ON UNIFYING MODULE INTERFACES

Bjarne Stroustrup

The Computer Laboratory
University of Cambridge
England

Abstract

This paper presents the outline of a uniform interface mechanism for activating different kinds of modules, e.g. processes, monitors, and procedures. The usefulness of such an interface in the design of modules and in the tuning of a system is discussed. The overheads involved in using it are explained, together with some implications that its general use has on the system structure.

Introduction

The low price of hardware has lead to the proliferation of hardware architectures and in particular opened new possibilities for using special hardware, extra processors, extra front-end computers, and even complete independent systems to enhance a basic system by speeding up specific functions or just to take some load off the main part of the system.

The exploitation of such semi-distributed hardware systems by migration of software modules from the main part of the system to supporting hardware is usually greatly hampered by the lack of a precise definition of modules and their dependencies, so that the movement of a module involves changes both to it and to all modules which activate it directly. Even simple multiprocessor systems present problems of how to organize software to exploit the extra parallellism offered by the hardware, because it is difficult to vary the amount of potential parallellism offered by the basic software to suit all the possible numbers of processors. In general one would expect that the decision of whether a given module should be implemented as a process or as a procedure will depend on the hardware configuration (and the job load).

A uniform interface to all modules provides a possiblity for varying the implementation of modules to suit a wide range of situations, e.g. a uniform interface to both local and remote processes (i.e. processes running on this machine and processes running on some other machine) <Wald72, Farb73> allows one to write a network interface process which runs unchanged either on the main computer or on a closely attached "NIP" computer. A uniform interface to processes and to modules activated in a procedure-like mannner (from now on simply called procedures) allows one to tune a system by providing more or less processes to suit the actual number of processors, e.g. a directory manager module implemented as a procedure on a uni-processor installation could be made into a process after the installation has been enhanced with a second processor.

It must be kept in mind throughout that the unification of the interfaces to the various kinds of modules is done to ease programming and to provide a facility for varying the calling aspects of their implementation without re-programming - not as a substitute for human understanding of the way the system runs.

If modules in a system typically are very large and complex such a variability would be extremely hard to achieve at a reasonable low cost because of the consequent complexity of the calling conventions and module specifications. Even if such an interface was defined the apparent variability would turn out to be useless as there would be relatively few such modules, and because large complex modules tend to rely on many resources being directly available so that they cannot easily be moved or changed.

In the following it will therefore, be assumed that modules are "quite small" and "simple", i.e. of the type and magnitude found in systems based on the principle of minimum privilege, type extension principle, data abstraction principle, etc.. This implies that a typical system module would be measured in hundreds of lines of (HHL) code, as opposed to tens or thousands, and would rarely access more than one "resource" at a time.

The Cambridge CAP system <Need77-1, Need77-2> is a prime example of this.

Programming Considerations

Modules which try to implement one single abstraction (and many which do not) appear to be of a common form:

```
BEGIN # pseudo Algol68 module #
  initialization code;
  DO # to infinity #
    get arguments;
    CASE first argument IN # one possible convention #
      ...
      services offered
      ..
    ESAC;
    return results
  OD
END; # of module #
```

Examples of modules conforming to this format are CAP protected procedures, most CAP processes, Simula67 classes, types, monitors, etc. in languages influenced by Simula. Obviously the CASE statement can be removed and services be offered as procedures where a language dependent interface is acceptable. Similarly the "get arguments" and "return results" can be made implicit.

If the module is implemented as a process the "get arguments" statement expands to the systems normal inter-process communication "receive a message" code, e.g.

```
WHILE messages (input)  =  none DO wait event OD;
X := receive message;
```

where X is of some standard "message" mode.

If on the other hand the module is implemented as a procedure a simpler scheme can be used for assigning a value to X, but again the assignment can only be made implicit at the cost of making the system language dependent, and does exist at some lower level anyway.

