Subroutine calls in the ancient world, before computers had stacks or heaps

devblogs.microsoft.com/oldnewthing/20240401-00

April 1, 2024

Raymond Chen

We take stacks and heaps for granted nowadays, but back in the very old days of computing, computers operated without a stack or a heap.

Tell a recent college graduate this, and you may as well tell them that there was a time when you didn't have instant access to millions of cat videos.

It's not too hard to imagine computing without dynamic memory allocation. You just have to use fixed-size memory buffers for everything. If you have to operate on variable-sized data, you reserved a fixed-size buffer of some capacity that is large enough to accommodate any data you would reasonably be expected to process, and if somebody asked for more, you just exited the program with a fatal error. If you were really nice, you would provide a compile-time configuration so your clients could adjust the maximum capacity to suit their datasets. And if you were really fancy, you wrote a custom allocator that operated on that fixed-size buffer so people could "allocate" and "free" memory from the buffer.

But operating without a stack? How did you call a function if you didn't have a stack for the return address or local variables?

Here's how it worked.

First, the compiler defined a secret global variable for each inbound function parameter, plus another secret global variable for each function to hold the return address. It also defined a secret global variable for each of the function's local variables.

To generate a function call, the compiler assigned the parameter values to the corresponding secret global variables, assigned the return address to the function's secret "return address variable", and then jumped to the start of the function.

The function read its parameters from its secret global variables, and used the pre-defined secret global variables that corresponded to its logically local variables. When the function was finished, it jumped to the address held in the function's secret "return address variable."

For example, suppose you had code like this, written in a C-like language:

```
int add_two_values(int a, int b)
{
    int c = a + b;
    return c;
}

void sample()
{
    int x = add_two_values(31415, 2718);
}
```

This would be transformed by the compiler into something like this:

```
int a2v_a;
int a2v_b;
int a2v_c;
void* a2v_retaddr;

int add_two_values()
{
    a2v_c = a2v_a + a2v_b;

    return_value_register = a2v_c;
    goto a2v_retaddr;
}

int sample_x;
void sample()
{
    a2v_a = 31415;
    a2v_b = 2718;
    a2v_retaddr = &resume;
    goto add_two_values;
resume:
    sample_x = return_value_register;
}
```

Check it out: We did a function call and return without a stack!

Now, you can optimize the ABI by, say, passing some of these values in registers rather than globals. For example, most processors had a special “link” register and a special instruction “branch with link” that automatically set the link register equal to the address of the instruction after the “branch with link” instruction, And maybe you optimize the calling convention to pass the first two parameters in registers, resulting in this:

```
int a2v_a;
int a2v_b;
int a2v_c;
void* a2v_retaddr;

int add_two_values()
{
    a2v_a = argument_register_1;
    a2v_b = argument_register_2;
    a2v_retaddr = link_register;

    a2v_c = a2v_a + a2v_b;

    return_value_register = a2v_c;
    goto a2v_retaddr;
}

int sample_x;
void sample()
{
    argument_register_1 = 31415;
    argument_register_2 = 2718;
    branch_with_link add_two_values;
    sample_x = return_value_register;
}
```

There was just one catch: You can't do recursion.

Recursion doesn't work because a recursive call would overwrite the return-address variable with the return address of the recursive call, and when the outer call completed, it would jump to the wrong place.

The programming languages of the day solved this problem by simply declaring it illegal: They didn't support recursion.[1]

Bonus chatter: Some compilers were even sneakier and used self-modifying code: The special return-address variable was really the address field of the jump instruction at the end of the function!

This was occasionally not so much a sneaky trick as a practical necessity: The processor might not support indirect jumps either!

After the practical value of subroutines was recognized, quite a few processors added a subroutine call instruction that worked by storing the return address at the first word of the subroutine, and beginning execution at the second word of the subroutine. To return from a subroutine, you execute an indirect jump through the subroutine start label. (As I recall, some processors stored the return address at the word *before* the first instruction of the subroutine.) Here's what it looked like using a made-up assembly language:

```
add_two_values:
    nop                     ; return address goes here
    add   r1 = r1, r2       ; actual subroutine begins here
    jmp   @add_two_values   ; indirect jump to return address

sample:
    mov   r1 = 31415        ; first parameter
    mov   r2 = 2718         ; second parameter
    bsr   add_two_values    ; call subroutine
    st    sample_x = r1     ; save return value
```

When the CPU executed the `bsr` branch-to-subroutine instruction, it stored the return address into the first word of `add_two_values` (overwriting the sacrificial `nop`) and began execution at the following instruction, the `add r1 = r1, r2`.

[1] FORTRAN initially didn't even support subroutines! Those were added in 1958. And support in FORTRAN for recursion didn't become standard until 1991, and even then, you had to explicitly declare your subroutine as `RECURSIVE`.