J.R. Bell and C.G. Bell (eds.), Minicomputer Software
North-Holland Publishing Company, 1976

TECHNIQUES FOR DETECTING AND PREVENTING MULTIPROGRAMMING BUGS

Bernard P. Cosell, John M. McQuillan and David C. Walden
Bolt Beranek and Newman, Inc.
Cambridge, Mass

ABSTRACT

Techniques are presented for analyzing the multiprogramming structure of a system and for verifying that a system does not contain any multiprogramming bugs.

INTRODUCTION

Multiprogramming bugs are those which can occur when two processes within a system attempt to modify a variable simultaneously or when one process interferes with the control structure of another by incorrectly sharing a subroutine with it. These bugs occur in multiprocessing systems, where the conflicting processes are truly concurrent, as well as in multiprogramming systems, where only one process can be active at a time. A bug of this type is frequently very difficult to locate because neither process can detect any anomaly. Furthermore, since a failure caused by such a bug depends on the coincident timing of the conflicting processes, a failure may occur only once a minute, once an hour, or even less frequently. Also, problems often will not show up until the next time the variable is referenced, which may occur long after and far removed from the actual source of the fault.

Dijkstra and others (1968) have done much theoretical work on the general problem of coordinating independent processes. However, when execution speed and program size are critical factors, the general procedures they developed are too expensive to use every time a variable is referenced. Further, in many environments the actual protection techniques available are fairly straightforward. It is our belief, then, that what is needed is not further work on "automatic" protection schemes, but rather techniques for identifying those portions of a program which actually require protection, thereby allowing the remainder of the program to be left unencumbered. Our work has focussed on these practical problems within the context of assembly language programming for small, real-time systems.

We concentrate for the most part on interrupt-based systems because this is the environment with which we are most familiar and in which we have done the most analysis. Conceptually, however, all multiprogramming systems merely simulate a multiprocessing system. The primary attribute of such systems is the requirement for rapid response to external events. Interrupts are the most common architecture with which such systems are constructed. Although the techniques we have developed are designed for interrupt-driven systems, they apply with little significant modification to the entire spectrum of multiprogramming systems.

As an example of a common interrupt bug, consider a system in which there is a loop to process data and an interrupt routine to service an input device supplying the data. The routines communicate by means of a variable which is incremented by the interrupt routine each time a datum arrives and is decremented by the processing loop each time a datum is processed. The variable is zero when the system is idle, indicating that there is no data waiting to be processed. Now, let us hypothesize that the interrupt routine and the processing loop look something like:

```
INTERRUPT:      SAVE ACCUMULATOR
                LOAD VARIABLE
                INCREMENT ACCUMULATOR
                STORE INTO VARIABLE
                RESTORE ACCUMULATOR
                EXIT

LOOP:           LOAD VARIABLE
                TEST ACCUMULATOR EQUAL TO ZERO
                IFSO GOTO LOOP
                DECREMENT ACCUMULATOR
                STORE INTO VARIABLE
                PROCESS A DATUM
                GOTO LOOP
```

An observant systems programmer will notice that the above routine, as simple as it is, contains an "interrupt bug". Assume that the variable has some value, n. If a datum arrives just after the LOAD VARIABLE instruction, an interrupt will occur and the interrupt routine will increment the variable to n+1. When the loop is resumed, the now-incorrect value of n will be restored to the accumulator and

will get decremented, and instead of containing n at the beginning of the next loop, the variable will contain n-1; a datum will have been lost.

One way to fix this bug is to rewrite the processing loop as follows:

```
LOOP:   DISABLE INTERRUPT SYSTEM
        LOAD VARIABLE
        TEST ACCUMULATOR EQUAL TO ZERO
        IFSO GOTO ENAB
        DECREMENT ACCUMULATOR
        STORE INTO VARIABLE
        ENABLE INTERRUPT SYSTEM
        PROCESS A DATUM (we assume no conflict here)
        GOTO LOOP

ENAB:   ENABLE INTERRUPT SYSTEM
        GOTO LOOP
```

The variable is being shared between two routines, and for a small portion of the time its value is "incorrect": from the LOAD until the STORE. During this period, the interrupt routine must be prevented from running and using the incorrect value of the variable, and the simplest way to do this is just to disable the entire interrupt system, preventing all interrupts.

A fairly simple rule suggests itself: if any routine modifies a shared variable, it must prevent any other routine from using the variable while it is being changed. This rule is the crux of interrupt programming.