In nearly all systems and languages the decision of how to call a module (e.g. should it be a process or a procedure) has to be taken at the time where the module is designed, because of slight variations in the program to implement the receiving end of the interface. A uniform structure (dependent on the existence of a uniform calling mechanism) postpones this type of decision to compile, system generation, link, or even call time for modules which are not directly concerned with parallellism. This simplifies the initial writing of the module and makes it possible to change the module with respect to the calling interface without re-writing and maybe without re-compiling it.

A uniform way of writing modules is necessary for achieving this degree of variability, but a uniform calling interface can be implemented without enforcing (or defining) such a standard module format. This implies that one can allow non-standard modules in the system, e.g. processes with more than one request channel - without losing the other benefits of a uniform calling interface.

The uniform interface itself would free the programmer from the need to know the implementation of modules he wants to call in order to generate the correct calling sequences. If benefits could conceivably be obtained from parallel execution of a called module, then the calling module could be programmed using a sequence of operations specifying the desired parallellism, e.g.

```
activate (X, arguments);
..
..
Y := results; # implies possible wait for X to finish #
```

otherwise a procedure-like activation would be used. e.g.

```
Y := call (X, arguments);
```

It must then be assumed that the reference (pointer, descriptor, capability, or whatever) X contains enough information to identify the type of the module denoted by X so that the operations "call" "activate", and "results" can be interpreted so as to use the systems standard procedure, process, monitor, or remote process calling conventions. The overhead of this is quite low. One statement to find or isolate the type, plus one case statement of the simplest form to select the set of calling conventions that the programmer would have specified directly in a traditional system. An extra overhead is incurred when possible parallel execution is specified where the activated module is in fact a procedure, but this is again marginal as it consists mainly in providing

storage for the results while the "parallel" execution of the callers module takes place.

As an alternative to a uniform calling interface one can hide the actual implementation of a module behind a library routine, or as in the CAP system bury it behind a protected procedure acting as an interface or maybe even encapsulating and protecting unknown processes and/or procedures. This standard technique gives some of the convenience of a uniform interface at the cost of an extra layer of software, but it is not general unless there is some way of enforcing that all activations of a module "hidden" behind an interface procedure in fact go through that procedure.

If this is not the case anyone wanting to change th implementation of the module will still have to make changes to (an unknown number of) user programs - some of which he might not have the permission to change. Also, if the encapsulation is not used for all modules a programmer will still have to find out whether (and if so how) his particular module is encapsulated.

If the use of interface procedures is enforced at compile time the system is likely to become dependent on a single language, and if the use is enforced dynamically by some form of capability mechanism the interface becomes intolerably slow - though this overhead is quite acceptable in special cases and has been used with great success in the CAP system.

A Set of Inter Module Communication Primitives

There are of course numerous possible sets of inter module communication primitives which fulfil the general design aims. The set presented below should be seen as only one that suited a particular prupose and environment.

The scheme is based on passing of fixed sized argument blocks (though the primitive operations could be defined using other data structures). An argument block is used both for passing arguments and returning results. It contains a few words without fixed interpretation, but normally interpreted as integers, plus a few words interpreted as capabilities (references, descriptors, pointers). In requests the "first argument" is conventionally interpreted as an integer specifying the service required, and in results it is interpreted as a return code. Zero indicates normal return. The "last argument" is a capability interpreted as a module return link. This implies that a module cannot (in general) be called recursively, that the system can be deadlocked by recursion or mutual recursion, and that a module can "return" to a module different from its caller. If the "last argument" is not specified no reply is expected, no results can be returned, and the module can wait for a new request.

