A Structured Semantic Design of Distributed Operating Systems

W. B. DASZCZUK
Institute of Computer Science, Warsaw University of Technology, Nowowiejska str. 15/19, 00-665 Warsaw, Poland

Many new multi-microprocessor systems are available or have been announced to the market. A method of structured operating-system construction to present a distributed hardware environment as a single computer to the user is proposed. Unlike many existing distributed operating systems, which are parallel and process-oriented, the new approach is based on a hierarchical structure of layers. The concept permits the designer to establish almost any dependencies between local operating systems. A distribution of UNIX-like systems in a heterogeneous multi-microprocessor environment is proposed.

Received September 1986, revised September 1991

1. INTRODUCTION

The aims of a classical operating system are primarily to allocate resources to user programs, and secondly to make a computer much more friendly than its hardware. A secure and elegant way to build an operating system for a one- or multi-processor system is to use structured programming methods. The philosophy of structured operating systems design is described in Refs. 3 and 19.

The structured approach to solving a problem is to split it into sub-problems and to establish relations between the sub-problems. This approach follows the Roman principle: *divide et impera*. A layer-structured operating system consists of a hierarchy of layers,[7] which are implemented with the operations of lower layers. A layer 'knows' nothing about the activities of higher layers. In the most abstract terms, the problem that concerns an operating system can be defined: 'run a set of user programs'. The layers of the operating system are resolutions of sub-problems, while the operating system is a resolution of the problem. The desired relations between layers are implemented by inter-layer dependencies. In the present paper, a layer is considered as a dynamically distinguishable entity. The protection of layers and their dynamic linkage is performed by the system kernel. Any layer can be extracted from the system and replaced by another performing the same set of operations in a different way. This is possible because the content of a layer does not depend on inter-layer relations.

Nowadays we have microcomputers that cannot be said to be classical one- or multi-processor systems. For instance, there are many heterogeneous multi-microprocessor systems based on Multibus. There are also 'intelligent' devices based on microprocessors which can be connected to Multibus. Heterogeneity may be expressed in both physical and functional terms. Such systems are multicomputers rather than multiprocessors.[8] In this paper, a computer in a multicomputer system is called a module. More precisely: a module is a classical one- or multi-processor computer which is serviced by a single kernel. A local operating system is installed on a module. A process does not 'know' to which processor within a module it is allocated.

* This research is supported by the CPBR 8.7 project. The views and conclusions contained in the paper are those of the author.

Computers are built of microprocessors in the same way as one builds things of Lego bricks. There are typical microprocessors, coprocessors, bus drivers, parallel and serial ports, interrupt drivers and many others, which may be combined in a practically unlimited number of ways. The purpose of this paper is to describe a design method for an operating system (not a number of interconnected operating systems) for a heterogeneous hardware environment arranged in a way similar to that in which Lego bricks may be arranged. The layer concept proves to be suitable for such an approach.

Section 2 presents the methodology of inter-module communication: the classical approaches and the new concept. Section 3 covers some implementation considerations. Various possible inter-module semantic dependencies are discussed briefly in Section 4. Section 5 presents the derivation of a module from a physical input/output device. Section 6 provides an example: a distributed operating system MOST. Some analogies are covered in Section 7. The appendix contains a description and skeleton code of the DIVA kernel.

2. SEMANTIC INTERCONNECTION CONCEPTS

2.1 Semantics of classical inter-module connections

In distributed systems, local operating systems are usually interconnected on some abstraction level. This interconnection makes some resources of a module visible to other modules. There are many possible semantics for such interconnections. For example: a standard UNIX system[18] permits the copying of a file between two modules' file systems (the *uucp* command); a system described in Ref. 1 provides a module with access to files owned by other modules; the Ring-Star system[13] lets a program be run on a peripheral module by a request from the central module.

It is, of course, possible to establish interconnection with other abstraction levels using the interconnection facilities provided by the system. The disadvantage of such a methodology is that a higher-level communication protocol uses a lower-level protocol as a base. Moreover, to communicate with a remote process, a process must nest down through the levels to reach the bottom level. Therefore, if there are several levels of the protocol, the communication scheme is quite complex. This scheme of layered protocol building is addressed in Ref. 21. The principle of this methodology is that a given layer X of local system A can talk to the corresponding (on the same level of abstraction) layer Y of local system B. These two layers can communicate by means of calls to the layers immediately below (say, P and Q) in systems A and B. Layer X sees (logically) layer Y of system B, layer Z of system C, etc. These layers constitute a level of abstraction in a distributed system. Layer X talks to other layers by means of calls to layer P. Layer P represents the whole world that is potentially visible to layer X – layers Y, Z, etc. We can understand layer P as a hardware machine for layer X. Layer X sees the neighbouring modules as indivisible local systems: layer Y represents the whole local system B, layer Z local system C, etc. In such a system, layers need not be dynamically distinguishable entities, since local and remote calls are distinct (local connections may be established statically, while remote calls are performed using special layers that implement a layered protocol).

For example, a protocol consisting of two layers established during the implementation of a distributed UNIX-like system is presented in Ref. 2. The lower layer is a network handler treated as a special file (device). The upper layer is a set of inter-process communication routines implemented in terms of the lower layer's *open*, *read*, *write*, *close* and *ioctl* functions.[18]

In a classical approach, modules are autonomous and clearly distinguished sites. The scheme for constructing a distributed operating system is to establish local operating systems first, and then to set up some dependencies between local systems.[1] The methodology of layered protocols was employed to establish closer links between computers distributed over the network. The result is a local area network of distinct and potentially heterogeneous computers.

2.2 Further approach: tightly coupled modules

Ever since the concept of C.mmp was announced, attempts have been made to build a distributed system from scratch rather than by interconnecting local systems. Examples of such systems include Cm*,[20] Micronet[22] and the Transputer.[15] These systems consist of large numbers of anonymous modules, but they present themselves as monolithic computers to their users. A single module is almost unable to work as a separate computer.

In such an environment, the operating system is built as a set of distributed processes communicating by means of a message-passing system. A local module's local operating system consists of two layers. The lower layer constitutes the message-routeing system, the upper one a set of local processes. Examples of systems constructed in this manner are Medusa,[16] Micros[22] and Chorus.[10] If such a system is constructed using Transputer,[15] the lower layer is built into hardware.

The concept of tightly coupled systems is used to distribute multiprocessors, as opposed to linking remote computers closer together by means of layered protocols (see Section 2.1). The common feature of these two approaches is that a process acting in a layer sees the whole underlying system as a monolithic message-passing system. Therefore, neither concept requires the dynamic linkage of layers.

