Interrupt mechanism for threaded code interpreter

Alexey A. Burtsev,
Obninsk, Russia,
bur@iate.obninsk.ru

Abstract

Various methods to support interrupt mechanism in programming systems, based on threaded code techniques, are analysed. Threaded code interpreters, which support interrupt handling at the level of machine code, have serious demerits. Another method of interruption for interpreter of threaded code are suggested. It makes possible to interrupt program only at the level of threaded code. It means, that body of interrupt handler will be executed not at the same moment, when the interruption is raised, but just after that moment, when the interpreter completes execution of its current command (primitive procedure, which body are defined in machine code).

Realization of interrupt mechanism according to this suggested method for DSSP (Forth-like system, developed in Moscow University) are described. Advantages of new interrupt technique for threaded code interpreter, especially concerning parallel programming and exception handling, are explained.

Introduction

Dialogue programming systems [1-4], based on threaded code interpretation technique and named Forth-like systems, combine high performance and code compactness of interpreted program with benefits of high-level programming language. So such systems can be applied in real-time programming.

Functionality of any Forth-like system is provided by threaded code interpreter [5,6]. For real-time application this interpreter have to be able to react qwickly on external and internal events so that to switch control immediately to the program block, handling occured event. It means, that such interpreter needs in interrupt mechanism like that hardware processor has.

Moreover, it is desirable for Forth-like system to have possibility to define interrupt handler in terms of the same level language as used for interpreted program. It means, that body of interrupt handler have to be executed not only as machine code, but can be interpreted as threaded code.

Let's analyse various methods, which give the threaded code interpreter such important capability to make interruption of interpreted program.

1. Threaded code interpreter interrupt mechanism at the level of machine code

First of all, let's examine an usual technique, which permits to use built-in hardware interrupt mechanism of base processor for interruption of interpreted program. For demonstration of this technique it is necessary at first to review main algorithms of threaded code interpreter operations.

1.1. Threaded code structure and its interpretation

Some varities of threaded code are known [7] : direct, indirect, token-indirect. For the demonstration let's use that variant of direct threaded code, which is applied in DSSP-32p, protected mode version of DSSP [2,3] (Dialogue System of Structured Programming, developed in Moscow University) for Intel 386 processor.

To describe main interpreter algorithms, using high-level notation, assume, that :

AP - arithmetic stack (operand stack) pointer ,

AT - top of arithmetic stack ,

CP - control stack (return stack) pointer,

CT - pointer to next location in threaded code body, which is saved as top of control stack during interpretation of new called procedure,

are appointed to base processor registers. And CP occupies system stack pointer register (SP), the same register, that used for saving return addresses in machine command CALL . So that control stack of interpreter becomes the same as system stack of base processor.

Also assume, that other machine registers, used to store temporary values, are referred as R0,R1,...,Rn ; machine memory is array M[0..Max]; M[A] means one memory location at adress A and it is considered as 32-bit value.

Let's agree to use for algorithm description: identifier with colon as label , assign statement as data transfer machine command, goto and call statements as control transfer machine commands. For example :

```
Adr: R1:=M[A];M[A]:=R2;R2:=R1; goto Adr; { jump to label Adr }
     if R1=0 then goto (R2); { if R1 equals zero then }
 {jump to location, which address is contained in register R2}
```

Assume, that arithmetic stack grows in order to increase address, and control stack grows in inverse order so that they both go to meet each other. Let's declare main operations on stacks as macros :

```
MACRO APUSH(X): M[AP]:=X; AP:=AP+1; ENDM
MACRO APOP(Y) : AP:=AP-1; Y:=M[AP]; ENDM
MACRO CPUSH(X): CP:=CP-1; M[CP]:=X; ENDM
MACRO CPOP(Y) : Y:=M[CP]; CP:=CP+1; ENDM
```

