AD-A092 103 ROYAL SIGNALS AND RADAR ESTABLISHMENT MALVERN (ENGLAND) F/G 9/2
POSER - A PROCESS ORGANISATION TO SIMPLIFY ERROR RECOVERY.(U)
JAN 80 J A MCDERMID

UNCLASSIFIED RSRE-MEMO-3249 DRIC-BR-72727 NL

1 OF 1
AD A
092103

END
DATE
FILMED
1-81
DTIC

RSRE

MEMORANDUM No. 3249✓

ROYAL SIGNALS & RADAR ESTABLISHMENT

POSER – A PROCESS ORGANISATION TO SIMPLIFY ERROR RECOVERY

Author: J A McDermid

PROCUREMENT EXECUTIVE,
MINISTRY OF DEFENCE,
RSRE MALVERN,
WORCS.

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Memorandum 3249

(18) LRIC

(19) BR-72727

Date: (11) January 1980

Title: (6) POSER - A PROCESS ORGANISATION TO SIMPLIFY ERROR RECOVERY,

Author: (10) J A McDermid

(14) RSRE-MEMO-3249

SUMMARY

(12) 8

This memo, describes a process organisation which has been developed with the aim of making automatic error recovery simpler than with conventional process organisations. It gives a high level description of the processes and of the channels (which provide the data communication). The memo, describes the more important aspects of implementation, and indicates how POSER has been implemented experimentally in a multi-computer simulation. The memo also briefly describes the checkpointing and error recovery mechanisms which would be used with POSER.

This memorandum is for advance information. It is not necessarily to be regarded as a final or official statement by Procurement Executive, Ministry of Defence

JOE

0) Introduction

This memorandum describes a process organisation which is proposed for use in the construction of fault - tolerant, real - time computer systems. The background to this proposal is outlined in [1]. The organisation is conducive to automatic checkpointing and error recovery. It is implementable on distributed computer systems, and the application programmer need not be aware whether processes to which he is communicating are in the same computer or not.

The organisation comprises processes and channels which are analagous (though not identical) to those used by MASCOT [2]. No shared data areas (pools in MASCOT terminology) are provided, and so communication between processes must be achieved entirely by message passing via the channels. The operation mode proposed is one of data flow at the highest level (process and channel interactions). Essentially the processes are conventional sequential programs. The channels simply move data, provide buffering, and produce the required process connectivity.

This organisation is advocated as it makes the causal relations between the process and channel activations clear and thus facilitates automatic checkpointing and (backward) error recovery. The organisation may not yield very efficient implementations of certain types of application, but the efficiency has yet to be evaluated. The organisation as described here has been implemented within a multi - computer simulation in Algol68RT [3],[4] and some attempt will be made to instrument this simulation in order to try to measure efficiency.

The process organisation is described here, primarily to provoke comments and reaction, and any (constructive) criticism will be welcomed.

1) High Level Description

1.0) Processes

A process is, in principle, a set of four objects (when loaded):

Process Code
Own Data
Input Parameters
Output Parameters

The process code is a fairly conventional procedure which may take parameters and deliver results. The own data is "long lived" data which is preserved from one process activation to the next. Normally the own data can only be accessed, and hence changed, by the code with which it is associated. The checkpointing and error recovery mechanisms provide exceptions to this rule (see sections 3 and 4). The process is eligible for running when (a suitable subset of) the input parameters are available and the output parameter space is empty - i.e. we are practising data flow at the process level. When a process is scheduled the process code (a procedure) is called, with the appropriate process parameters as its parameters, and this (conceptually) executes until completion, delivering its results into the output parameter space. This call - execution - return sequence corresponds to one process activation.

1.1) Channels

The construction of channels is essentially the same as that of processes. Channels are only permitted to move data, not to modify it. This restriction can only be enforced by programmer discipline in my simulation. This restriction could be enforced in several ways, e.g. by providing additional features to the application language (as has been done for MASCOT), or by the operating system supplying (generic) channels, as would be possible on FLEX [5].

