1057

Bell Laboratories

ect: Calling UNIX System Entries
From Fortran or
Going to Sleep in Fortran

date: June 25,1975

from: I. A. Winheim
MF-75-8234-30

ABSTRACT

A new UNIX routine has been written that provides an interface to the UNIX timeout mechanism. With this mechanism a Fortran process may suspend execution for a specified number of seconds by the following sequence:

```
call sleep(seconds)
```

This memorandum will describe how this routine works, as well as describing in detail how a user written assembly language subroutine interfaces to Fortran.

Bell Laboratories

subject: Calling UNIX System Entries From Fortran or Going to sleep in Fortran

date: June 25, 1975

from: I. A. Winheim MF-75-8234-30

MEMORANDUM FOR FILE

I. Introduction

A new Fortran routine has been written that provides an interface to the UNIX timeout mechanism. With this mechanism a Fortran process may suspend execution for a specified number of seconds by the following sequence:

```
call sleep(seconds)
```

This memorandum will describe how this routine works, as well as describing in detail how a user written assembly language subroutine interfaces to Fortran.

II. Functional Description

The interface to the UNIX timeout mechanism that puts a process to sleep in Fortran is executed by using the call:

```
call sleep(seconds)
```

The call has essentially the same effect as the existing calls in C and assembly language. The process calling sleep halts for a specified number of seconds. The number of seconds must be a single precision unsigned integer (16 bits) that may not exceed 65,535 seconds (18.2 hours).

The similarities between the call to sleep in C, in assembly language, and in Fortran end here. The reason for this is tnat Fortran cannot interface directly with the operating system. A factor in preventing a direct interface is the way storage is allocated. The C language uses a 16 bit word as the basic precision for integer variables. Fortran uses two 16 bit words for integer variables and possesses a whole class of precisions which the C language does not have. Another reason that Fortran cannot interface directly with the operating system is that Fortran runs interpretively and special linkage conventions are used. An assembly language program is necessary to interface to the system.

The assembly language subroutine for the sleep interface is shown in Figure 1.

The assembly language subroutine for the sleep interface is in a special format for an interface to Fortran (see discussion below). The code essentially picks up the low order 16 bits of the argument passed to it. (Fortran uses two 16 bit words for integer values while C uses only one). This is accomplished by placing the contents of register r3, which by convention points to an Argument List, into register r1. This address is then incremented to the second entry in the Argument List (this entry contains the address of the argument being passed to sleep). This address is then placed in register r1 and incremented making r1 point to the second half of the double word integer argument. This value is then placed in r0 where the system expects to find it and a "sys sleep" is executed. This causes a trap to the UNIX operating system where the trap handler provides an entry point directly into the system through the System Entry Point Table. The system sleep interface is the thirty-fifth (35) entry in that table (SYSENT) and must be defined in the subroutine.

III. General Description of how Fortran Works

When a Fortran program is compiled and loaded the result is an executable file, a.out, which contains the Fortran Interpreter, an Interpretation List (list of primitive routines to interpretively execute the Fortran statements), and any needed assembly language subroutines (see Figure 2). If several Fortran fragments are compiled and loaded together they will each have a separate Interpretation List. A simple Fortran program having one call to sleep is shown in Figure 3.

The Interpretation List produced for this Fortran program is shown in Figure 4 and consists of a table of addresses which specify to the Interpreter the subroutines and their arguments to be executed to simulate the Fortran. This set of subroutines implement the primitive operations of addition, subtraction, assignment, and call.

The Interpretation List (called "main:" in this case because it is created from a main program rather than subroutine) can be divided into three parts. The first section is the list of primitive operations to be performed to interpret the Fortran (Section A of Figure 5). The next section contains the compile time constants and variable space (Section B of Figure 5). The last section (not shown in this example because a "main" Interpretation List does not return values) consists of space required within a subroutine Interpretation List for linkage and for returning values.

There are three types of entries in the Interpretation List, two for primitive operations (operations that do not require another more basic operation to function) and one for a subroutine call.

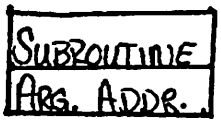
Figure 6 shows the formats of the three types of entries. There are two types of entries for primitive operations, one for monadic (single operand) and one for dyadic (double operand) operations. The first type involves fetching or storing arguments on the stack or in memory respectively(monadic), and the second involves binary (dyadic) operations. The Interpretation List for fetching or storing primitive operations consist of the address of the assembly language subroutine performing the move and either the address of the argument if the argument is being fetched from memory or the destination if the argument is being stored in memory. The Interpretation List format for a binary operation consists of the address of the assembly language subroutine implementing the operation. There are no arguments associated with the binary operations because the stack is used as a group of accumulator registers. The operation is therefore performed on the last two entries in the stack and the result is put back on the stack.