A SYSTEMATIC APPROACH

The first step in understanding a system is to determine its interrupt structure, which usually follows from an analysis of the desired response characteristics of the system and its I/O devices. This will result in a directed graph in which the nodes correspond to the interrupt routines and the arcs indicate which routines should not be interrupted by which others. For typical computers, the interrupt system will constrain this structure to be linear, but in others a full graph is realizable.

In our particular case, although the computer in fact poses no constraints, we chose to implement a linear system. There are two primary reasons for this choice. First, more complicated structures offer little, if any, advantage over a linear system. Second, the interrelationships of the interrupt routines become much more complicated when structure is non-linear. Except as noted, for the remainder of this paper we will deal exclusively with linear interrupt structures. The approaches and analyses can be carried over to more complicated structures fairly easily, but in doing so the rules we describe will rapidly become very difficult.

The next step is to identify the system's interrupt routines. This identification is usually a simple matter of starting at each interrupt entrance and tracing out the control paths until the interrupt exits are reached. Non-reentrant shared code and subroutines are treated as though they were variables; that is, the techniques described below for dealing with variables are applied to determine an "effective interrupt level" for the code or subroutine, and then the analysis proceeds as though the code were actually a part of the level thus determined, the procedure repeated iteratively until all of the code in the system has been associated with an interrupt.

For each variable in the system, the highest priority routine which references it may do so with impunity. However, all lower routines must take care that they are not interrupted while modifying the variable in question. If the lower routines lock out interrupts at (and implicitly below) the level of the highest routine, they will be assured error-free access to the variable*. Thus, for each variable we must determine an "effective interrupt level", the level at which interrupts must be disabled (either implicitly by being the interrupt routine at that level or explicitly in lower levels) to guarantee safe access. To do this, examine each such element and determine which routines share it. The "highest priority" routine is the one which cannot be interrupted by any of the other routines sharing the element; its level is assigned to the element.

In an existing system, locating an interrupt bug probably requires some degree of insight into the structure of the program. Nonetheless, by methodical application of the above rules, it is possible to verify the structure of an entire system and discover any bugs present in it. In fact, for some bugs this may be the fastest way to locate them.

---

* It is primarily the absence of this property that makes non-linear structures difficult to deal with.

If we number the interrupt levels so that a given level can be interrupted by all lower-numbered levels and implicitly inhibits all higher-numbered levels, then the effective (or "hardware") levels we have just assigned are simply the lowest numbered levels which use each element. Now, when any higher numbered routine uses an element, it must at least disable interrupts up to that level. Notice that what the routine is actually doing is making itself appear to the interrupt system as though it were the lower numbered interrupt. Thus, one can view the situation as being that the lower priority routine has, through software, "become" a higher priority routine. We refer to this procedure as making the routine modify its "software" interrupt level.

The hardware level of a routine reflects which routines it can interrupt, and the software level indicates which routines are prevented from interrupting it. At the point of the interrupt entry, these levels are the same, and a routine can neither alter its hardware level nor make its software level higher than its hardware level. It can, however, decrease and restore its software level as necessary to insure safe access to any variables it might need.

Our basic approach consists of a mechanism which simply maintains and displays the hardware and software levels for each line of code and variable in the system. With this information as a guide, the programmer can proceed on his own to resolve any conflicts that are pointed out. The level information is inserted, modified and acted upon in a completely manual fashion; it is less an augmentation of our programming system than it is a documentation technique. Nonetheless, it has proven valuable for several reasons: 1) the rules are quite simple and deterministic, and lead to a high level of program correctness when properly applied; 2) the documentation aspect of the system is, in itself, valuable; and 3) forcing the programmer to be aware of multiprogramming issues improves the overall reliability of his programs.

There are six macros to declare the hardware and software levels of the program. In addition to setting the levels, they perform various consistency checks. The first three assemble appropriate code to achieve the desired effect upon the interrupt system:

1) INT N declares the interrupt entrance at level N.
2) INH N locks interrupts at level N.
3) ENB restores the interrupt system to the current hardware level.

Because programs have transfers and subroutine calls and non-contiguous fragments of code, there are analogs to INT, INH and ENB which effect the declaration for the purposes of checking but presume that the interrupt system is already at the declared level, and thus do not generate any code:

4) LEV N declares the code to be hardware level N. This is an implicit INT.
5) LCK N declares the code to be at software level N. This is an implicit INH.
6) RET declares the code's software level to be equal to its hardware level. This is an implicit ENB.

