SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 9, 149–155 (1979)

Programs as Higher Level Subroutines†

DOUGLAS JONES, A. B. BASKIN, THOMAS CHEN

Medical Computing Laboratory, School of Basic Medical Sciences, University of Illinois at Urbana–Champaign, Urbana, Illinois 61801, U.S.A.

AND

LOUIS BLOOMFIELD

Regional Health Resource Center, 1408 West University Avenue, Urbana, Illinois 61801, U.S.A.

SUMMARY

The subroutine call is one of the most fundamental of program control constructs. Despite this, it is rarely implemented at the job control or task level in existing commercially available software systems. When the feasibility of adding a general program calling mechanism to an existing system was investigated, it was apparent that there exist constraints on the amount of state information which could be saved on behalf of the calling program. A mechanism with low run-time overhead which saves a minimal amount of state information has proven to be easily integratable into a commercially available operating system. This mechanism has had a significant impact on the ease of development and support of large systems of programs. Examples of the use of the new calling mechanism include a program development system and a recursive directory manager for hierarchical directories.

KEY WORDS Calling sequences Operating systems Job control languages

INTRODUCTION

Although the subroutine call is one of the most fundamental of program control constructs, it is not supported at the program or task level by many commercially available operating systems. When the feasibility of adding a general program calling mechanism to the MAX IV operating system on the MODCOMP IV computer was investigated, it became apparent that only a minimal amount of state information could be saved on behalf of the calling program. This precludes the implementation of general mechanisms such as those supported by the UNIX,[1] HYDRA[2] or CAP[3] systems. A mechanism with minimal storage requirements is described here. This mechanism was implemented under MAX IV and has proven to be of great utility in the support of large systems of loosely coupled programs.

Ideally, operating systems should allow programs to call on the services of other programs just as programming languages allow procedures to call each other within a program. On most systems, service and input/output routines may also be treated by the programmer as a special class of subroutines. In fact, the distinction between system services, user subroutines and programs can be made to disappear entirely on some systems, such as those

† This work was supported in part by Grant Number 1 T15 LM 07011–01, National Library of Medicine, NIH and, in part, by Grant Number 1 RO3 HS 02839–01 from the National Center for Health Services Research, HRA.

0038–0644/79/0209–0149$01.00

Received 6 March 1978
Revised 15 May 1978

with capability based addressing.[4] All linkage conventions may be described in terms of process activation; for example, conventional procedure linkage can be described in terms of the calling procedure activating the called procedure and then suspending itself until it is resumed by the termination of the called procedure. Regardless of the nature of the routines involved, there must be a calling sequence by which one routine invokes another and a return sequence by which the calling routine is resumed. Additionally, some data structure must be provided, typically a stack, where the identity and data of the caller can be saved while it is suspended.

When all of the procedures needed by a program are known in advance, they may be compiled or link edited into a single load module. Though this approach may be adequate for some applications, it becomes difficult or impossible when the procedures which are to be connected are written in languages with incompatible calling sequences (for example, a FORTRAN program calling a PASCAL procedure[5]). Link editing can be cumbersome when the routines in question are large and used by many other programs; in this case, many different link-edited versions of the same code must be stored which differ only in their connection to differing calling environments. Additionally, on systems with relatively small virtual address spaces, the number of nesting levels can be severely limited by the requirement that the calling procedure must remain addressable while the called one is executing.

The use of a job control language for program interconnection is the most commonly posed alternative to link editing (excluding the 'chain' or 'link' services of many systems which merely provide a GOTO from one program to another). This overcomes the problems of connecting programs written in diverse languages as well as allowing programs to be selected dynamically at run time, either interactively or by the automatic generation of job control language for later interpretation. However, even if an ideal job control language is available, this still imposes rather severe limits on overall program structure unless user programs can also call on the job control language processor. These limits are equivalent to those of a programming language where only the main program may call subroutines; admittedly, one could write any program in this language but the result would tend to be clumsy and full of extraneous global variables.