2.3 The new concept

There is a gap between classical systems (those using layered protocols), and tightly coupled systems. In a classical system, it is difficult to establish various kinds of inter-module dependencies at the same time. Typically, all but the highest layers of a protocol are set up to service this highest, semantic level. The new concept moves the classical, potentially heterogeneous system in the direction of a tightly coupled system. The idea is to set up a hierarchy between layers across the distributed system. Each layer (potentially) can access layers from both its local and neighbouring modules. The only limitation is the requirement of partial ordering in inter-layer dependencies. A local system is made visible to other modules' layers as a set of layers rather than as one indivisible unit. In the previous approaches, a layer could talk to the corresponding layer in a neighbouring module. In the new concept, the layers of neighbouring modules are accessible to a certain layer as if they were layers of its local operating system. A layered protocol is replaced by a common mechanism of layer calling. Simple call/return parameter passing is required. This mechanism should be hidden in the lowest level of abstraction in the local operating system, the kernel. An inter-module call may be syntactically identical to a local call, since the layers are dynamically linked. A designer need not even know whether the called layer is local or remote. Moreover, an operation could be implemented locally in one version of the system and remotely in another. The presented interconnection scheme allows the designer to establish different kinds of semantic dependencies between modules at the same time, as will be shown in Section 4.2. The dependencies outlined in Section 2.1 can be found among the dependencies thus established.

In the new concept, the modules are brought much closer together than in classical systems, yet they are still autonomous and clearly distinguished sites and the system may therefore be built out of physically and functionally heterogeneous modules. On the other hand, systems built in a manner following the new concept may cover a much wider set of computer architectures than Cm*,[20] Micronet[22] or Ring-star,[13] which are based on homogeneous distributed modules.

2.4 Process–transaction scheduling equivalence

There are several dualities and equivalences of phenomena in natural science such as wave-particle duality and mass-energy equivalence in physics. In electronics, the product: (amplification × bandwidth of an amplifier) is constant. In distributed systems, there is one more equivalence.

If there are t transactions (calls to operating system from user programs) to be served by m modules, $t > m$, two methods maybe applied: (*a*) scheduling of transactions; (*b*) scheduling of processes.

In case (*a*), in every module there is a set of processes, interconnected by a message-passing system. The transaction migrates between modules, where partial services are performed by local processes. In this approach a pair: (process, message buffer) is a prototype of a local operating system. The internal structure of local systems is of no importance as far as a distribution feature is concerned. Many distributed operating systems are built

31-2

in this manner.[1,2,16,17,22] A type of message-passing system construction is presented in Ref. 10. Principle (*a*) is brought to perfection in the Occam concept.[15]

In case (*b*), each transaction is served by an associated process. This process migrates between modules performing partial services for a transaction. The real concurrency degree of a module (the number of processes implemented in a module) is finite, and usually much less than *t*, therefore processes associated with transactions must be scheduled. In this case scheduling of transactions is replaced by scheduling of processes. A layered structure is appropriate for building a system in this manner. Case (*b*) is based on inter-process synchronization, case (*a*) on process communication. Some aspects of this duality are addressed in Ref. 14.

The advantage of process scheduling is that it does not require a special message-passing system. In addition, a hierarchial structure of layers (*b*) is more deadlock-protected than a parallel structure of processes (*a*). The possibility of a deadlock in message-passing communication (not in resource allocation!) is addressed in Ref. 16.

3. IMPLEMENTATION CONSIDERATIONS

3.1 Classical inter-module communication

Each of the classically interconnected distributed systems[1,13,18] contains a special software unit dedicated to intermodule communication. In the present paper this unit is called a network handler. Of course, there can be several network handler levels, one for each level of communication abstraction. Network handlers existing on a given level may be distinct because they may implement different protocols. A network handler 'sees' the local operating system of a neighbouring module as one indivisible unit of the same abstraction level. A local system does not 'know' whether its neighbour is constructed structurally or not! The overheads of the presented methodology are: network handlers; special processes in callee modules (a caller decides when the communication is to occur, so a callee must be able to receive signals at any time); many additional layer calls and returns (calls to network handlers). Several examples of classically interconnected distributed UNIX-like systems are covered in Refs. 2, 13 and 17.

3.2 New concept-implementation considerations

Since in the new concept local and inter-module layer calls are uniform, the only way to call a layer is to do it by invoking the kernel. The kernel is a unit that handles the trivial operations of an operating system, i.e. inter-layer transitions (calls and returns) and inter-process communication and synchronization. Executing a layer call, the kernel decides whether to enter a local layer or to send a signal to a neighbour module's kernel. In order not to overcomplicate the kernel, which is expected to be a small and simple unit, the interconnection of modules can be performed by a simple handshake mechanism (which will be presented in the appendix).

A local system connected to other modules' local systems by means of network handlers can speak to them in various 'languages' (protocols). The physical interconnection of modules may be as loose as a computer network. Interconnection following the new concept (local systems are treated as sets of layers) is performed by the kernels of modules, so every kernel in the system must speak the common language (handshake). This concept requires a tighter hardware interconnection. Modules must be joined by an interrupt net or an equivalent signal-passing system. They have to operate a common memory or fast communication links for parameter passing. These requirements are fulfilled by stems based on Multibus like μ^*.[8] Environments such as Cm*[20] and Micronet[22] are also acceptable for the new concept (the message-passing system may be used as a basis for call/return parameter passing).

4. SEMANTIC INTERCONNECTIONS OF LOCAL OPERATING SYSTEMS

4.1 Decomposing a local system into layers

Typically a local operating system is specified as a program written in a structured programming language. The structure of such a system is natural, i.e. it depends on sets of operating system features corresponding to specific implementation areas. A typical set of layers in a layer-structured local system is:
- device handlers,
- a file system,
- a niche supervisor,
- niches.

A niche is treated statically as 'a place to run programs'. The term 'niche' is taken from ecology: it denotes an area which may be occupied by anyone, but once occupied, cannot be occupied by anyone else. Niches are occupied by user tasks, i.e. the dynamic entities corresponding to the execution of user programs. A supervisor implements a system interface for programs. Operations like 'run a program', 'terminate a program', 'send a signal to a program' are implemented in this layer. A file system manages the address space of the mass-storage devices by means of an information structure. Device handlers unify access to devices and give them some non-hardware features.