1.2) General Considerations

The set of processes and channels which make up an application system is considered to be fixed, although their mapping onto the physical machine may vary with time. Thus an application program in the proposed formalism corresponds to a "frozen system" in MASCOT terminology. Since conventional hardware and operating systems do not directly support the proposed structure, additional software has to be provided to achieve this. This additional software is described in the following section.

2) Implementation

2.0) Data Flow

In order to achieve the effect of data flow at the process and channel level, the operating system has to provide special facilities. For processes and channels communicating within one computer, or via some form of common store, the data "movement" can be provided by mapping the appropriate part of the output space of the sender on to the corresponding part of the input space of the receiver. This achieves the data passing, but the presence of the data stil has to be made known to the scheduling mechansm (see section 2.1).

If the communicating objects are in different computers, and there is no suitable common store, then the relevant parts of their input and output spaces will be separate and the operating systems are responsible for transmitting the data between these data areas. The operating systems will use the same process structure as the application system at this interface, hence the transmission software will be made eligible for scheduling by the presence of the data. The receiver software, supplied by the operating system, will be activated when the data arrives from the communication medium (activation will probably be by means of an interrupt), and it will pass the data into the input space of the receiving process or channel.

The interfaces presented by the operating system software are such that the application programmer need not be aware that the communicating process and channel are in different computers. This is obviously essential if the process and channel mapping (distribution) can change in the lifetime of the system.

2.1) Scheduling

There is, in effect, a scheduling condition associated with each process and channel specifying which subsets of the input data have to be present, and what output space has to be free, in order that the process or channel may be run. Obviously the set of processes and channels which can be run at any time can be determined by comparing the state of the input and output spaces of each process and channel with the appropriate scheduling condition. Having established which processes and channels can be run, the one to be run can be selected by some algorithm which should try to be "fair".

The scheduling mechanism described above is essentially a polling type of mechansim when implemented on a conventional machine and is obviously rather inefficient. More efficient scheduling mechanisms can be devised, and a considerable improvement could be made even on conventional hardware. However the greatest improvement in scheduling performance could be achieved by providing special purpose hardware to perform, or support, the data flow communication.

Since the main purpose of the simulation is to check principles, rather than to consider problems of efficiency, it uses a polling technique which is little more elegant than that described above. The scheduling conditions are supplied by providing a procedure for each process and channel which evaluates the condition and delivers a boolean which is true if the process or channel can be run, and false otherwise.

2.2) Practical Implementation

This section describes the implementation of the above process structure as it is in the simulation. I would expect the implementation to be very similar in a real system. Each process runs within a "shell" which contains the scheduling condition. The contents of the shell are supplied by the application programmer, but the mode of the shell is constrained by the system. The shell may be regarded as providing the "virtual machine" necessary for process execution and data flow, which the underlying machine does not provide.

The process structure propsed could be implemented "on top" of MASCOT by treating the process and channel shells as MASCOT activities, and incorporating (essentially trivial) MASCOT channels to perform the data flow between these activities.

3) Checkpointing

3.0) The need for checkpointing

In order to recover from errors which may arise in a system, and hence to tolerate the faults which caused them (without losing any work), it is necessary to periodically make copies of the states of the components in the system which are essential to its operation. This process is known as checkpointing. A rationale for checkpointing is developed in [6], the most important point being that the checkpoints must represent the (mechanistic) causal relations between events in the system (process and channel activations). This is why a process structure in which these relations are manifestly obvious (due to the data flow operation) is advocated. The use of pools in MASCOT whilst not actively obscuring these relationships, does not (in the authors opinion) make them sufficiently obvious.

3.1) Checkpointing Mechanisms

The desired checkpointing can be achieved by taking copies of the data that flows down channels, and of the processes own data between process activations. The checkpointing must be partially performed from within the process and channel shells (unless we introduce unconcscionable cheating to circumvent scope rules). The operating system produces the actual checkpoints, and manages (garbage collects etc.) the checkpoints. The application system has to provide the data for inclusion in the checkpoints. This provision of data involves "unravelling" the own data (or changes to it) into a form suitable for transput, and delivering it to the operating system checkpointing procedures. This work is performed by a procedure defined within the shell. Protocols have to be provided to ensure the correct (consistent) generation of checkpoints - these are described in [6].