IMPLEMENTATION REQUIREMENTS AND THE RESULTING MECHANISMS

Several requirements shaped the interprogram linkage mechanisms that were implemented. The MODCOMP IV has a relatively small virtual address space (128K bytes), and the MAX IV operating system supports segmenting mechanisms that are insufficient to allow the removal of idle code from the virtual address space when unneeded. Under MAX IV, each transient task must have a unique resource description cataloged on secondary storage. Thus, there would be a high overhead if interprogram linkage were implemented in terms of intertask linkage. The need for fast interactive response time and an unwillingness to develop the required storage management routines ruled out the use of secondary storage for saved copies of inactive calling programs, while the re-entrancy of many frequently used programs allowed load times to be largely ignored. Additionally, the new mechanism was required to support the old job control language processor; this had been supported by a primitive two-level task structure in which the job control processor was restarted whenever it or one of its overlays terminated or was aborted.

It is important to observe that it is not necessary to save entire calling programs with all of their data while they are suspended. Given a strong distinction between programs

and their data, only the data must be saved; the program itself can be reloaded when it is to be resumed. In many cases even much of the data is redundant. For example, a job control processor does not need its work buffers saved during a call; it only needs to be restarted and told where and in what file it was when it left off, with perhaps a small amount of state information. The same holds true for text editors and a number of other common applications.

The above considerations suggest the following departure from the procedure call model for interprogram linkage: instead of suspending the execution of the calling program until the called program terminates, the calling program is itself terminated before the called program begins execution and it must be reloaded and restarted when the called program terminates. Effective use of this type of linkage requires that the calling program pass parameters not only to the called program but to the version of itself that is to be reloaded when the called program terminates. Thus, the only data saved on behalf of the calling program will be that which it has explicitly saved for itself.

The resulting mechanism requires a fairly small stack in which each entry consists of enough information to load a program and the parameters to be passed to that program when it is started or restarted. The top of the stack always contains the name of the current program and its parameters. Three new system services are needed: a call service, a return service and a service to interrogate the stack top and determine whether the current program was started by being called or by the termination of a called program.

SECURITY AND ERROR RECOVERY CONSIDERATIONS

An important attribute of any calling mechanism is the action to be taken when a routine is called incorrectly or when a routine is called that exceeds its authority. When the mechanism is to be used to support job control and interactive system functions, some form of error recovery is necessary (clearly a job stream should not be terminated just because one job contains an error).

The only error recovery mechanism easily implemented under the constraints already mentioned involves treating fatal errors on the part of a program as a special class of return. If this is done, then a restarted program must be able to distinguish between the normal and error restart conditions. Furthermore, many circumstances arise where error recovery is not desired, so when a program calls on another, it must have the option of requesting that it not be restarted when there is an error. This is equivalent to requesting that if the called program aborts, the caller is also to be aborted.

With this error reporting convention, programs may be written using a limited form of either recovery blocks[6] or the undesired event protocols of Parnas.[7] In the first case, the calling program must save enough information to restore its state before the call in case it is restarted and discovers an error. In the second case, the called program must recognize a potential error and reach a defined state before setting an error flag indicating the nature of the error and aborting itself. Clearly, the small amount of restart information that can be saved during a call is the primary limit on the use of either approach.

When programs can have differing privileges, the calling mechanism can play an important role in preserving system integrity. If a program calls on another, it should be possible for that program to pass a subset of its privileges to the called program. In addition, it should be possible for a program to limit explicitly its use of some privilege while retaining the right to pass that privilege to programs it calls. Thus, the call service itself must allow a specification of which privileges are to be passed and the calling mechanism

must save both the privileges available to the caller and those that it may pass to programs it calls. It should be noted that programs under MAX IV may run in either privileged or user state and that the basic unit of protection for all other resources under MAX IV is the task, not the program.