The primitive operations are:

ACTIVATE (X,A)

will run the module identified by X with the argument block identified by A. If possible X should run in parallel with the calling module.

```
Y:= GET ARGUMENTS (C)
```

will wait for some ACTIVATE or CALL of the current module to occur, and then make Y denote the argument block sent. The argument C denotes a notional communication channel. It can take two values "request" and "reply". This makes it possible to separate new requrests from replies. This is a minimal facility which can be expanded to allow more complicated strategies for running modules in parallel.

```
Y:= CALL (X,A)
```

is equivalent to

```
ACTIVATE (X,A); Y := GET ARGUMENTS (reply)
```

so it is not logically necessary, but can be implemented more efficiently as a primitive operation.

There is obviously a need for precise specification of the size of argument blocks, their management, the underlying message system for local and remote processes, how the "module return link" finds its way into the "last argument", what references are and how they are translated, what the communication channels used by GET ARGUMENT are, etc., but this is beyond the scope of this paper. It can be seen though that a module can be written without knowledge of the implementation of its calling interface (which therefore is variable), and that it can activate other modules serially or in parallel without knowledge of their implementation:

```
BEGIN # module #
  initialize;
  DO
    arg := GET ARGUMENTS (request);
    CASE first argument OF arg IN
      ..
      one service:
          ..
          res := CALL (A,arg2);
          IF first argument OF res ¬= 0 THEN error FI;
          ..
      ..
      another service:
          ..
          ACTIVATE (B,arg2);
          ..
          ..
          res := GET ARGUMENTS (reply);
          IF first argument OF res ¬= 0 THEN error FI;
      ..
    ESAC
    ACTIVATE (last argument OF arg, results)
  OD
END; # of module #
```

One final problem remains though. A process or a monitor module can access resources used by it alone without any form of interlocking, as an interlock mechanism is implicit in its implementation, but a module implemented as a procedure with instances in many processes must use an interlock. The simplest solution to this is to use a simple interlock mechanism on each access to such resources in all cases, knowing that the overhead involved in finding it locked will never be paid in the process and monitor cases. This will ensure that the potential advantage in using procedures in the case where the critical section is short relative to the whole module will be obtained. If the system does not contain an interlock mechanism which reduces to a single test instruction in the unlocked case, such a mechanism can easily be provided to ensure that the (expensive) system interlock mechanism will only be used when necessary.

Implications for System Structure

There are three major constraints which the interchangeability of procedures, processes, and remote processes imposes on the general structure of the system.

There can be no implicit passing of information in calls between procedures in the same process. This is arguably a good thing anyway, but for efficiency reasons tricks like not saving the registers in this case and so achieving a cheap way of passing integers is not uncommon.

The global context for a job cannot be provided in the data structure defining the current process (process base, state word) as the current process is quite likely to consist of only the module itself. Global parameters like the current input stream and the user identifier must either be banned and substituted by explicit passing or be provided as a special "current job" module or data structure. This has the effect of making the processes slightly cheaper to use, and also easing the implementation of scheduling and store management algorithms based on jobs rather than processes.

Error codes must always be returned to the calling module unless something else is specifically requested. This is essential in the case where the called module is implemented as a remote process and therefore susceptible to soft errors <Metc72>.

Reconfiguration of Software

Given a uniform inter module communication system one can experiment with the implementation of modules. This allows one to obtain data related to design decisions in the area of parallellism/serialization, and to tune a system to a specific hardware configuration and/or work load. In this way the choice of implementation of a module can be seen as the first and most powerful of the system short term scheduling parameters. As a matter of fact if one is willing to accept a process like format for all modules most of the effects of using the interface for specifying modules as processes and monitors can be obtained by using a flexible scheduling algorithm with extreme parameters.

Given the variability and consequently the possibility for data collection through experimentation there are three possible ways of exploiting the interface as a tuning tool:

1: simple experimentation and adjustments of the implementation by a system manager, who can specify the implementation of modules at system generation time in much the same way as he adjusts other scheduling parameters. Changes in the system (in particular in connection with remote processes) will be infrequent and wholley dependent on human decisions.