Using given assumption, examine main interpreter algorithms :

```
MACRO NEXT  : R2:=M[CT]; CT:=CT+1; goto (R2); ENDM
MACRO IBEGIN: CALL Interp; ENDM
      Interp: CPOP(R1); CPUSH(CT); CT:=R1; NEXT;
       IEND : CPOP(CT); NEXT;
```

IBEGIN operation is applied as header of threaded code body, which is formed as result of compilation of new procedure, declared like :

```
: <new_procedure_name> <word> ... <word> ;
```

IBEGIN operation, realized here as macros, is intented to start interpretation of new procedure body. For this action, it pushes CT in control stack and assign to CT such new value, that points to the begining of following threaded code. And then NEXT operation is executed.

NEXT operation, also realized as macros, is used at the end of every procedure, defined in machine code. It jumps to body of next procedure, which reference is placed in following location of current threaded code body. Exactly CT points to this location. Thus, NEXT operation is intented to continue interpretation.

IEND is completion interpretation operation. Its reference have to be placed in the end of every interpreted threaded code body. This operation restores previous value for CT, poping it from control stack and then performs NEXT action.

This three operations (IBEGIN, NEXT, IEND) are fundamental operations of threaded code interpreter. Together they make up the basis of its functionality.

1.2. Threaded code interpretation on interrupt request

To develop real-time program in FORTH-like system it is desirable to support possibility to form body of interrupt handler procedure in threaded code. It means, that threaded code body of interrupt handler have to be interpreted at any moment, at any location, where program may be interrupted.

Now we'll try to explain, how such possibility was supported in DSSP-32p before. Body of procedure to be interpreted as interrupt handler was being formed with special header (let's name it interrupt header) by means of INT command before procedure compilation :

```
INT : IHANDLER P1 ... PX ;
```

Then interrupt procedure IHANDLER was appointed to interrupt vector V by means of command :

```
V LINK IHANDLER
```

Interrupt header consists of machine code for calling subroutine from address IntHndl. IntHndl block (machine code, begining at the address IntHndl) is intented to start threaded code of the body, address of which was placed in the top of control stack. Let's examine algorithm of IntHndl-block and explain its main actions :

```
IntHndl:            { Control stack CS: RA,BA }
    CPUSH(R2); ... CPUSH(Rn);      {CS: RA,BA,R2,...,Rn }
    R2:=M[CP+n-1]; { R2:=BA, address of interrupt handler body }
    M[CP+n-1]:=R1;{R1 replaces BA} {CS: RA,R1,R2,...,Rn }
    CPUSH(AT);CPUSH(AP);CPUSH(CT); {CS: RA,R1,R2,...,Rn,AT,AP,CT }
    CT:= adr(ARetInt); { CT:= address of body ARetInt }
    goto (R2);
ARetInt: adr(RetInt) { address of body RetInt }
RetInt :           { Control stack CS: RA,R1,R2,...,Rn,AT,AP,CT }
    CPOP(CT);CPOP(AP);CPOP(AT);   {CS: RA,R1,R2,...,Rn }
    CPOP(Rn);...CPOP(R2);CPOP(R1);{CS: RA }
    IRETURN { return from interrupt to location RA }
```

When interrupt V occurs at any program location (RA), it calls interrupt procedure from address of the interrupt header, that was appointed to it. At this moment address of interruption location (RA) is pushed in system stack. Then the interrupt header calls subroutine from address IntHndl, pushing address of following body (BA) in system stack. So IntHndl-block begins to execute, having two values in control stack (RA,BA) .

At first, it saves all processor registers (except CP) in control (system) stack, because they may be corrupted during interpretation of interrupt handler body. Address (BA) of the body to be interpreted is poped from control stack to temprorary register (R2) and then is used to transfer control to the body of interrupt handler. Before this, CT is assigned to address (ARetInt) of body, that contains reference to machine code block (RetInt-block), which is intended to finish interrupt handler interpretation. Control will

be transfered to this block, when during NEXT operation value CT points to ARetInt.