IMPLEMENTATION

In order to support the new linkage mechanism under MAX IV, an extensible stack area was added to each task control block. Each entry in this stack contains the name of a program and its parameters, along with a privilege limit and information about when that program wishes to be restarted. The top entry always refers to the currently running program and contains additional information about why that program was started or restarted and what program ran previously. Two new services, CALL and WHYME, were added to the system (see Figure 1). The CALL service replaces the stack top with new information

Figure 1. A PASCAL description of the system-independent aspects of the services implemented

```
TYPE return = (exit, abort);
     returns = SET OF return;
     loadfilename = { system dependent };
     parameters = { system and application dependent };
PROCEDURE call    (restart: returns
                       { specifies under what circumstances the caller wishes
                         to be restarted, a value of [] indicates that the
                         caller never wishes to be restarted.                  }
                   ; privilege: boolean
                       { if false, the called program may not run privileged
                         even if the caller was allowed to be privileged.      }
                   ; myname: loadfilename
                       { the load module to be used in order to restart this
                         program.                                              }
                   ; myparameters: parameters
                       { the parameters to be used in restarting this
                         program.                                              }
                   ; newname: loadfilename
                       { the load module to be called.                       }
                   ; newparameters: parameters
                       { the parameters to the called program.               }
                   );EXTERNAL;
PROCEDURE whyme (VAR why: returns
                       { if (why = [exit]), the current program was restarted
                           after some program called by it terminated;
                         if (why = [abort]), the current program was restarted
                           after some program called by it aborted;
                         if (why = []), the current program was called by some
                           other program;
                         (why = [exit, abort]) will never occur.               }
                   ;VAR myname: loadfilename
                       { the load module name used in loading the current
                         program.                                              }
                   ;VAR oldname: loadfilename
                       { the load module that was running before the current
                         program was loaded; this is the program that aborted,
                         exited or called as reported by why.                  }
                   ;VAR parameter: parameters
                       { the parameter values passed to the current
                         program by the caller or by some previous
                         incarnation of itself.                                }
                   );EXTERNAL;
```

indicating how and when to restart the current program, and then pushes a new entry on the stack describing the called program before loading and running it. The existing EXIT and ABORT services were modified to accomplish the return function by searching down the stack for an entry with appropriate restart conditions and loading and starting the indicated

program. The WHYME service allows a program to inspect the stack top and discover why it was loaded and what parameters were passed to it.

Because of extensive use of existing parts of MAX IV and careful attention to facilities already present, these new services only involved a few hundred lines of assembly code and required one man-month of effort to specify, implement and debug. For example, existing MAX IV internal utility routines were used for stack management and user program loading. A further simplification was achieved because the existing MAX IV global task variable structure was sufficient for storing return values. Thus, it was only necessary to implement parameter passing by value.

EXAMPLES

The system services described above were used to implement the user interface to an interactive time-sharing system, including the sequencing of users from the logon processor to the appropriate user environment to the logoff processor. The time-sharing system is based on the assignment of one task per user terminal (see Figure 2). At the root of each

Figure 2. The hierarchic structure of programs accessible to a user at an interactive terminal, including the program development system. Deeper indenting levels represent called programs in the hierarchy. Thus, program indented under a given program may be called by that program

```
ROOT
    LOGON—Initial password check, determines user privilege.
    —A program selected by LOGON on a user dependent basis; i.e.:
    MEDIKAS—A special semantic network data base manager.
        —Any of a number of component programs of MEDIKAS.
    AUTHOR—A program development system.
        DIRECT—Hierarchic directory manager selected when the user tries to
                edit a directory.
            DIRECT—Recursive descent into directory structures.
            —An editor selected to edit a node of a directory on the basis of
             file type and user preference.
        —An editor (selected as from within DIRECT) run when the user tries to
         edit a data or text file.
        COMPRUN—Compile and run a program.
            —Compiler selected by language field of file type.
            —The compiled program.
        COMPCAT—Compile and catalog a program.
            —Compiler selected by language field of file type.
            —Cataloger for compiler output.
        —User selected previously catalogued programs; i.e.:
        BATCH—A traditional Job Control Language.
        MEDIKAS—A data base user interface.
    .
    .
    .
    LOGOFF—Updates system usage records.
```

such task is a program that is loaded when that task is activated. On initial startup, this root program calls on the logon processor. The logon processor in turn returns the name of the program that the logon records indicate the current user is to run, and the root calls that program. The root requires privilege in order to establish and de-establish itself, but unless the logon records of the user indicate that the user is allowed to run privileged programs, the root never passes any form of privilege to the programs it calls.