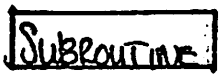
The subroutine call format is more complex, and is also shown in Figure 6. The first argument is the address of the assembly language subroutine implementing the call primitive. The second argument is the address of the subroutine to be called. (See below for the format of a subroutine.) The third and fourth arguments give the location and length respectively of an Argument List which contains the parameters to be passed in the call. The final argument indicates the number of bytes to be returned from the called subroutine (a function of the type and precision of the returned argument.)

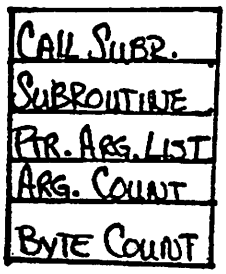
The format required for all Fortran subroutines is shown in Figure 7. The entry point must be the name of the subroutine with a period appended to the end. The first word of the subroutine is the address of an area for returning values. The second word of the subroutine must contain the address of the real entry point of the subroutine (address of the executable code). It is this entry that enables linking to the Interpretation List. Figure 8 shows the sleep subroutine with the entries labelled. No value is returned by sleep but a pointer to a returned value and space for the value are reserved.

IV. Conclusion

UNIX Fortran, because it uses an Interpretation List and must execute assembly language subroutines for each statement executes slower than other "real" compilers. Interfaces to the system, or for that matter to code written in another language, must be done through an assembly language interface. The linkage conventions, the Intertretation List, and the different precisions of variables are unique traits of this Fortran, and because of these differences the libraries are not mixable.







EXAMPLES:

RVAL4; C0

IAD4;
(integer add 4 bytes)

CALL; SLEEP.; F+0.; 1.;0

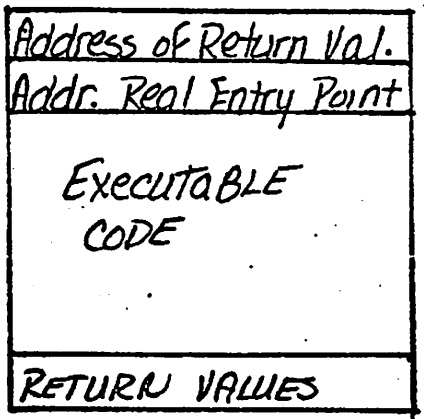
PRIMITIVE OPERATION FORMATS

FIG. 6

ENTRY POINT :

REAL ENTRY PT:

bss :



FORMAT OF A SUBROUTINE

FIG. 7

```
sleep = 35.
.globl  sleep.
.globl  retrn
```

GLOBAL DEFINITIONS

```
sleep.:
        horse
        .+2
        mov     r3,r1
        tst     (r1)+
        mov     (r1),r1
        tst     (r1)+
        mov     (r1),r0
        tst     r0
        beq     zero
        sys     sleep
zero:   jmp     retrn
.bss
horse:  .=.+4
```