Each word of code that is assembled and shown in the listing is accompanied by its hardware level, and also by its software level if different. Variables are indicated with a V and the determination of their effective levels and the verification of their correct use is done manually, or otherwise, distinctly from the assembly process. For example, some levels that have been compiled for our IMP program (Heart 1970) include:

```
M2I = 0   Modem-to-IMP
I2M = 1   IMP-to-Modem
I2H = 2   IMP-to-Host
H2I = 3   Host-to-IMP
T.O = 4   Timeout
TSK = 5   Task
BCK = 6   Background
```

A sample use of these levels might be (in TASK):

```
5     LDA THIS        /GET THIS PACKET
5     INH I2M         /LOCK OUT I2M
5 1 STA EMQ XI        /ADD NEW PACKET TO QUEUE
5 1 STA EMQ X         /EMQ = END OF MODEM QUEUE
5 1 ENB               /COME BACK TO TASK LEVEL
5     JMP FOO
```

The effective levels for THIS and EMQ must be "known" (for example, by reference to a previous assembly). THIS is on level 5; it is a temporary in Task. EMQ is on level 1; it is shared by I2M, Timeout and Task.

The INH-ENB mechanism is somewhat simplistic; in practice, things can be quite a bit more complicated. The principles always remain the same, but it is not always clear just how to prevent a given routine from running. There can be

assorted interlocks in the software: it might be that a particular interrupt cannot occur (for example, because the device is known to be inactive), or it might be that the higher priority routine cannot take a path which accesses the variable in question (for example, in setting up the parameters of a datum before a flag indicating that there is a datum has been set). However, once the key places in the system are pointed out, it is almost always easy to implement the controls or verify that they are already present.

In systems with more complicated interrupt structures, applying the rules becomes correspondingly more difficult. Nonetheless, even the most intractable of structures will yield to the steadyhanded application of the rules. We give three examples. First, consider two user-level jobs within a timesharing system. The timesharing system itself is probably a complicated interrupt-driven system, but let us direct our attention only to the two user jobs. The nature of a timesharing system is that at any point a user job can be interrupted, "swapped out", and another user job run in its place. User jobs typically have no explicit way to "inhibit" one another. So if two jobs wish to communicate through some shared structure, be it shared memory or a shared disk file, considerable care must be given to prevent interrupt bugs from cropping up. Again, the mechanisms chosen will vary widely, but the basic rules can be used again and again to pinpoint those portions of a program which require protection.

Second, consider two processes (for instance, coroutines) that both run at background level, which suspend and resume processing on some basis (this is, in essence, the heart of a polling system). If these two processes share a common resource the following kind of bug can occur:

```
A                          B

LOAD X                     LOAD X
...                        ...
(suspend processing)       (suspend processing)
(resume processing)        (resume processing)
...                        ...
STORE X                    STORE X
```

Because B can run between the time A suspends and resumes and vice versa, either's STORE X could be an error if the other has run and changed X. Again, the problem is that a routine modifies a shared variable in an "interruptable" way. A systematic solution to this problem can also be developed based on the assignment of local levels within the main hardware level to all routines and variables and the use of the rule that a routine implicitly inhibits all lower priority levels and only its own sublevel -- each sublevel considers the other sublevels at its level as being "higher" than it.

Our third example of a situation requiring an extension to our systematic approach is that of a true multiprocessor. Here the problem of concurrent access to shared resources is present at all times. Protection in such an environment can be effected by the well-known uninterruptable "test and set" instruction (Dijkstra 1968) as implemented, for example, in (Heart 1973). However, now a new type of problem arises: when routines require several "locks" at once in order to proceed, deadlocks can occur if all routines do not take and release locks in a coordinated way. An extension to our approach is of value here: if the locks are assigned levels and routines are constrained never to take a lock with a lower number than one it already has (i.e., locks must be taken in strictly ascending order), our system can be expanded to provide the necessary sequencing information to allow the avoidance of lock-ordering deadlocks.

AUTOMATING THE APPROACH

In this section we discuss a means of making our approach automatic. The procedure we are going to describe is a "cookbook" for assigning levels and verifying that there are no bugs. Much of this work can be done by a program, and the procedure has been organized with that in mind.

For the first step, we identify and assign hardware levels to all of the executable code in the system. We start at the actual hardware interrupt entrances and assign to them the actual level of the interrupt they represent. We then trace through the code in the natural way, assigning the starting hardware level to all code (not in subroutines) "reachable" from the interrupt entrances (an instruction which occasionally skips assigns its level to the next two locations, a transfer assigns its level to its effective address(es), etc.). A subroutine call propagates its level to all of its possible exits.