In this arrangement, the program associated with a user's logon records can be used for many purposes. If the user is an 'end user' of some system service, such as a particular data base management system, then that user may be routed directly to the programs managing that service. Users involved in program development could be routed directly

to a job control language interpreter, but it is frequently preferable to use a specialised program development system. A program development system was implemented that remembers the name of the program being worked on and automatically identifies the programming language being worked with. The result is a system with only four commands (edit, compile and run, compile and catalogue, run catalogued program), where each command has a single optional parameter, the program name, which defaults to the previous value of that parameter.

A third application, directory management, demonstrates the advantage of a recursive program calling sequence. When the program development system recognizes a request to edit a directory, it calls the directory manager instead of a text editor. Among other functions, the directory manager allows selection of directory entries for editing. Given a hierarchically structured file system, the directory manager must call itself recursively each time that a subdirectory is to be edited. The recursive calls for editing sublevels of the director structure request return only on normal exit so that a user can escape from a deeply nested directory structure to the program development system by aborting the directory manager.

It is important to note that the new linkage mechanism can be used to call on any programs that ran under the old system. Programs that ran as user programs or utilities can still be called, and they will return correctly because of the redefinition of the EXIT service. The old job control processor poses a slightly more difficult problem because it overlays itself with other programs and expects to be reloaded and restarted when they exit. The problem is that the old overlay service does not save the identity of the caller. This can be solved by calling the job control processor indirectly through a special program responsible for restarting it when one of its overlays exits.

CONCLUSION

Although the program calling mechanism described is less general than a traditional procedure calling mechanism, its impact on the development of complex program systems has been profound. The use of the calling mechanism for a program development system, a hierarchical directory manager and various special purpose 'end user' interfaces has demonstrated the desirability and utility of using programs as higher level subroutines. Considering the short time that the program linkage mechanism itself took to implement and the user acceptance of program systems based on it, such a mechanism deserves serious consideration for implementation on other existing operating systems.

ACKNOWLEDGEMENTS

The authors would like to acknowledge the efforts of Lawrence Sherman who coded the directory manager and the program development system. We would also like to thank the referee for his constructive comments.

REFERENCES

1. D. M. Ritchie and K. Thompson, 'The UNIX time-sharing system', *Comm. ACM*, **17**, 365–375 (1974).
2. W. Wulf, E. Cohen and W. Corwin and coworkers, 'HYDRA: the kernal of a multiprocessor operating system', *Comm. ACM*, **17**, 337–345 (1974).
3. R. M. Needham and R. H. Walker, 'The Cambridge CAP computer and its protection system', in *Proc. of the Sixth ACM Symp. on Operating Systems Principles*, Purdue, 1–10 (1977).
4. P. J. Denning, 'Fault tolerant operating systems', *Comput. Surveys*, **8**, 359–389 (1976).

5. P. R. Mohilner, 'Using PASCAL in a FORTRAN environment', *Software—Practice and Experience*, **7**, 357–362 (1977).
6. P. M. Melliar-Smith and B. Randell, 'Software reliability: the role of programmed exception handling', in *Proc. of an ACM Conference on Language Design for Reliable Software* (Ed. D. B. Wortman), Raleigh, N.C., 95–100 (1977).
7. D. L. Parnas and H. Wurges, 'Response to undesired events in software systems', in *Proc. of Second Int. Conf. on Software Engineering*, San Francisco, 437–446 (1976).