The above hierarchy (with some modifications) covers such different application areas as the SOM-5 embedded hierarchical database,[6] the Toronto University UNIX-like TUNIS system[9] and a family of operating systems described in Ref. 1.

4.2 Semantic interconnections in distributed systems

A question could be asked: why not establish all the kinds of interconnection presented in Section 2.1, i.e. the ability to access a file, to copy a file and to run a program (in a neighbour module)? The new concept makes it possible to set up these, and many other inter-module dependencies, all at the same time. Fig. 1 shows various possible inter-module connections and their meanings. Inter-module dependencies outlined in Section 2.1 are among those presented in Fig. 1: C,[18] D,[1] B[13]. Interconnection in terms of the new concept is semantic because it allows the designer to establish semantic dependencies between layers of local systems.

5. THE INPUT/OUTPUT DEVICE AS A MODULE

The concept of semantically interconnected modules

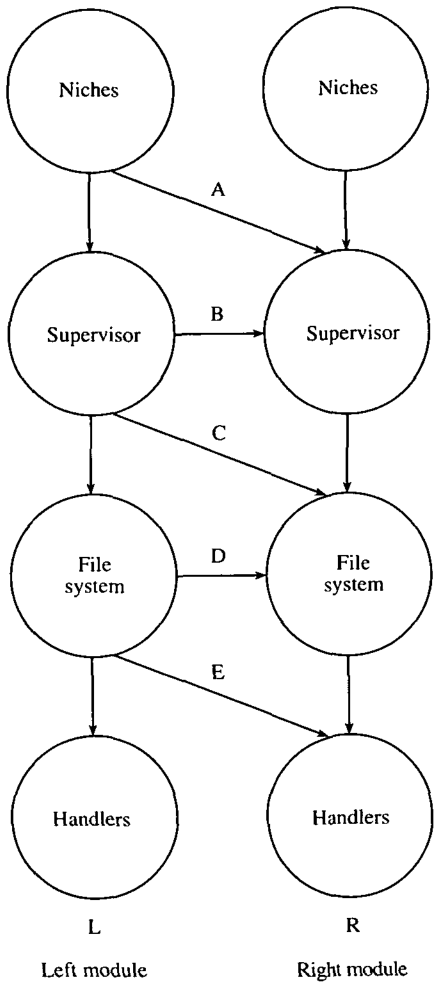

Figure 1. Various possible inter-module connections. A, the L niche is served by the R supervisor. B, the program in the L niche can be transferred to the R niche. C, the L module niches see two independent file systems, L and R. D, the R file system is a part of the L file system. E, the R devices are immediately visible to the L file system. The connections (kernel activities) are represented by arrows between layers.

may be derived from making a physical device more and more intelligent.

The first step is a simple character-oriented device which can read or write one character, and which signals an interrupt when the transmission is completed. Such a device may be treated as a defective layer in the system: it needs a 'push' from the caller after each character is transmitted, and the call parameters are different from the parameters of an ordinary layer calling.

The second step is a device driven by a direct memory access unit. It can signal the caller when the whole operation (a transmission of a sequence of characters) is completed. It is a less defective layer, since it does not need a 'push' after every elementary transmission. Yet the layer (device) call parameters are still non-standard in such a system.

The third step is a device supported by its own internal processor. It may be programmed to satisfy the standard layer-calling requirements in the system. So the device is a module in the distributed system (E in Fig. 1), and the new concept system may use it as a single-layered local system.

The fourth step is a hierarchy of layers established in the device's module. These layers define sets of operations on some abstraction levels that allow the programmer to use the device as a virtual device. The hierarchy constitutes a multi-layered module.

The last step is a layer-structured local operating system provided with one or several niches. Such a system may perform its own activity and constitutes an autonomous module in the distributed environment.

6. AN EXAMPLE: THE 'MOST' OPERATING SYSTEM

6.1 Overview

The MOST operating system has been implemented in MERA-system Laboratories ('most' means 'bridge' in Polish). The semantic interconnections concept is used in this system. MOST is a UNIX-like system in a distributed environment.

In the MOST system a layer is considered as a dynamically distinguishable and protected entity, as described in Refs. 7 and 11. This has several advantages, for example: an error in a layer does not propagate to other layers; a layer may be replaced by another, implementing the same set of operations in a different manner (such a replacement does not require the recompilation of the whole system); user tasks can operate using various virtual machines related to the subset of layers they see.[7]

A layer can access other layers (enjoys a privilege to call them) if they are passed to it as actual parameters during the system generation. If a parameter is not a local layer, in a callee module containing this layer there are special processes dedicated to handle the calls from the neighbours, as described for the classical concept. Yet these agent-processes are hidden in a kernel, so they are invisible on the abstraction level of a structured programming language. From the designer's point of view, a process moves from a module to a neighbour.

6.2 Notation

As a notation, the structured language PROSO (modified Concurrent PASCAL[4]) proposed by Chrobot[7] is used in the design of MOST. The differences between PROSO and Concurrent PASCAL are:

- *queue* variables replaced by *delay-resume* primitives;
- layer construct, proposed in Ref. 5; the semantics of this construct are extended to meet the requirements mentioned in the previous section;
- coprocedure instead of sequential process; a coprocedure is a procedure which, when called, continues executing concurrently with its caller (and never terminates);
- local system construct, which embraces layers of a module and makes layers owned by neighbours visible to local layers (lets the programmer pass them as parameters to local layers); the externally visible layer is marked **entry**.

The latter feature prevents the operating system being split into two separate areas: the operating system itself and its user tasks. A user task is a system layer consisting of a single coprocedure (in the case of a niche for sequential program, of course).

6.3 Structure of local MOST

The structure of local MOST is presented in Algorithm 1. It corresponds to any of the systems L and R in Fig. 1. The algorithm includes the core-style implementation of the *exec* system call, as an example of decoding an operation of a layer into operations of lower layers. The action performed by the initial statement of a *niche* is to start a process acting as a user task. This process (coprocedure *task*) invokes the *exec* system call to run the *shell* – a UNIX command interpreter. The *shell* is contained in a file named '*sh*'. The *exec* system call is decoded by the supervisor into the *file system*'s operations. First, the file containing a program whose name is supplied as a parameter of the *exec* command is opened. Then, the file content is read into memory, and the file is closed. In the *file system* the operations *open* and *read* are implemented as sequences of calls to the *block handler* to read a number of disc sectors. *close* is an empty operation in this case. To illustrate that a process need not be started in the niche-layer, the *releaser* coprocedure in the *file system* is shown. This process frees sectors when a file is unlinked.