4) Error Detection and Handling

4.0) General

In order that a system can be made fault - tolerant, and that use can be made of checkpoints, errors which arise in the system must be detected. This will be done by a mixture of routining procedures and gross checks within the operating system, and detection mechanisms incorporated within the application system. These detection mechanisms must be provided by the application programmer, since only he understands the application, and hence only he can decide when it is operating correctly or not. However the operating system will provide facilities to assist the application programmer. The checks performed by the operating system will not be discussed here as they are not relevant to our current topic.

4.1) Error detection within a Process or Channel

The code for the processes and channels comprise conventional sequential procedures. These procedures will be block structured, and will, in general, consist of a hierarchy of blocks. The error handling (often known as exception handling) mechanisms are provided on a "per block" basis. Any error detected within a block will invoke exception action to try to complete the action of the block correctly. Success in error handling should (normally) be undetectable at the block above. Failure to handle the error will be reported to the level above. It is not necessary to cater for errors at each level in the hierarchy. If no error handler is supplied at any particular level the error is passed up to the nearest level in the hierarchy for which a handler is provided. This type of hierarchical exception handling can be implemented in a number of different ways:

by careful defensive programming in a conventional programming language;
by use of special language facilities (and the implicit support software) such as recovery blocks[7];
by use of a machine which supports these concepts, e.g. FLEX [5];
by use of an appropriate language e.g. ADA [8].

If an error can not be handled within a process or channel it will be reported to the process or channel shell.

4.2) Process Level Error Recovery

Recovery from errors which are passed to a shell will, in general, affect more than one process and will require cooperative action between the different operating systems. Recovery will normally be achieved by restoring the state of the system to one (which is hoped to be) prior to the occurrence of the error. This technique is known as backward error recovery and it can beimplemented using the checkpoints established whilst the system was running normally. This type of error recovery will not be discussed further here, but is considered in some detail in [6].

5) Conclusions

POSER is advocated for a number of reasons. The data flow operation makes checkpointing relatively easy, and makes the causal relationships between the process and channel activations clear. The structure allows relatively simple reconfiguration, as there are no common data areas to constrain the mapping of the processes and channels onto the hardware.

The attributes of the structure from the points of view of scheduling efficiency and ease of design of application programs have not been evaluated, but are being considered.

Many optimisations of the scheduler are possible - for example, only polling a process after the state of its parameters has changed. However it is expected that the most useful result of a study of the scheduling in POSER will be the definiton of hardware support to the data flow mechanisms which will make scheduling easy and fast.

The question of application program design is being tackled by producing an example program to run on the simulation. The program chosen is a model of an air defence system which is fairly typical of real time programs, and which is quite large (several thousands of lines of Algol 68). A version of this program implemented using the MASCOT philosophy is available so it will be possible to make some qualiatative comparisons of the complete programs.

There are many practical problems associated with the implementation of the proposed process structure, and more particularly, with the error recovery mechanisms. Perhaps the most fundamental problem is that of reducing the amount of data which has to be stored in the checkpoints. These problems will be considered as the development of the simulation proceeds.

6) References

[1] Fault Tolerant Computing, RSRE Memorandum 3197, J A McDermid, 1979.

[2] MASCOT a Modular Approach to Software Construction Operation and Test, RRE Tech. Note 778, H R Simpson, K Jackson, 1975.

[3] Algol 68R Users Guide, P M Woodward, S G Bond, HMSO, 1974.

[4] Parallel Processing and Simulation, P M Woodward, MOD unpublished work, 1974.

[5] An Introduction to the FLEX Computer System, RSRE Report No. 79016,J M Foster, C I Moir, I F Currie, J A McDermid, P W Edwards, J D Morison, C H Pygott, 1979.

[6] Checkpointing and Error Recovery in Distributed Systems, J A McDermid, RSRE Memorandum (to appear).

[7] Recovery Blocks in action: a system supporting high reliability, Proc. Int. Conf. Software Engineering, San Fransisco, 1976.

[8] Preliminary ADA reference manual, J D Ichbiah et al, ACM Sigplan Notices, Vol. 14 No. 6June 1979.

"References quoted are not
necessarily available".