<DEUTSCH>FJCC.NLS;10, 30-MAR-72 15:28 LPD ;

(control)

The idea of >process< in MPS is similar to that in Krutar's system [] and also to the original idea of process in the context of operating systems [Dennis?].

Each process has a control state, consisting of a procedure call stack, and a static environment, which is a record containing a set of variables whose lifetime is identical with that of the process itself.

As in [Krutar], many processes may share the same code; the static environment and the base of the call stack are allocated when the process is created.

The process is also the level at which sharing of information and delimitation of name spaces occurs: all the procedures in a given code module (the static prototype of a process) share declarations for static data, and their instances in a given process all refere to the same instances of this data.

This is in constrast to previous software process systems where the mechanisms for data sharing and common environments have been left to preexisting facilities like linking loaders [Krutar] or forced into a tree structure [BBN LISP].

Each MPS process also has an owner, which receives control when the owned process fails to handle an exceptional condition; this is discussed in more detail below.

All three inter-process control facilities in MPS (procedures, ports, and signals) have essentially the same appearance to the user: a control transfer allows passing an arbitrary argument record and receiving an arbitrary result record.

("Arbitrary" meaning that the user has his choice at compile time: the discussion of data structures elsewhere in this paper describes the user's options in this regard.)

MPS allows constructing the argument record (in the caller) and decomposing it (in the callee) separately from the actual call; this reflects our belief that each control and data structures in MPS should be individually available to the user for his use in ways we may not have forseen.

All three mechanisms also involve an initiator or >subject< and a receiver or >object<; the way in which a given subject and object become connected together are the subject of the next section of this paper.

Consequently, in all cases calls and returns are potentially symmetric.

Although the syntax of the three facilities is slightly different

at present, we recognize that it should be unified in the future, in line with the MPS philosophy that the user of a virtual facility (external piece of code) should not have to know how that facility happens to be implemented.

>Ports< allow symmetrical passing of control from one process to another.

The running (subject) process stops and the object process resumes after the arguments have been passed.

The object process' control state is intact from when it last stopped.

No storage is allocated or released.

The object process can test which port control arrived through, so that a single process can provide several virtual facilities.

Obvious applications of ports include various kinds of transducers [Conway] [Krutar].

Another class of applications follows from the observation that a port connection can be broken and another process inserted in between without either of the original participants' knowledge: thus tracing and monitoring can be carried out in a flexible and simple manner.

>Procedures<, in addition to having all the properties that one expects from past stack-implemented languages like ALGOL or LISP, exist within processes rather than as free-standing entities; thus a group of procedures can easily share persistent data not accessible from the outside.

This resembles the ALGOL idea of 'own', but in ALGOL both the isolation of 'own' data and the accessibility of procedures follow the same nested naming structure, so that it is often inconvenient or impossible to have free-standing packages of procedures for carrying out tasks like managing a private data base.

(This problem is conventionally circumvented by some kind of linking loader which disregards the language's scope rules!)

Note that, while a procedure's static data environment is that of the process in which it resides, its control environment is that of its caller; thus calling a procedure in a given process does not disturb that process' suspended control state, even if the procedure does a port call on behalf of the running process.

>Signals<, which correspond roughly to PL/I CONDITIONS, allow a procedure to intercept selected exceptional conditions arising when control has temporarily left the process.

When a signal is generated, a control stack is scanned to find a procedure activation which is willing to handle this signal.

This is normally the control stack of the running process; however, a signal can be directed to a specified process, for example to cause an internal state change in anticipation of a port call.

The signal then becomes a call on a piece of code within the receiving procedure; no information is lost, so the signal catcher has access to the control state at the time of generation as well as the data of the catching activation.

The catcher may elect to resume from the point of generation (presumably having taken some kind of corrective action), abort the computation back to the catching activation, or propagate the signal onward (continuing the scan from the catcher's activation or in another process).

If the catcher chooses to abort, each activation being killed has a chance to accept a special signal called UNWIND and clean up any data structures which may be in transient states.

This ability, which is not present in any other known system, attacks a serious problem which arises in any system with software interrupts, namely that routines which manipulate data structures must resort to some kind of unpleasant interlock system to prevent themselves from being aborted in critical sections.

The ability to reject a signal, possibly redirecting it to a specific process, after accepting it and examining the message or doing some other computation allows natural handling of special cases of a general exceptional condition; no other known system has this facility.

If no catcher can be found within the target process, the scan continues in the owner of that process, and so on.

In this way the ownership structure can be used to provide a handler for a given exceptional condition for a whole group of processes.

Note that signals, like ports and procedures, and unlike PL/I CONDITIONS or LISP RETFROM, may involve passing a general argument record, so they can be viewed as calls on virtual (external) facilities in exactly the same way.

Since we expect signals to be used heavily for internal communication within groups of modules, we have taken some care to make enabling a signal (PL/I ON unit) very cheap and the generation and catching still comparable in cost to a procedure call.

All exceptional conditions, including logical faults such as attempts to use unconnected ports and machine conditions such as attempts to write on read-only information, are converted to signals, so that the user has full access to the trap facilities

of the extended machine.

One interesting application of signals might be in an interactive system where a general line input routine might recognize a particular word or character as a request for assistance and generate a "help" signal.

The innermost (i.e. most specialized) facility that had provision for helping would catch the signal and provide appropriate commentary; the signal would automatically propagate up to less specific levels as needed.

This illustrates the key motivation for signals, namely a desire to avoid forcing the complexity of an environment (internal or external) on all programs in the environment.

We can summarize the foregoing discussion by giving pseudo-programs for each of the three mechanisms.