6.4 Semantic dependencies in distributed MOST

MOST has been implemented in a distributed environment consisting of various Intel family microprocessors attached to Multibus. It is not a true multiprocessor because of its heterogeneity. A minimum hardware configuration consists of one 8080 module and one 8086 module. The 8080-based computer is a slow *text module*, dedicated to the preparation of source documents. The 8086-based computer is a fast-calculating *main module* performing such activities as compilation and numerical computation. Algorithm 2 constitutes the skeleton code for the system. The semantic dependencies are as follows.

(1) The text module's shell may be moved to a niche in the main module (B in Fig. 1). When working on its own module (autonomous mode), the shell applies a limited set of UNIX commands (for instance, there is no background processing or pipes). This is because the text module contains only one niche. When the shell is moved to the main module's niche (non-autonomous mode), it operates as an ordinary UNIX program. Then it can create a family of programs using the *fork* system call.

(2) In the autonomous mode, a user program sees the text module's file system (B in Fig. 2). In the non-autonomous mode, the program sees the main module's file system and the text module's file system as a part of the main module's file system (A + B in Fig. 2, D in Fig. 1), while other main module programs (not transferred from the text module) see the main module's file system only (A in Fig. 2). The text module's display remains an input/output device of the shell after its move to the main module's niche.

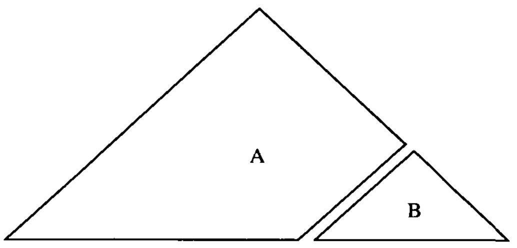

Figure 2. Main module's (A) and text module's (B) file system trees.

The extended (and, possibly, not final) version of the MOST system consists of a number of text modules and two modules adopting the device handling functions of the main module (E in Fig. 1). They are the *disk driver* and *multiplexer*. In this version, the main module is replicated and constitutes a classical multiprocessor (not a multicomputer!).

The UNIX-like system will not be the only application, because MOST is a scheme of operating system design rather than an instance of an operating system. Various kinds of systems will be built as MOST. Of course, all of them will use the same kernel, and a majority will include the layers of device handlers.

7. ANALOGIES

A task force of Medusa[16] installed on a cluster of Cm*[20] may be considered as a local system installed on a module. The main difference (in addition to those described in Section 2) is that task forces implement operating system utilities, such as file systems or memory managers, rather than autonomous local systems. An inter-cluster communication failure, a run-time extraction or a destruction of a cluster may crash the entire system. MOST contains similar utility systems (multiplexer and disk driver), but it also contains autonomous modules (main module, text module). The latter are still able to work locally in case of a communication failure, or the removal or destruction of a neighbour module.

This facility brings the new concept closer to the system described in Ref. 1, although in the latter there is only one interconnection level (data ports).

Each individual module of Micronet[22] or Cm*[20] can be considered to be a basis for a single-layered local system. Specifically, there is an analogy between the concept of interconnection of heterogeneous modules and an early version of Micronet consisting of a number of Micros[22] and a number of UCSD Pascal local systems. Modules in Micros and Medusa are interconnected at the task level. Higher-level protocols between tasks are established using the basic protocol (messages) supported by local operating systems, as described in Section 1.

Since the kernel of a MOST-like system is small and simple, a semantically interconnected operating system is static and non-reconfigurable on the operating system level. Yet a user task running in a niche can be transferred around the modules if there are semantic interconnections between local system supervisors. Thus a system may act like Micros or Medusa above the supervisor level.

The static image of the new concept is dedicated to architectures of several or tens of functionally distinct, specialised modules rather than to hundreds or thousands of homogeneous and anonymous modules like Cm*, Micronet or μ*.[8] The new concept does not require special hardware to perform physical interconnection like Slocals and Kmaps of Cm* or front-end computers of Micronet.

System schemes like those of Medusa and Micros may be considered as degenerate cases of the semantic interconnections principle.

8. CONCLUSIONS

To establish minicomputers and mainframes, the work of hundreds of people, large amounts of money up to millions of dollars, and years of time were needed. A microcomputer can be built by one person, for a few hundred dollars during free afternoons. This is possible because a microcomputer is constructed in a brick-style, like a house built of Lego bricks. Application software can be established in a similar way, using software packages such as graphics, databases, spreadsheets, window managers, etc. An operating system can be built in a brick-style by means of the structured programming methods. And the concept of semantic interconnections of local operating systems permits a distributed operating system to be constructed in a similar manner.

This concept enables the designer to remove all network handlers from the classical system, and to replace them with the common interconnection strategy hidden in the kernel. Yet this concept lets him preserve **some** network handlers, namely handlers of 'unintelligent devices' and of non-standard modules in the system, 'speaking' a language different from a simple handshake.

A structured approach to operating system design allows one to establish a family of systems.[11] Every layer works regardless of the activities of higher layers, so a number of systems may be constructed using the given layer operation set as primitives. The layer simply defines a virtual machine on a certain abstraction level. Therefore, it may be used for several purposes, depending on the set of layers installed above it.

The concept of semantic interconnection extends this facility to cover distributed applications, since the level of abstraction defined by a layer may be applied as a basis for both local and remote higher layers. The new structuring method partly bridges a gap between the classical approach of layered protocols and the idea of tightly coupled systems. A distributed structured operating system, constructed using semantic interconnections of modules, is like a Cartesian product of processes, layers and modules (Fig. 3).

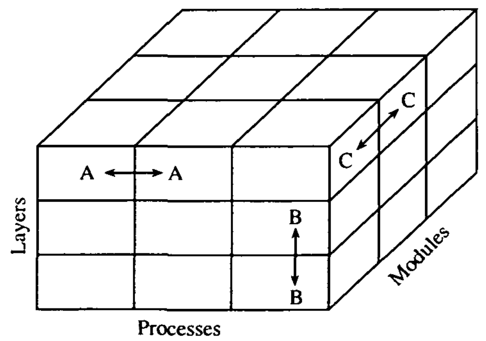

Figure 3. A Cartesian-like product of processes, layers and modules: A, interprocess communication and synchronisation (concurrent system); B, layer calling (structured system); C, intermodule layer calling (distributed system).

The MOST operating system is under development in the MERA-System laboratories, concurrently for the main module and the text module.