At last, RetInt-block will restore previous values of all saved registers, poping it from stack, and then perform interrupt return machine command.

Such technique of interrupt handling (described above) was being applied in DSSP-32p for a long time (until version 4.41). According to this technique, interrupt handler, defined as threaded code procedure, is called at the same time, when interrupt is raised. As a result, interpreted program is interrupted just after termination of current machine command. So such interrupt technique for interpreter of threaded code may be named as interpreter interrupt mechanism at the level of machine code.

This mechanism has some valueable properties. It permits to call interrupt handler quickly and to define the handler as high-level procedure. Unfortunately, it has also some serious demerits.

1.3. Demerits of interrupt mechanism at machine code level

The main uncomfortable habit of interrupt mechanism, desribed above, is demand to construct interrupt handler only as correct subroutine. It means, that the only possibility to leave body of this procedure is to perfom return statement. Other ways to transfer control to any location out of this procedure is forbidden. In the case of violation of this demand normal return to interrupted program can't be assured. So it becomes impossible to use in body of interrupt procedure such desirable operations as raise exception, send signal to parallel process, transfer control to coroutine.

Such strict requirement for construction of interrupt procedure is conditioned by the method, that is applied to invoke interrupt handler. As above algorithms illustrate, termination of interrupt procedure have to be accompanied by specific recovery actions, which modify control stack. So attempt to continue program, escaping this actions, usually leads to program crash.

By the way, many programing systems have similar interrupt mechanism, and mentioned above requirement is an old subject for critics [8]. But for system, based on threaded code interpreter, this requirement enlarges and concerns any operation, which affects not only control stack of interrupted program, but arithmetic stack too.

It is clear, that attempt to modify control stack of interrupted program during execution of interrupt handler procedure can prevent from making normal interrupt return and so this attempt may lead to unforeseen results. But it is not obvious, why interrupt handler procedure haven't to modify arithmetic stack of interrupted program. Let's explain, why such modification may be incorrect because of unpredictability of interrupt moment.

Assume, that our interrupt handler (CLR2) have to clear (assign zero value to) top and suptop of the arithmetic stack of interrupted program every time when interrupt will occur. Suppose, that interrupt is raised during execution of such operation, that performs actions with arithmetic stack, for example, exchanges values of top and subtop (E2 for DSSP or SWAP for FORTH). Let's examine machine code bodies of following procedures :

```
E2a: R1:=M[AP-1];{p1} M[AP-1]:=AT;{p2} AT:=R1; { 1'st variant }
E2b: R1:=AT;{p1} AT:=M[AP-1];{p2} M[AP-1]:=R1; { 2'nd variant }
CLR2: M[AP-1]:=0; AT:=0; NEXT;
```

If interrupt will be raised at the point p1 or p2 (in procedure E2 of any realization variant), then result of interrupt handling by means of procedure CLR2 will be incorrect: either top or subtop of arithmetic stack will retain nonzero value (the same, that will be saved in R1). This is the result of collision between interrupted program and interrupt handler, both attempting to perform actions with common data (arithmetic stack) at the same time.

Such collisions is conditioned by the method of calling interrupt handler just at the moment, when interrupt has been raised. The interpreted program can be interrupted when it has already started to execute primitive operation with arithmetic stack, but hasn't completed it yet. And execution of this current primitive operation, which has machine code body, will be broken in time. So it leads to the same conflicts as in parallel program.

To avoid such conflicts , it is necessary to guarantee, that interpreted program won't be interrupted during execution of any primitive procedure for performing interrupt handler. Such guarantee permits to solve problem of mutual exclusion between interrupted program and interrupt handler, because every primitive operation will be executed as one indivisible action. Let's examine another method of interrupt handling, which ensures this capability.

2. Interpreter interrupt mechanism at the level of threaded code

New interrupt method, which now is described here, for the first time was suggested by author for DSSP-80 [9]. This method permits to execute body of interrupt handler not at the same moment, when the interrupt is raised, but just after that moment, when the interpreter completes execution of its current command (primitive procedure, which body are defined in machine code). Let's explain, how to realize this method.

