of the system in continuous operation is significantly extended by using this technology (the Amulet-based robots use 3v logic throughout their digital hardware). The Amulet processor is presented in Furber [2].

3.2. Robot Hardware Sub-Systems

A simple robot controller based around a single Amulet has been designed and built. A revision of the device is planned. The current design operates by memory-mapping each sub-system into the Amulet's physical address space, and where appropriate, a device can interrupt the processor when it has data ready. The sub-systems are:

- Infra-Red near-range sensors - Frequency modulated IR burst signals are transmitted from the robot and a measure of the strength of the reflected signal is computed on the basis of the number of pulses received (all using dedicated hardware). The result is buffered and the processor interrupted.
- Sonar far-range sensors - Sonar pulses are emitted by the robot and the time to receipt of each reflected pulse is measured by dedicated hardware. Results are buffered and the processor is interrupted.
- Whiskers, Bump sensors, and Digital inputs - A number single bit digital inputs are provided. Each may be configured as passive (read-only), or active (causes a processor interrupt on a low to high transition).
- Light Dependent Resistors (analogue inputs) - a number of passive byte-wide analogue inputs are available via an AtoD converter. Currently used with LDRs for manoeuvring by servo beacon or

line following etc.

- High Power Outputs - For driving DC motors.
- Motor Shaft Encoders - Provide a passive read-only digital state that represents the rotational position of the motor shafts (for velocity control of the DC motors).
- Serial link - An umbilical is used to initialise the robot from a workstation, and can be re-connected when a situated experiment is complete to dump logged data for subsequent post-processing.
- Radio link - A high latency, low bandwidth radio link is provided for each robot. The link is ‘fault-probable’ and broadcast only (half duplex). Data is transferred from the sender as a single 28 byte packet which is broadcast to all robots *within reception range*. Upon receipt of the whole packet the receiving processor is interrupted.
- Fluxgate Magnetometer (compass) - A passive read-only digital device.
- Other devices include digital outputs for driving status LEDs and an on-board speaker, and a 16 by 2 character LCD display.

Each sub-system is fully independent and need only be active when required by the application (e.g. the Sonars can be turned off when not needed, resulting in no interrupts and hence zero resources are consumed).

The next two sections describe how the real-time version of Arena has been designed and implemented to support use of the co-operating robots.

3.3. Structure of Real-Time Arena

Figure 2 shows the structure of the new real-time version of Arena. The real-time variants of the user-level managers currently provide only a basic level of functionality. The Amulet processor does not

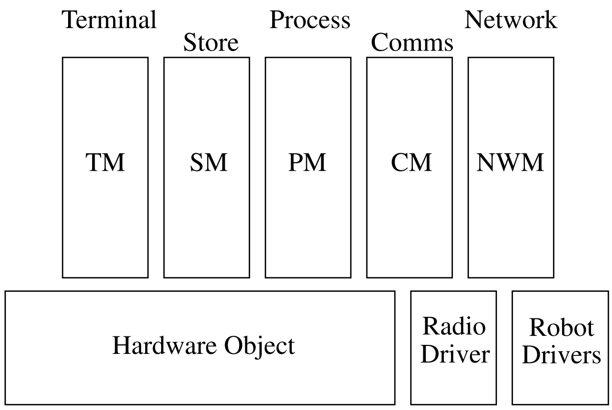

Fig.2. Structure of real-time Arena.

have a memory management unit, and hence the new real-time version of the Arena HWO only implements a subset of the normal 34 function interface (all functions relating to page contexts and address translations are absent). The user-level managers offer the following functionality:

TM - Provides access to the host’s `stdin` and `stdout` when the umbilical is connected.

SM - Attempts to improve on the efficiency of the C `malloc()` function.

CM - This is currently a null manager - i.e. there has been no requirement (as yet) for provision of inter-thread communication. Therefore threads must communicate via shared variables. However a CM could be implemented for any communication paradigm such as message passing or CSP without major problems.

NWM - This is under development and will support the radio link. It will consistently maintain a small region of shared memory between the robots, but other models could be used.

PM - see section 3.4.

3.4. Event Handling and Thread Scheduling

When an interrupt occurs, the Arena HWO makes an *upcall* to the process manager to decide if an event thread needs to be scheduled in response. Since the process manager resides at the user level, the algorithm it uses to make scheduling decisions is entirely under user control. Hence the user can control the 'priority' associated with normal user threads and event threads. This means that the state of the application in control of the robot can be made known to the process manager, and it will vary its scheduling decisions accordingly. This is achieved by associating an event-id with each interrupt that is passed in the upcall so that the user can make a scheduling decision. Arena does not maintain fixed views of relative thread 'priorities' (e.g. given two threads one with event id = 5 and one with event id = 8, the PM may sometimes decide to schedule 5 before 8 and at other times may do the opposite - depending purely on the state of the application).

The real-time Amulet HWO identifies the source hardware sub-system when an interrupt occurs and assigns an *interrupt-id* to the interrupt to indicate this. This hardware interrupt-id is then translated into a software *event-id* via a look-up table within the HWO. The event-id is passed in the upcall to the PM which then makes a scheduling decision based on the event-id (and the current state of the system).