Acknowledgements

The inspiration for this paper is due to Staszek Chrobot. He also helped to formulate the principle of process/transaction scheduling equivalence. Professor Wladyslaw M. Turski helped to produce the final form of the paper. The author's collaborators in MERA-System laboratories and Marek Gondzio gave important advice.

APPENDIX – THE 'DIVA' KERNEL

The IVA kernel of the local MOST operating system supports the primitives of the PROSO programming language. It performs the following generation functions:

(1) adding layers to the system;

(2) starting processes (coprocedure calling); this can be done only during the system generation, because the total number of processes is the main invariant of the IVA kernel;

and the following run-time functions:

(3) inter-layer communication (layers are dynamically protected by the kernel, and the only way to call a layer is by invoking the kernel);

(4) synchronisation of processes in monitor layers by delay–resume operations (it could be done using Hoare's conditions[12] or Brinch Hansen's queue variables as well);

(5) process-interrupt synchronisation;

(6) traps (exceptions) handling.

The semantic interconnection of local systems is performed by their kernels, instances of DIVA (distributed IVA). The kernel of a calling local system must:

(7) distinguish local calls from calls to neighbouring modules;

(8) implement a protocol for calling neighbours;

(9) handle returns from calls.

The kernel of a callee local system must:

(10) handle requests (calls) from its neighbouring modules;

(11) redirect requests to agent-processes in the layers which have been called;

(12) distinguish between local returns and returns from calls from neighbours;

(13) implement a protocol for returning from remote calls.

Since every module may be both a caller and a callee, its kernel must perform all of the functions 1–13. Algorithm 3 presents a formal specification of the DIVA kernel. The representative set of the local kernel entry operations is:

- for inter-layer communication, *call* a layer and *return* to the calling layer,
- for synchronisation of processes, *delay* and *resume*.

These operations are the only ones that change a process state. The following states of a process in a layer are allowed:

- *active*, occupying a monitor layer critical region;
- *ready*, waiting for a monitor layer critical region or running in a non-monitor layer,
- *delayed*, after execution of *delay*,
- *calling*, after execution of *call* a layer.

The data structures that support these kernel operations are *layer descriptors* and *process descriptors*. The Cartesian-like product of these two vectors determines *layer state records*, which describe the status of processes in layers. A *layer state record* holds the contents of processor registers of a process in the layer.

Fig. 5 shows an instantaneous state of the kernel data structures for a frozen state of a sample system consisting

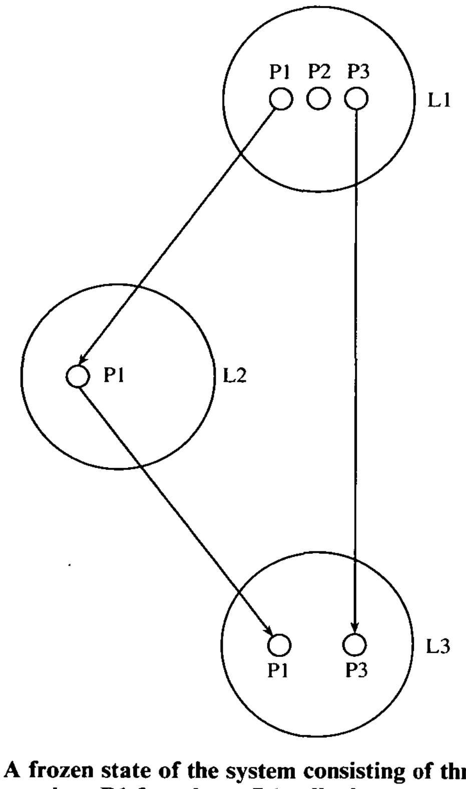

Figure 4. A frozen state of the system consisting of three layers; the coprocedure P1 from layer L1 calls the entry procedure of layer L2, and the latter calls the entry procedure of layer L3; the coprocedure P2 is running in the layer L1; and the coprocedure P3 from layer L1 calls the entry procedure in layer L3.

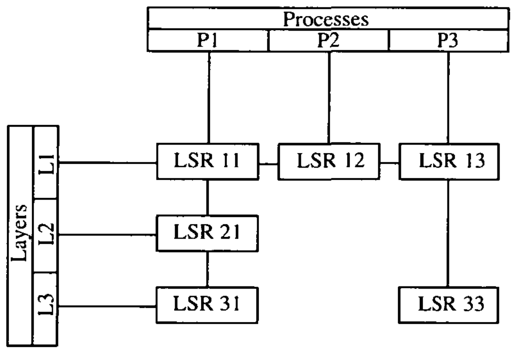

Figure 5. The kernel data structures for the frozen state of the system shown in Fig. 4.

of three processes and three layers, Fig. 4. In a process dimension (vertical in Fig. 5), the layer state records are organised in stacks. After a process has called a layer, a *layer state record* is pushed onto the stack. It is popped when a process returns to a calling layer. In the initial state each process has only one *layer state record* on its stack. This is a record describing the status of the process in the layer where it was started. In a layer dimension (horizontal in Fig. 5), the *layer state records* are joined to *layer descriptors* through queue links. A queue collects all processes that are in the same state in a layer.

Inter-module communication is implemented as follows:

- Each kernel is supplied with a parameter *own* which is a unique identifier for its module in the system.
- There is a table *home*, which for every layer 'known' to the local system contains the identifier of this module in the system to which a layer belongs. So the kernel contains module identifiers for all its local layers and for all layers in neighbouring modules, accessible to its local layers. If *home* [*layer*] = *own*, then the layer is local to the module; otherwise it belongs to a module identified by *home* [*layer*]. For all layers, for which *home* [*layer*] $\neq$ *own*, a table *orig* contains names of layers in their original modules.
- Inter-module call and return are performed by a handshake mechanism. Each module has its own area in a common memory. This area is called a *window*. A module owning the *window* can read and write to it, but other modules can only read. If the kernel performs an inter-module *call* or *return* operation, it writes the *call parameters* into its own *window* and causes an interrupt in the neighbouring module (or sends another kind of signal). Then it waits for a reply.
- After receiving an interrupt signal from a neighbour, the kernel copies the *call parameters* from the neighbour's window to its own memory area, and then sends a reply for the interrupt. The *call parameters* are received by the callee while executing the inter-module *call* operation, and by the caller while executing the inter-module *return* operation.
- There is a special process (or processes) called an *agent* which is dedicated to handle calls from the neighbouring modules. In the called layer, an *agent* behaves just like an ordinary process.
- If a process invoking the *return* operation is an *agent*, the operation is an inter-module *return*.