With all of the main line code dealt with, we next examine each subroutine to see if all of its calls have had their hardware levels assigned. If so, we assign the minimum of the calling hardware levels as the hardware level of the entry to the subroutine. We then assign hardware levels to the rest of the code in the subroutine, and then loop back to find other subroutines all of whose calls have had their levels assigned.

This procedure should result in every line of executable code having a hardware level assigned to it. If some code has remained unassigned, then it is not ever executed. If some code has attempted to have several hardware levels assigned to it, there is probably a bug in the control structure.

Next we assign hardware levels to the data and variables of the system. First, all of the read-only variables (i.e., constants) are located and marked as such. All other variables are assigned the hardware level of the minimum of their referents. This should assign a hardware level to all variables, and every used word in the program, both code and data, will now have a hardware level assigned to it.

Now we assign software levels to the code. Each interrupt entrance is assigned a software level equal to its hardware level. Then, for all instructions which are neither at the returns of subroutines nor instructions which affect the interrupt system (e.g., LOCK or UNLOCK) we propagate the software levels as we did the hardware levels. The level of the instruction following one which affects the interrupt system should simply reflect whatever was done to the interrupt system, and then propagation can continue. Subroutines are a little more complicated.

We initially assign to each subroutine entrance a level of zero and propagate that assignment through the subroutine to determine a tentative exiting software level for each exit from the subroutine. Then, as we encounter a subroutine call instruction (either in the main code or in another subroutine) we compare the software level of the call and the tentative software level of the subroutine entrance. If the calling level is lower-numbered than the subroutine level, each place that the subroutine could exit to is assigned the software level of the associated exit. If the calling level is higher-numbered than the subroutine level, the subroutine entrance is reassigned with the level of the call and the new level is propagated through the subroutine (which may entail changes in other subroutines). If the software level of any exit from the subroutine is changed, then every call to that subroutine thus far processed must have the affected exit reassigned and this new level must be propagated to the succeeding instructions. Since such a change can only increase an instruction's software level number (and there are only a finite number of levels), this procedure will eventually terminate. When it does terminate, each subroutine will have been assigned the maximum of the software levels of all of its calls, and the effects of such an assignment will have been propagated back through the calling sequences.

At this point, all executable code should have both hardware and software levels assigned to it. If, at any point, the system has attempted to assign two different software levels to a word (except as a result of reassigning a subroutine), it assigns the maximum. If, at any point, it has attempted to assign a software level larger than the hardware level, then there is a bug.

Now we are ready to verify that all of the data references are bug-free. First, many types of data references are "interrupt-proof"; e.g., an isolated LOAD or STORE reference. "Dangerous" references to variables must be checked, however. For each such reference, the scope of the reference must be identified. Within each such scope, determine the largest software level occurring anywhere within it (not forgetting to trace through any subroutines called). Then verify that the hardware level assigned to it is equal to the derived reference-software-level. Any violation is an error: if the hardware level is less than the software level, then the reference is not sufficiently locked to insure a bug-free reference; if the hardware level is greater than the software level, then the reference is excessively locked and perhaps system performance can be improved by increasing the software level of the reference.

This procedure should verify that there are no interrupt bugs in the control or data structures of a program. The ambiguities (e.g., in determining "dangerous" versus "safe" references) can usually be eliminated by the cooperation of the programmer; for example, the programmer could just use the op-code SSTORE (which would assemble in the same way as a STORE) to indicate a "safe" store, or he could use a different mnemonic for potentially skipping instructions which are known, in the context, never or always to skip.

CONCLUSION

Our approach has been threefold: 1) we have analyzed exactly what properties and constructs within a system could give rise to multiprogramming conflicts; 2) we have written macros for our assembler which note the various protection structures and log any deviations (i.e., protections where none are needed, and portions which require additional protection); and 3) we have designed a groundwork from which a more complete and more automatic system could be built.

Finally, we have extended the theoretical basis of our system so that other types of multiprogramming structures, e.g., multiprocessing and polling systems, can be handled. We have designed the basis of an automatic programming system which would allow the writing of multiprogramming conflict-free programs with min-

imal unnecessary overhead. The rules, although difficult to incorporate into an assembly language, could easily be included in an appropriate higher level language.

ACKNOWLEDGMENTS

This work was supported by the Advanced Research Projects Agency of the U. S. Department of Defense under Contracts F08606-73-C-0027 and F08606-75-C-0032. We are also grateful to a number of our colleagues at Bolt Beranek and Newman Inc.: Stephen Butterfield and Joel Levin, who helped develop the system described; William Mann, who offered helpful criticism; Robert Brooks, who helped prepare the manuscript; and our mentor in programming, William Crowther.