In practice the upcall mechanism can be inefficient for interrupts which occur very frequently, but only occasionally require a higher level software response. To improve efficiency, if a given response is required that is independent of the rest of the system, then the PM can notify the HWO that the specified event-id does not require an upcall, but rather can be handled by a given function invoked directly by the HWO. This handling will occur as soon as the processor accepts the interrupt, and will be executed in a supervisor state (normally the processor leaves the supervisor state and returns to user state as the upcall is made). When the function completes, the thread that was running before the interrupt occurred is resumed, all of which is transparent to the PM. These two methods of handling events are referred to as *internal* event handling (entirely within the HWO) and *external* event handling (within the PM). This facility is shown in more detail in Figure 3.

The Amulet supports two levels of hardware interrupt, IRQ (Interrupt ReQuest) and FIRQ (FastIRQ) (see Figure 3). The robot controller hardware allows each hardware sub-system to be connected to either interrupt pin via jumper selection. The interrupt-to-event translating Look-Up-Table is contained within the HWO (but its contents is defined by the PM) so that the relative priority of internally handled event threads can be dynamically altered[1] - HWO-internal scheduling decisions are made solely on the basis of which outstanding event thread has the highest event-id.

At any time the PM can change (i) the function associated with a given event-id, (ii) the event-id (i.e. priority) associated with an interrupt-id and (iii) the level at which a given event-id is handled (internal HWO level or external PM level).

Various output-orientated robot sub-systems need to be updated frequently and regularly (e.g. the motor drivers when under velocity control). As for input-orientated devices, output devices can also be controlled easily and independently of the rest of the application, again through the use of interrupts. This is achieved by using an event handler invoked at regular intervals by the Amulet's own internal clock-timer.

Given this simple but highly flexible model of thread and interrupt control, it is claimed that whenever a thread switch occurs it is always the application's choice of thread that is scheduled to run next, and with the minimal overhead (HWO level scheduling is highly optimised).[2] These are two of the most critical obstacles to be overcome when constructing a system that is to operate in real-time (these issues are discussed further in Hills [4]).

3.5. Related Work

The performance versus flexibility pay-off obtained from using non-interruptable / interruptable event

1. Though due to the processor design, there is no way to give an internally-handled IRQ event priority over an internally-handled FIRQ event.
2. The HWO is written almost entirely in assembly language. The processor offers two levels of hardware interrupt - IRQ and FastIRQ - Arena exploits these facilities by making full use of the register optimisations offered by FastIRQ mode.