The implementation of generation of the system (adding layers and starting processes) is omitted from Algorithm 3. The handshake mechanism is performed by pairs of routines: operation *call* with interrupt service routine *arrive*, and operation *return* with interrupt service routine *leave*. The **interrupt** construct defines an interrupt service routine which is treated as an ordinary kernel entry; **interrupts** are mutually exclusive with other entries. The kernel is a monitor[12] as far as the mutual exclusion of entries is concerned. As there is no internal synchronisation mechanism in the kernel, only a busy form of waiting is allowed. This facility is suitable for the synchronisation of kernels of distinct modules during the handshaking.

Note that the inter-module versions of the *call* and *return* operations are split in a very symmetric manner. The caller invokes *call* and *leave* operations in the calling *layer descriptor*, while the callee invokes the *enter* and *return* operations in the called *layer descriptor*.

The software interface (handshake) presented in Algorithm 3 is not tight. The callee kernel can answer a signal and start an *agent*, while the caller may decide that a call has been unsuccessful. The algorithm can be made tighter, for instance by the caller sending a clearing signal when it decides that a call fails.

REFERENCES

1. E. Akkoyunlu, A. J. Bernstein and R. Schantz, An operating system for a network environment. *Proceedings of a Symposium on Computer Communication Networks and Teletraffic, Polytechnic Institute of Brooklyn, Brooklyn, NY*, **22**, 529–538 (1972).
2. G. S. Blair, J. A. Mariani and W. D. Shepherd, A practical extension to UNIX for interprocess communication. *Software – Practice and Experience*, **13** (1), 45–58 (1983).
3. P. Brinch Hansen, *Operating System Principles*. Prentice-Hall, Inc., Englewood Cliffs, NJ (1973).
4. P. Brinch Hansen, *Concurrent Pascal, a Programming Language for Operating System Design*. Information Science Technical Report no. 10, California Institute of Technology (1974).
5. S. Chrobot, Layer – a language construction for concurrent structural program design. *Information Processing Letters*, **4** (5), 113–117 (1976).
6. S. Chrobot, Operating system SOM-51 for the MERA-400 minicomputer. *Informatyka* (12), 15–18 (1980) (in Polish).
7. S. Chrobot, *Structured Operating Systems Design*. Polish Cybernetic Society, Warsaw, Poland (1981) (in Polish).
8. D. Del Corso, An experimental multimicroprocessor system with improved internal communication facilities. *Euromicro Journal* 4, 326–332 (1978).
9. P. A. Ewens, R. C. Holt, M. J. Funkenhauser and D. R. Blythe, *The TUNIS Report: Design of a UNIX-compatible Operating System*. Technical Report CSRI-176. University of Toronto, Ontario, Canada (1986).
10. M. Guillemont, The CHORUS distributed operating system: design and implementation. *Proceedings of International Symposium on Local Computer Networks, Florence, Italy* (1982).
11. A. N. Habermann, L. Flon and L. Cooprider, Modularization and hierarchy in a family of operating systems. *Communications of the ACM* **19** (5), 266–272 (1976).
12. C. A. R. Hoare, Monitors: an operating system structuring concept. *Communications of the ACM* **17** (10), 549–557 (1974).
13. A. I. Karshmer, D. J. Depree and J. Phelan, The New Mexico State University Ring-Star system: a distributed UNIX environment. *Software – Practice and Experience* **13** (12), 1157–1168 (1983).
14. H. E. Lauer and R. M. Needham, On the duality of operating system structures. *Operating Systems Review* **13** (2), 3–19 (1979).
15. D. May and R. Shepherd, Occam and the Transputer, *Proceedings of IFIP 10.3 Workshop on Hardware Supported Implementation of Concurrent Languages in Distributed Systems, Bristol* (1984).
16. J. K. Ousterhout, D. A. Scelza and P. S. Sindhu, Medusa: an experiment in distributed operating system structure. *Communications of the ACM* **23** (2), 92–105 (1980).
17. R. F. Rashid, *An Inter-process Communication Facility for UNIX, CMU-CS-80-124*. Department of Computer Science, Carnegie-Mellon University, Pittsburgh (1980).
18. D. M. Ritchie and K. Thompson, The UNIX time-sharing system. *Communications of the ACM* **17** (7), 365–375 (1974); also *Bell System Technical Journal* **57** (6.2), 1905–1929 (1978).
19. A. C. Shaw, *The Logical Design of Operating Systems*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1974).
20. R. J. Swan, S. H. Fuller and D. P. Siewiorek, Cm* – a modular multimicroprocessor. *AFIPS Conference Proceedings* **46** 637–644 AFIPS Press, Montalve, NJ (1977).
21. M. Witt, An introduction to layered protocols. *Byte* **8** (9), 385–398 (1983).
22. L. D. Wittie and A. M. van Tilborg, Micros, a distributed operating system for Micronet, a reconfigurable network computer. *IEEE Transactions on Computers* **C-29** (12), 1133–1144 (1980).

ALGORITHM 1. The local MOST operating system

```
type local_MOST = local system ;
  type char_handler = layer monitor ;
    procedure entry read (...);
      begin ... end ;
    procedure entry write (...);
      begin ... end ;
    begin ... end ;
  type block_handler = layer monitor ;
    procedure entry read (...);
      begin ... end ;
    procedure entry write (...);
      begin ... end ;
    begin ... end ;
  type file_system = layer monitor
      (ch:char_handler;bh:block_handler);
    procedure entry open (...);
      begin ... bh.read (...) ... end ;
    procedure entry read (...);
      begin ... bh.read (...) ... end ;
    procedure entry close (...);
      begin ... end ;
    procedure entry ...
    ...
    coprocedure releaser (...);
      begin ... end ;
    begin ... releaser (...) ... end ;
  type supervisor = layer monitor (fs:file_system);
    procedure entry fork (...);
      begin ... end ;
    procedure entry exec (a,...);
      begin ...
        fs.open (a);
        fs.read (contents);
        fs.close (a);
        ...
      end ;
    procedure entry exit (...);
      begin ... end ;
    procedure entry wait (...);
      begin ... end ;
    procedure entry ...
    ...
    begin ... end ;
  type niche = layer class (s:supervisor);
    coprocedure task (...);
      begin ... s.exec (sh) ... end ;
    begin ... task (...) ... end ;
  var
    console : char_handler ;
    disk : block_handler ;
    fs : file_system ;
    s : supervisor ;
    p0,p1 : niche ;
  begin
    init console, disk, fs(console, disk),
        s(fs), p0(s), p1(s)
  end.
```