In the following, D refers to the static data base register and LB to the process data base register.

```
(ProcedureCall)

   Create argument record;

   Save return and D on current stack;

   Branch through descriptor;

   Load new D;

   Create local variable record on stack;

   Copy arguments;

(ProcedureReturn)

   Create return value record;

   Branch through return (on stack);

   Load old D;

   Copy arguments;

   Destroy local variable record;

   Destroy argument and return value records;

(Port)

   Create argument record;

   Save return in process state area;
```

```
   Branch through descriptor;

   Copy arguments;

   Load new LB and D;

(Signal)

   Create message (argument) record;

   Search stack for catcher;

   | none          ↑     | found

   V               |     |

   Go to owner's stack;  |

                         V

   Save return and D (and LB if necessary) on selected stack;

   Create local variable record (for message only);

   Copy arguments;

   Load D (and LB if necessary) for catcher;

   Branch to catcher code;

(SignalResume)

   [Identical to ProcedureReturn except that LB may change as
   well as the D.]

(SignalAbort)

   Back to catcher's activation?

   | no          ↑

   V             |

   Delete top frame;
```

(cnotes)

(binding)

Since a primary goal of MPS is to separate the specification of a facility from the implementation of that facility wherever possible, most of the information about the meaning of names which the programmer must supply at compile time is interface information.

Thus the principal declaration for data objects is a record declaration; static environments of processes, dynamically

created data records, parameter lists of procedures, ports, and signals, activation records (frames, local variables, ...), and the MPS relative of named COMMON are all declared in essentially the same way.

This declaration does not commit itself in general to when instances of the declared records will be created, who will create them, or how the various instances will be shared, although process environments are tied to process creation and destruction and activation records are tied to procedure entry and exit.

In a similar vein, the programmer must declare the interface specifications of all the procedures, ports, and signals he uses in a particular module, but need not (and in some respects cannot) say anything about how to join these up to actual external facilities.

We have in mind that during early development of a program, when the human overhead of writing declarations might be onerous, and also in those few cases where it is impossible to specify the interface properly within the language, we will have some kind of interpretive mechanism which will verify the compatibility of data across control interfaces at run time.

To allow the specification of a complex of facilities to appear in a single place, MPS provides for data modules as well as program modules.

A data module simply contains declarations which may be effectively made part of any program module with a single statement of the form INCLUDE (data module name) in the program module.

The intention is that ALL the declarations which constitute the data interface between a group of modules be placed in a single data module, which is then INCLUDEd by each program module.

In this way administrative control over interface changes, a constant source of headaches in the development of large systems, can be centralized, but the implementation of a facility can be changed without any of its users' knowledge.

This is not sufficient by itself to assess the potential impact of an interface change; however, we intend to rely on NLS (the information structuring system which provides editing capability for MPL programs) to assist us further in this regard.

A similar idea was implemented in [SUE].

At run time, the programmer has facilities for concretizing the abstract relationships which he declared at compile time.

Thus there are CREATE and ALLOCATE facilities for creating processes and data records, and a JOIN statement for connecting

an external control reference with its target (i.e. external procedure reference with instance of procedure, port reference with port reference, and signal generator with signal catcher).

The traps for unlinked control transfers are converted to signals so they may be processed at the time of the transfer.

(Only hardware limitations prevent us from doing the same for data references.)

In fact, there is no clear separation between a "link" phase and a "run" phase; linking is carried out by programs which may be called at any time, so that the user may choose between traditional pre-linking and Multics-style dynamic linking on a case-by-case basis.

> The symbolic names of all variables are readily accessible at run time to allow the Multics ideas of local names and symbolic linking to be exploited.

For example, a programmer could choose to bind those procedure names which refer to standard library facilities at the time a process was created (and perhaps create instances of those facilities at the same time), but defer binding his own modules tgether so he could regain control at the terminal when an inter-module call was made.

He might want to define conventions for procedure names and write his own dynamic linker to use these conventions to decide where to look for the object of a given call.

He could have a program which used a symbolic file to specify the interconnections of his configuration and made all the connections, or placed monitoring processes on all connections on a given list (we expect to have some programs like this as part of a standard library).

> This example indicates an important difference between the MPS binding facilities and those of linking systems like the /360 link editor: bindings can be changed or even undone at any time.
>
> The latter implies that the user can force relinking by name, either overall or local, to introduce a new module into a running configuration to replace an existing one.

In the case of a module that maintained a complex table, the programmer might choose to create the table when the process was created and have a separate process for each instance of the table, or he could pass the necessary context information with each procedure call (this decision would have to be made at compile time).

Note that since all bindings of external names are under user control, a system can be configured or reconfigured without rewriting any part of it.

In fact, we believe that a properly written system completely separates the programs that provide facilities from the programs that connect them together; this organization can be extended to multi-level systems.

The following table summarizes the binding properties of the various kinds of named objects in MPS.

Type of object	Declared to compiler	Specified at run time
INCLUDEd data module	File name for module; Names of default base pointers (per scope)	Creation and destruction of instances; Assignment of instances to pointers
Process data	Names and types of variables	Creation and destruction of instances (processes)
Procedure local data	Names and types of variables	Creation and destruction of instances (activations; automatic on entry and exit)
General records	Names and types of components; Names of default base pointers (per scope)	Creation and destruction of instances; Assignment of instances to pointers
External procedure	Names and types of arguments and results; /File name to look on;/	Connection to actual <process, procedure entry>

	: /Projections/renaming/ :	
External port	: --Same as procedure-- :	: Connection to actual <process, port>
External signal	: --Same as procedure-- :	: Connection to actual signal code;
	:	: Catching activation
	:	: (at generation)

(bnotes)