2: Adaptive algorithms can be introduced into the system controlling the implementation of all modules. For this to be possible the implementation of the uniform interface must be capable of handling changes on the fly. This in itself is costly (but possible), and a suitable algorithm would not be easy to find. This has been attempted with distributed systems <Farb73, Case77>.

3: It is conceivable that an algorithm could be found which given a specification of the hardware, the module interdependencies, and a "typical" work load could compute a suitable set of implementations for the modules. This would be ideal, but even specifying the inputs would be extremely difficult.

An additional problem with reconfiguration must be mentioned. It is possible to introduce a deadlock into the system by taking a module which acquires two resources in a time dependent manner and has previously been implemented as a process and making it into a procedure. Such problems must be handled by human intelligence - the uniform interface is not designed as a substitute for human knowledge of the system implementation, only as an aid in the design phase and for subsequent tuning.

Possible Implementation Methods

The uniform inter module communication mechanism is an interface to the usual calling mechanisms rather than a separate new mechanism. This implies that the implementation problems lie in avoidance of overhead rather than in actual building of new facilities. There are basically two ways of implementing it;

One can implement the communication primitives in something which can recognize the various types of modules and then perform the appropriate actions. Because of its complexity this "something" must be a high level facility - a module or a set of modules. This implies that one will incur a large fixed overhead in all cases and for that reason end up finding that the use of any kind of module will have a cost of the same order of magnitude as a process.

The alternative is to allow the simplest inter module communication primitive - e.g. a microprogrammed instruction for procedure like call of a module (i.e. no process switch or interlocking implied) - to escape to more complicated and therefore slower mechanism when the primitive is passed a reference it cannot interpret. If this technique is used in all cases the extra overhead will be nil in the simplest case and one escape action in the next simplest case etc. As an example one can take a CAP like system. The microprogram can handle the simple (protected) procedure call. A module in the

callers own process (known to the microprogram) can handle calls to other processes via the message system, and calls to remote processes will have to be redirected to a network interface process known to the simple message system. This method minimizes the overhead in all cases.

Conclusion

It is possible to write most modules in a system without considering their possible implementation as a procedure, process, monitor, or whatever. In doing so one achieves a powerful tool for tuning the software to specific hardware configurations and jobloads without noticeable overheads. The usefulness of this technique appears to be limited to systems in which modules are fairly small, context swtiching not too expensive, and where information is passed explicitly rather than relying on shared data.

Future Work

An inter module communication system of the class described in this paper has been used as the interface between modules in an operating system based on the CAPs operating system. This system is written in Simula67 and is running synthetic jobloads on a variety of (simulated) hardware configurations. It is used for providing data for a project to evaluate the use of procedures, monitors, and processes, and to explore the possibilities for using remote processes to exploit semi-distributed hardware configurations. Similar schemes for unifying process and procedure calls are being microprogrammed for the CAP computer.

Acknowledgements

This work was done while holding a research fellowship from the Danish Natural Science Research Council. I am grateful to Roger M Needham for constructive comments on an earlier version of this paper.

References

<Case77> Casey, L. and Shelness, N.
A Domain Structure for Distributed Computer Systems
ACM-OSR vol 11 no 5 (Nov.77) pp 101-108

<Farb73> Farber, D.J. et al.
The Distrubted Computing System
7th annual IEEE Comp. Conf.(1973)

<Need77-1> Needham, R.M. and Walker, R.D.H.
The Cambridge CAP Computer and its Protection System
ACM-OSR vol 11 no 5 (Nov.77) pp 1-10

<Need77-2> Needham, R.M.
The CAP Project - An Interim Evaluation
ACM-OSR vol 11 no 5 (Nov.77) pp 17-22

<Metc72> Metcalfe, Robert M
Strategies for Operating Systems in Computer Networks
Proc. ACM National Conference, June 1972, pp 278-281

<Wald72> Walden, David C.
A System for Interprocess Communication
in a Resource Sharing Computer Network
CACM vol 15 no 4 (April 72) pp 221-230