Unlike previous interrupt method, new method can't be realized without modification of main operations (IBEGIN, NEXT, IEND) of the interpreter. Indeed, in order to provide new method it is necessary to modify algorithm of main interpreter loop so that it became possible to transfer control to the interrupt handler from some point inside the main loop, where usually interpreter terminates current command and starts new one. Note, that such point may be somewhere in NEXT operation.

NEXT operation is the most important operation of interpreter. It is executed every time, when interpreter performs primitive command. In order to provide high performance of interpreted program, NEXT operation must be as fast as possible. By the way, machine code of NEXT operation in previous version of DSSP-32p contains only two machine commands of base processor (Intel 386). How can we modify NEXT operation effectively in order to perform control transfer to the handler, when interrupt occurs ? Let's examine and evaluate the following variants of this modification.

As first variant, we can enable interrupts in the begining and disable interrupts in the end of NEXT operation in order to guarantee, that interrupt can be raised only during NEXT operation. But such variant adds at least two machine command to the NEXT operation. Also it will requre to perform primitive input/output operation without using interrupts.

As second variant, we may insert in NEXT operation conditional statement for checking, if interrupt was already raised or not. For this purpose, additional logical variable (IntFlag) can be set, when

interrupt occurs, and then may be checked during NEXT operation. Jump to interrupt handler body can be performed after this checking only if IntFlag became true. But this variant also requires at least two additional machine commands for NEXT operation.

At last, we can propose the variant, which adds only one machine command to NEXT operation. First of all, let's explain its essence. Suppose, that we have modified macrooperation NEXT so that it contains only one machine command to perform indirect jump, using content of additional register or memory word (named ITNext). Let ITNext point to previous machine code of NEXT operation while interpreter functions without any interrupts. But every time, when interrupt occurs, ITNext will be changed in order to point to machine code, which have to start interrupt handler. Thus, now in order to jump to body of its next command (primitive procedure), interpreter have to perform one new machine command (indirect jump) and then to execute machine code of previous NEXT operation (that consists of two Intel-386 machine commands for DSSP-32p).

Because this last variant is the most effective among examined above, it was applied in DSSP-32p as new interrupt method for threaded code interpreter. So let's describe realization of this interrupt mechanism in detail :

```
ITNext - register-pointer, used for indirect jump in NEXT operation
   ITNext:= adr(NextI); { during system initialization }
MACRO INEXT: R2:=M[CT];CT:=CT+1;goto (R2); ENDM { previous NEXT }
      NextI: INEXT; { machine code, to which ITNext usually points }
MACRO  NEXT: goto(ITNext); ENDM {new NEXT operation for primitives }
new machine code of interrupt header : CALL IntrSave
IntrBuf: { location for saving pointer to interrupt handler body }
IntrSave:                { Control stack CS: RA,BA }
    CloseIntr; { disable interrupts }
    CPUSH(R1); R1:=M[CP+1]; { R1:= BA } {CS: RA,BA,R1 }
    IntrBuf:=R1; { save pointer to interrupt handler body }
    ITNext:= adr(IntrHndl); {address of block to start the handler}
    CPOP(R1); CP:=CP+1;                {CS: RA }
    IRETURN { return from interrupt to location RA }
IntrHndl: { this block is intented to start the interrupt handler }
    R2:= IntrBuf; { get address of interrupt handler body }
    ITNext:= adr(NextI); { restore usual value for pointer ITNext }
    OpenIntr; { enable interrupts }
    goto (R2);{ jump to body of interrupt handler }
```

Subroutine IntrSave is called from machine code of interrupt header, placed before body of interrupt handler. So when this subroutine starts, top of control stack points to interrupt handler body. This pointer is poped from stack to be saved in temporary variable IntrBuf. Before this action all interrupts have been disabled. At last, subroutine IntrSave assigns address IntrHndl to ITNext and performs return from interrupt.