POINTER TO RETURN VALUES

POINTER TO REAL ENTRY POINT

EXECUTABLE CODE

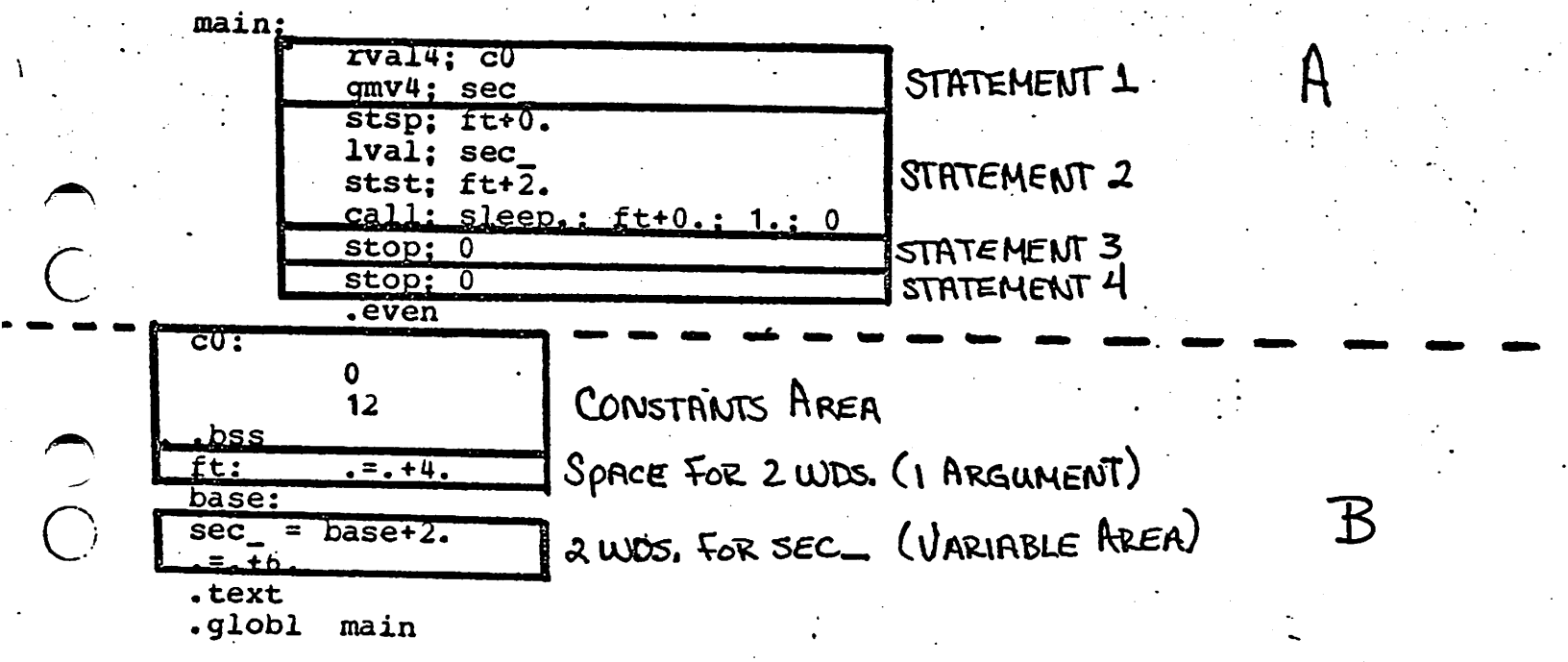
RESERVED SPACE FOR RETURN VALUES

SLEEP SUBRUOTINE

FIG. 8

```
main:
        rval4; c0
        gmv4; sec_
        stsp; ft+0.
        lval; sec_
        stst; ft+2.
        call; sleep.; ft+0.; 1.; 0
        stop; 0
        stop; 0
        .even
c0:
        0
        12
.bss
ft:     .=.+4.
base:
sec_ = base+2.
.=.+6.
.text
.globl  main
```

INTERPRETATION LIST
FIG 4

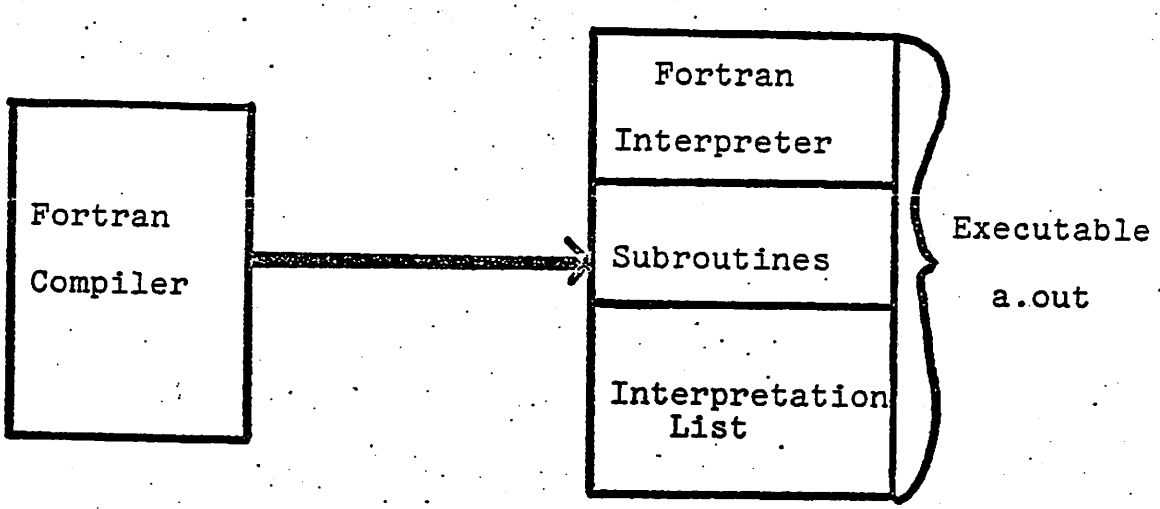


INTERPRETATION LIST
FIG 5

```
sleep = 35.
.globl  sleep.
.globl  retrn

sleep.:
        horse           / this subroutine goes through two
        .+2             / levels of indirection before
        mov     r3,r1   / the actual argument is reached
        tst     (r1)+   / the argument is put into r0
        mov     (r1),r1 / and tested for zero (error)
        tst     (r1)+
        mov     (r1),r0
        tst     r0
        beq     zero
        sys     sleep
zero:   jmp     retrn
.bss
horse:  .=.+4
```

SLEEP SUBRUOTINE
FIG 1



Result of Compile & Load
Fig 2

```
integer sec
        sec = 10
        call sleep(sec)
        stop
        end
```

FORTRAN PROGRAM
FIG 3

For more detailed information on UNIX Fortran see "A Description of How UNIX Fortran Works" by I. A. Winheim TM-75-8234-3.

V. Acknowledgement

I would like to thank T. M. Raleigh for his help in unravelling how UNIX Fortran works.

I. A. Winheim

I. A. Winheim

MH-8234-IAW-nroff

atts.:
Figures 1-9

Copy (with att.) to
MERCURY Category - UNSU
Department 8234
G. L. Baldwin
B. A. Tague