ALGORITHM 2. The distributed MOST operating system

```
type MOST = system ;
  type text_module = local system
    (mms : main_module.supervisor);
    type char_handler = layer monitor ;
    begin ... end ;
  type block_handler = layer monitor ;
    begin ... end ;
  type file_system = layer monitor
  ch : char_handler; bh : block_handler);
    begin ... end ;
    type supervisor = layer monitor (fs : file_system;
        mms : main_module.supervisor);
      begin ... end ;
    type niche = layer class (s : supervisor);
      begin ... end ;
    var
      console : char_handler ;
      disk : block_handler ;
      entry fs : file_system ;
      s : supervisor ;
      p : niche ;
    begin
      init console, disk, fs(console,disk),
        s(fs,mms), p(s)
    end ;
  type main_module = local system
      (tmfs : text_module.file_system);
    type char_handler = layer monitor ;
      begin ... end ;
    type block_handler = layer monitor ;
      begin ... end ;
    type file_system = layer monitor
        (ch : char_handler;
        bh : block_handler;
        tmfs : text_module.file_system);
      begin ... end ;
    type supervisor = layer monitor (fs : file_system);
      begin ... end ;
    type niche = layer class (s : supervisor);
      begin ... end ;
    var
      consoles : char_handler ;
      disks : block_handler;
      fs : file_system;
      entry s : supervisor;
      p0,p1,...,pn : niche;
    begin
      init consoles, disks, fs(consoles,disks,tmfs),
        s(fs), p0(s), p1(s), ... , pn(s)
    end ;
  var
      tm : text_module;
      mm : main_module;
    begin
      init tm (mm.s), mm(tm.fs)
    end.
```

ALGORITHM 3. The distributed kernel program – parts marked ... are not of interest

```
type DIVA = monitor (own : module_number);
  const none = 0 ;
  const max_p = ... ;
  const max_l = ... ;
  const max_s = max_l ;
  const max_ag = ... ;
  const hand_shake_timeout = ... ;
  type process_number = 0..max_p ;
  type agent_number = 1..max_ag ;
  type layer_number = 0..max_l ;
  type stack = 0..max_s ;
  type call_parameters = ... ;
  type agents = class ;
    var list : {a structure} of
      record
        p : process_number;
        m : module_number;
        ag : agent_number
      end ;
    procedure entry insert
        (p : process_number; m : module_number; ag :
        agent_number);
      begin
        {add '(p,m,ag)' at the end of list}
      end ;
    procedure entry remove (m : module_number;
        ag : agent_number; var p : process_number);
      begin
        {remove a node containing 'm' and 'ag'
          from the list, and fill the reply 'p' parameter}
      end ;
  begin ... end ; {agents}
  type window = class (a : address);
    procedure entry fill (par : call_parameters);
      begin
        {send 'par' into the window of address 'a'}
      end ;
    procedure entry get (var par : call_parameters);
      begin
        {get 'par' from the window of address 'a'}
      end ;
    begin ... end ; {window}
  type windows = array [module_number] of window;
  type layer_state_record = class ;
    type registers = ... ;
    var reg : registers ;
    var lay : layer_number ;
    procedure entry fill (par : call_parameters;
        l : layer_number);
      begin
        lay: = l;
        {allocate a stack for the calling process in 'lay'}
        {fill the bottom of stack with 'par'}
        {set initial value for 'reg'}
      end ;
    procedure entry remove (var par : call_parameters);
      begin
        {deallocate the stack}
        {return reply parameters to 'par'}
      end ;
    procedure entry set (var par : call_parameters);
      begin
        {get 'par' from the top of stack in 'lay'}
      end ;
    procedure entry put (part : call_parameters; var l :
    layer_number);
      begin
        {put 'par' on the top of stack in 'lay'}; l : = lay
      end ;
    procedure entry start ;
      begin
        {send 'reg' to the processor's registers}
      end ;
  begin ... end ; {layer_state_record}
  type process_descriptor = class (own : module_number; w :
    windows);
    var top : stack;
    var state : array [stack] of layer_state_record;
    var cm : module_number ;
    procedure entry inter_module_call (m : module_number);
      var par : call_parameters;
      begin
        state [top].get(par);
        w [own].fill(par)
      end ;
    procedure entry agent_start (m : module_number; l :
        layer_number);
    var par : call_parameters;
    begin
      cm: = m;
      w [cm].get(par);
      top: = 0;
      state [top].fill(par,l)
    end ;
  procedure entry push (l : layer_number);
    var par : call_parameters;
    begin
      state [top].get(par);
      top: = top + 1;
      state [top].fill (par,l)
    end ;
  procedure entry pop (var l : layer_number; var m :
      module_number);
    var par : call_parameters;
    begin
      state [top].remove(par);
      if top = 0 then
      begin
        m: = cm;
        w [own].fill(par)
      end
      else begin
        m: = own;
        top: = top − 1;
        state [top].put(par,l)
      end
    end ;
  procedure entry inter_module_return (m : module_number;
      var l : layer_number);
    var par : call_parameters ;
    begin
      w [m].get(par);
      state [top].put(par,l)
    end ;
  begin ... end ; {process_descriptor}
type queue = class ;
  procedure entry insert (p : process_number);
    begin
      {add 'p' at the end of queue}
    end ;
  procedure entry get (var p : process_number);
    begin
      {get first in the queue into 'p' and remove it from the
      queue; if the queue is empty then 'p' = none}
    end ;
  procedure entry remove (var p : process_number);
    begin
      {remove 'p' from the queue and return 'p'; if 'p' is not in
      the queue, 'p' = none}
    end ;
  function entry first : process_number ;
    begin
      {'first' = first in the queue;
      if the queue is empty then 'first' = none}
    end ;
  begin ... end ; {queue}
type layer_descriptor = class ;
  var kind : (mon,cls);
  var active : process_number ;
  var ready,delayed,calling : queue ;
  procedure entry delay ;
    begin
      if kind = mon then
      begin
        delayed.insert(active);
        ready.get(active);
      end
    end ;
procedure entry resume (p : process_number);
  begin
    if kind = mon then
    begin
      delayed.remove(p);
      if p ≠ none then
      begin
        ready.insert(active);
        active: = p
      end
    end
  end ;
procedure entry call (p : process_number);
  begin
    calling.insert(p);
    if kind = mon then ready.get(active)
    else ready.remove(p)
  end ;
procedure entry leave (p : process_number);
  begin
    calling.remove(p);
    if p ≠ none then
      if (kind = mon) and (active = none) then
        active: = p
      else ready.insert(p)
  end ;
procedure entry enter (p : process_number);
  begin
    if (kind = mon) and (active = none) then
      active: = p
    else ready.insert(p)
  end ;
procedure entry return (p : process_number);
  begin
    if kind = mon then ready.get(active)
    else ready.remove(p)
  end ;
function entry choose : process_number ;
  begin
    if kind = mon then choose: = active
    else choose: = ready.first
  end ;
begin ... end ; {layer_descriptor}
var processes : array [1..max_p] of process_descriptor;
var layers : array [layer_number] of layer_descriptor;
var free_agents : array [agent_number] of boolean;
var active_calls : agents;
var home : array [layer_number] of module_number;
var orig : array [layer_number] of layer_number;
var w : windows;
var current_p : process_number;
var current_l : layer_number;
procedure choose ;
  var i : layer_number;
  var p : process_number;
  begin
    for i in layer_number do
      if home [i] = own then
      begin
        p: = layers [i].choose;
        if p ≠ none then exit
      end ;
    current_l: = i;
    current_p: = p;
    processes [current_p].start
  end ;
procedure entry CALL (l : layer_number);
  var reply : agent_number;
  var i : 0..hand_shake_timeout;
  begin
    if home [l] = own then
```

```
   begin
     layers [current_l].call (current_p);
     processes [current_p].push(l);
     layers [l].enter (current_p)
   end
   else begin
     processes [current_p].inter_module_call (home[l]);
     {send an interrupting signal 'ARRIVE'
     to the 'home [l]' module,
     with parameter 'orig [l]'};
     i: = 0;
     repeat
       {read 'reply' from 'home [l]'};
       i: = i + 1
     until (i = hand_shake_timeout) or (reply ≠ none);
     if reply = none then {ERROR}
     else begin
       layers [current_l].call (current_p);
       active_calls.insert(current_p, home[l], reply)
     end
   end ;
   choose
  end ;
interrupt ARRIVE
   (m : module_number;
   l : layer_number);
  var ag : agent_number;
  begin
   {get a free agent 'ag' from the table 'free_agents'};
   {send reply 'ag' to the module 'm'};
   process [ag].agent_start(m,l);
   layers [l].enter(ag);
   choose
  end ;
procedure entry RETURN ;
  var l : layer_number;
  var m : module_number;
  var reply : agent_number;
  var i : 0..hand_shake_timeout;
  begin
   layers [current_l].return(current_p);
   processes [current_p].pop(l,m);
   if m = own then layers [l].leave(current_p)
   else begin
     {send an interrupting signal 'LEAVE' to the module
       with parameter 'current_p'};
     i: = 0;
     repeat
       {read 'reply' from 'm'};
       i: = i + 1
     until (i = hand_shake_timeout) or (reply ≠ none);
     free_agents [current_p]: = true
   end ;
   choose
  end ;
interrupt LEAVE (m : module_number;
   ag : agent_number);
  var p : process_number;
  var l : layer_number;
  begin
   active_calls_remove(m,ag,p);
   processes [p].inter_module_return(m,l);
   {send reply other than 'none' to the module 'm'};
   layers [l].leave(p);
   choose
  end ;
procedure entry DELAY;
  begin
   layers [current_l].delay;
   choose
  end ;
procedure entry RESUME (p : process_number);
  begin
   layers [current_l].resume(p);
   choose
  end ;
begin ... end.
```