Afterwards, when current primitive procedure will try to execute NEXT operation, then jump to IntrHndl location will be performed. Block of machine code, begining from this location, is intented to start interrupt handler. It gets address of the handler body from variable IntrBuf, restores usual value for ITNext, then enables interrupts and, at last, jumps to the interrupt body, using the address saved in IntrBuf by subroutine IntrSave.

According to suggested method, reaction on interrupt is accomplished in two stages. At first stage (performed by subroutine IntrSave) interrupt occurence is only registered. And then at second stage (started by IntrHndl block) the handler of registered interrupt is really executed. First stage is short, it acts just at the moment, when interrupt is raised, and terminates quickly by

return from interrupt. Second stage starts only after interpreter completes its current command. So interrupt handler can't prevent interrupted program to perform current operation correctly.

Moreover, note, that interrupt handler is executed so as if interrupted program called it in the point just after completed current command. In other words, interrupt handler is executed as usual procedure, but which is called in unpredictable moment. And so this interrupt method permits to use in interrupt procedure any operation , that can be used in usual procedure. Exactly this permission is the most important advantage of suggested interrupt method.

Conclusion

New suggested interrupt method was realized in DSSP-32p since version 4.41. Emloyment experience of new version DSSP-32p testified that general performance of interpreted program didn't decrease significantly, but new capabilities of its interrupt mechanism extended the sphere of its application. In general new version of DSSP-32p became more useful for real-time programming.

Thanks to new interrupt mechanism it became possible in DSSP-32p to use in body of interrupt handler some desirable operations, which was forbidden by old interrupt mechanism. And now we have got opportunity to perform as a reaction on interrupt the following actions: to transfer control to coroutine, to make context switching to another parallel process, to start new process, to send signal to any process, to raise exception in context of current or any other process.

New capabilities of suggested interrupt method , in particular, have been required for development of some program modules at the level of DSSP programming language for supporting parallel processing. Processes are executed by one processor as coroutines. In order to switch processor to next process on timer interrupt , it is necessary to perform coroutine transfer from the inside interrupt handler body. And this has becomed possible only thanks to new method of interrupt handling in DSSP .

Note, that we named previous interrupt method of threaded code interpreter as interpreter interrupt mechanism at the level of machine code. Comparing new suggested interrupt method with previous one, we consider, that new interrupt method deserves to be named as interrupt mechanism for interpreter at the level of threaded code.

References

[1] Leo Brodie. Starting FORTH. An introduction to the FORTH language and operating system for beginners and professionals. Prentice-Hall, 1981.

[2] Brusentsov N.P., Zaharov V.B., Rudnev I.A., Sidorov S.A. Dialogue System of Structured Programming DSSP-80. In book "Dialogue microcomputer systems", Moscow State University, 1986, pp. 3-21. (in Russian)

[3] Shumakov M.N., Sidorov S.A. DSSP and Forth : A comparative analysis. Proceedings of the EuroFORTH'96 Conference, 1996.

[4] Motalygo Valo G. PS - a FORTH-like threaded language.- BYTE, 1981, v.6, No.10, p.462.

[5] Bell J.R. Threaded Code. - Communication of the ACM, 1973, v.16, No 6., p.370-372.

[6] Brusentsov N.P. Structured programming and threaded code. In book " Arhitecture and software of digital systems", Moscow University, 1984, pp. 3-9. (in Russian)

[7] Ritter T., Walker G. Varieties of threaded code for language implementation.- BYTE, 1980, v.5, No.9, p.206.

[8] Hunt J.G. Interrupts.- Software: Practice & Experience, 1980, v.10, No.7, p.523-530.

[9] Burtsev A.A. Peripheral monitor as development of DSSP input/output architecture. In book "Dialogue microcomputer systems", Moscow State University, 1986, pp. 42-51. (in Russian)