REFERENCES

E.W. Dijkstra (1968), "Cooperating Sequential Processes" in Programming Languages, F. Genuys, Ed., Academic Press, pp. 43-112.

F.E. Heart et al. (1970), "The Interface Message Processor for the ARPA Computer Network," AFIPS Conference Proceedings, Vol. 36, pp. 562-563, section on "Control Organization."

F.E. Heart et al. (1973), "A New Minicomputer/multiprocessor for the ARPA Network," AFIPS Conference Proceedings, Vol. 42, pp. 529-537.

ADDITIONAL BIBLIOGRAPHY

C. Gordon Bell and Allen Newell (1971), Computer Structures: Readings and Examples, McGraw-Hill, pp. 82-83, 123, 267-272, 274-283, 411, 423, 452-453, 458-461, 481-482, 553-555.

D.W. Davies and D.L.A. Barber (1973), Communication Networks for Computers, John Wiley & Sons, pp. 70-77, 465, 467-471, 480, 491-492, 558.

Elliott I. Organik (1973), "Software Interrupts," Computer System Organization -- The B5700/B6700 Series, Academic Press, pp. 63-76; also "Hardware Interrupts," p. 111.

Nicklaus Wirth (1969), "On Multiprogramming, Machine Coding, and Computer Organization," CACM, Vol. 12, No. 9, pp. 489-498.

Edward Yourdon (1972), "The Priority Control Program," Design of On-line Computer Systems, Prentice-Hall, Inc., pp. 394-421.

Discussion on: "Techniques for Detecting and Preventing Multiprogramming Bugs" by B. P. Cosell, J. M. McQuillan, D. C. Walden

Smith: In the last operating system I built, I ran into trouble with a low priority routine which must run every 10 minutes. When the scheduled time was approaching, the operating system gradually increased the priority of this low priority program.

Cosell: That is a classical problem in time-sharing systems. Also at the beginning you said there were 10 minutes to do the job at a low priority, and after 2 1/2 minutes you give the job more priority, after 3 1/2 minutes more, etc. We have not tried these at very slow speed, (on the order of every 3-5 minutes). We have not found the need to have dynamically changing priorities. We are content to live within the linear structure. I do not know exactly how to answer your comments: your task must get done in 10 minutes. I certainly know how to write code to make that happen. But that kind of code is almost impossible if it shares a variable with almost anything else in the system. When the code starts, the higher priority routines are interrupting it, so it has to do all the protection. As it gets more and more priority, it begins to be able to defer the now lower priority routines and they must now protect themselves from it. So it sounds as if in the end your system has to protect everything all the time. We certainly can do that kind of thing but we have not done it thus far.

Bell: One faces a similar but somewhat easier problem in the case of optimizations of disk seeks. There, one wishes to service the shortest seek time first, but without making anyone wait forever to be serviced. The way that we solved that is by having something called a fairness count, which is specified at system-build time. Every nth time (where you select n) the person who has been waiting the longest gets serviced in spite of whatever other rules may apply in the system. When you get multiple levels of priority, the situation becomes more complicated, of course.

Rekdal: More questions?

Matre: I think the problem might be best solved by use of semaphores because then you can explicitly protect single variables, without turning off the whole systne.

Cosell: True, in the kind of problem I am addressing it is

implicit that semaphores are fine. But you have to be careful,unless your machine has an uninterruptable load and clear to zero or some instruction of that variety. You have the problem of just how to manipulate the semaphores. If you want to read and take the semaphore, you have to read it, make sure you can get it, and if you decide to take it, it may be too late. Somebody may have already taken it. You need some way of guaranteeing that you are protected. Between the time when you look at the semaphore, and the time that you decide that yes you can proceed.

Matre: Okay, but you can turn off the interrupts exactly for the few instructions when you change the semaphore.

Cosell: Assuredly, as for example I did right here in my little example. Instead of doing the kind of thing I am referring to, you can determine the absolute level to which you want to activate the interrupts. Then you go to whatever pain it takes to only deactivate to level 3 leaving levels 2 and 1 active. If your program is organized carefully, you can usually arrange to protect a small number of instructions. It is a negligable system degradation to turn off the entire interrupt system for a few microseconds. That can barely hurt anything.

But, nonetheless, this particular kind of technique can identify which portion of code actually require lock instructions.