Announcements

5–8 January 1992

Second International Symposium on Artificial Intelligence and Mathematics, Fort Lauderdale, Florida

Approach of the Symposium

The International Symposium on Artificial Intelligence and Mathematics is the second of a biennial series featuring applications of mathematics in artificial intelligence as well as artificial intelligence techniques and results in mathematics. There has always been a strong relationship between the two disciplines; however, the contact between practitioners of each has been limited, partly by the lack of a forum in which the relationship could grow and flourish. This symposium represents a step towards improving contacts and promoting cross-fertilisation between the two areas. The editorial board of the *Annals of Mathematics and Artificial Intelligence* serves as the permanent organising committee for the series of Symposia.

Sponsors

The symposium is sponsored by Florida Atlantic University and IJCAII. Additional funding is pending. Partial travel subsidies may be available to young researchers.

Information

Contact Frederick Hoffman, Florida Atlantic University, Department of Mathematics, PO Box 3091, Boca Raton, FL 33431, USA (E-mail: hoffman@acc.fau.edu or hoffman@fauvax.bitnet) for further information and to receive future announcements.

25 and 26 May 1992

Conference on Eiffel, Damstadt, Germany.

Organiser German chapter of the ACM e.V./Gesellschaft für Informatik e.V, FA PS and PE.

Information: Prof. Dr Hans-Jürgen Hoffmann, University at Darmstadt, Department of Computer Science, FG PÜ, Alexanderstr. 10, D-6100 Darmstadt, Germany. Tel: +49 6151 163410. Fax: +49 6151 165550. E-mail: EARN/BITNET: XIPHJHO@DDATHD21.

29 June to 3 July 1992

Second International Conference on the Mathematics of Program Construction, Oxford, UK

Call for Papers

The second International Conference on the Mathematics of Program Construction is to continue the theme set by the first – the use of crisp, clear mathematics in the discovery of algorithms. But recent developments within this philosophy have shown that the approach is remarkably diverse, applying not only to sequential programs but also to parallel or concurrent applications, real-time and reactive systems, and even designs realised directly in hardware. In this the second conference, therefore, it is hoped to take advantage of the ever-widening impact of precise mathematical methods in program development.

The second conference is to be held at St Catherine's College which, while situated only a few minutes' walk from the city centre, lies on Oxford's famous 'punting' river (flat-bottomed boats propelled by poles) in bucolic surroundings.

Attendance will be limited to 150–200. Submissions of papers (5 copies) should be sent to Carroll Morgan, clearly marked *MPC*, by 20 January 1992; acceptance will be notified from 9 March; and final camera-ready copy would in that case be due by 8 June for distribution at the conference and publication.

Although there is no page limit on submissions, the usual advantages of brevity are strongly commended.

Carroll Morgan and Jim Woodcock, Programming Research Group, 11 Keble Road, Oxford OX1 3QD, UK. Tel: +44 865 273840. E-mail: carroll@prg.ox.ac.uk; jimw@prg.ox.ac.uk.