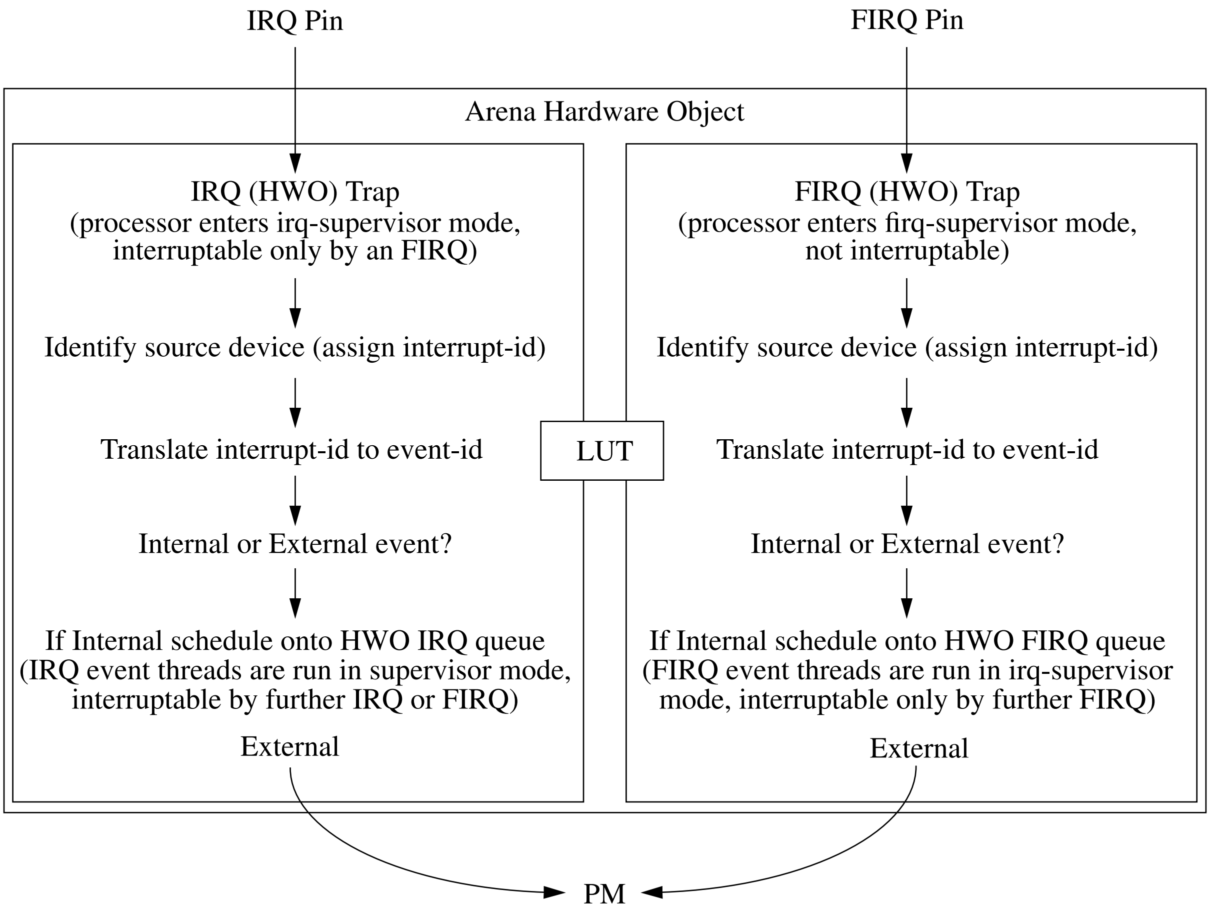

Figure 3. Interrupt handling within the real-time Arena HWO.

handlers respectively is described in Stodolsky [10], and a method called 'optimistic interrupt protection' based on interruptable event handling is described; but the authors do not offer a method of effectively combining both approaches as does Arena.
Within the HWO, Arena treats interrupts as threads for the purpose of synchronisation (and does not mask out further interrupts of the same priority level from occurring). This approach is broadly similar to that used by Solaris 2 as described in detail in Kleiman [5].
Further information about Arena and related systems may be found in Mayes [6] and [8].

3.6. Further Work

Currently real-time Arena is being tested on the first version of the Amulet robot. This will be used as the platform for the development of the software support for the radio link which is currently being built. The full system will then be used for a number of demanding robotics experiments. The initial goal is to develop a distributed variant of a map-building technique based on Kohonen's Self-Organising Feature Maps, as developed by the robotics research group at Manchester University (Nehmzow [9]). Following this stage of the research, it is hoped to build a second generation Amulet-based robot controller and use this to control a number of industrial 'scout' robots manufactured by Nomadic (USA) Inc.

4. Acknowledgements

This project is a collaboration between three research groups within the Computer Science department at Manchester University.[1] The authors would therefore like to thank Prof. Steve Furber, Steve Temple

1. This work is funded under EPSRC grant GR/L 79369.

and Dave Clark of the Amulet Processor group, and Ulrich Nehmzow and Andy Pickering of the Robotics research group. We also acknowledge the work of M.U.C.S. undergradute Simon Rackstraw who built the prototype Amulet based robot controller.

References

[1] R.Ford, A.Nisbet & J.Bull. User-level VSM optimisation and its application. *Proc. 2^{nd} Int. Workshop on Applied Parallel Computing* (Lecture notes in Computer Science 1041), pp 223-232. 1995.

[2] S.Furber. *ARM System Architecture*. Addison-Wesley. 1996.

[3] K.Ghosh, B.Mukherjee & K.Schwan. A Survey of Real-Time Operating Systems. Georgia Institute of Technology, College of Computing, Technical Report GIT-CC-93/18, 1994.

[4] T.Hills. Structured Interrupts. *ACM SIGOPS Operating Systems Review* 27(1). pp 51-68. 1993.

[5] S.Kleiman & J.Eykholt. Interrupts as Threads. *ACM SIGOPS Operating Systems Review* 29(2). pp 21-26. 1995.

[6] K.R.Mayes, S.Quick, J.Bridgland & A.Nisbet. Language- and application-oriented resource management for parallel architectures. *Proc. 6^{th} ACM SIGOPS European Workshop*, pp 172-177. 1994.

[7] K.R.Mayes, J.Bridgland & S.Quick. Customisable Resource Management for Parallel Database Systems. *Parallel Information Processing*, J.A.Keane (Ed), Stanley-Thornes, pp 197-183. 1996.

[8] K.R.Mayes & J.Bridgland. Arena - A Run-Time Operating System for Parallel Applications. *Proc. 5^{th} Euromicro Workshop on Parallel and Distributed Processing*, London, 1997, pp 253-258. IEEE Comp. Soc. Press, 1997.

[9] U.Nehmzow, T.Smithers & J.Hallam. Location Recognition in a Mobile Robot Using Self-Organising Feature Maps. *Information Processing in Autonomous Mobile Robots*, Springer-Verlag, 1991.

[10] D.Stodolsky, J.Chen & B.Bershad. Fast Interrupt Priority Management in Operating System Kernels. *Proc. Usenix Symposium on Microkernels and Other Kernel Architectures*, San Diego, USA. pp 105-110. 1993.

[11] M.Ward & P.Townsend. EDS Hardware Architecture. *Lecture notes in Computer Science 457*, pp 1-26. 1990.