Passing Arguments to Functions and Returning Results

So far, our examples have all been void functions with no arguments. Although all legal examples, they have been a bit peculiar in this respect. Clearly that's not the way a C programmer would have written these functions. However, there is a method in this seemingly peculiar approach. It turns out that passing parameters is a little tricky, and requires an understanding of indexing. Now that we t (hopefully) understand indexing from the last chapter, we can look at how parameters are passed to functions, how results are returned from functions, and how local variables are handled.

The ABI (Application Binary Interface)

The ia32 architecture itself does not dictate solutions to the problems we are studying in this chapter. In fact using the instruction set that we have learned, we could work out many different approaches for solving these problems.

However, we certainly need to have the caller of a function and the function itself agree on how parameters should be passed and how results should be returned. If we were writing a complete program in assembly by hand, we could make up any set of rules we liked. We could even use different rules and conventions for each function. However, that would be difficult to keep track of. Furthermore, that won't help the C compiler. Note that the C compiler is called upon to generate calls to a function in one file, and the function itself in another file. The compilation of the two files is performed entirely separately. We could have different rules based on the function prototype, but really it is far simpler to have one set of rules which everyone agrees on. This set of rules is called the *Application Binary Interface* (ABI), and is typically designed as part of the design of the architecture. What we are really discussing in this chapter is precisely this set of rules. So really this is a chapter on the ABI.

Note that it is not only the compiler, but also other tools that rely on the ABI. For example, a useful function in a debugger is to be able to trace the history of calls at the point of a breakpoint (gdb, the debugger we will be using, calls this a back trace). This back trace can also print the values of arguments at each level of call. Clearly the debugger must know the ABI and know that the functions are all following these rules.

Passing Arguments to Functions

Let's start by looking at how arguments are passed to functions. Many different schemes are possible for solving this problem. On many architectures, the convention is to pass at least the first few parameters in registers. That works well if you have lots of registers, since it means that you don't need to store anything in memory. The caller puts the value of the argument in a register, and the function itself can use the value in the register.

However, this obviously does not work for an arbitrary number of parameters, and in any case the ia32 definitely does not have "lots of registers". A common solution for passing extra parameters that do not fit into registers is to place them on the stack before the call. In the case of the ia32 architecture, the convention is to place *all* the parameters on the stack.

Let's look at a specific example. We will take the linked list example from the previous chapter, and modify it to pass parameters as arguments rather than as global variables:

```
struct node { unsigned val; struct node *next; };

unsigned result;

void find (struct node *h, unsigned val) {
  while (h) {
    if (h->val == val) result = 1;
    else h = h->next;
  }
  result = 0;
}
```

Now our **find** function has two parameters. The convention is that these parameters will be placed on the stack in reverse order by the caller, so that when **find** receives control, the parameters are already on the stack. Let's assume that the value in **ESP** before the start of the call is **00100004**. Then by the time find receives control, the stack will look like:

```
Location 00100000  copy of value of argument val
Location 000FFFFC  copy of value of argument h
Location 000FFFF8  return point past call of find   ← ESP
```

The next question to answer is how the function can access its parameters. The answer comes from the previous chapter on indexing. Indeed the reason we have been avoiding functions with arguments up to this point is that we needed to understand indexing before we can understand the addressing of parameters. You will remember that we noted that the *register indirect with offset* addressing mode is particularly useful. Well it is exactly what we need to answer this question. On entry to the function we have **ESP** pointing to the return point, and the arguments are just above it on the stack. That means that we can address the argument **h** (the first argument) at **[ESP+4]** and the argument **val** (the second argument) at **[ESP+8]**. Let's look at the code that gcc generates for the function find:

```
_find:
L2:
        cmp     DWORD PTR [esp+4], 0
        je      L3
        mov     eax, DWORD PTR [esp+4]
        mov     eax, DWORD PTR [eax]
        cmp     eax, DWORD PTR [esp+8]
        jne     L4
        mov     DWORD PTR _result, 1
        jmp     L5
L4:
        mov     eax, DWORD PTR [esp+4]
        mov     eax, DWORD PTR [eax+4]
        mov     DWORD PTR [esp+4], eax
```

```
L5:
        jmp     L2
L3:
        mov     DWORD PTR _result, 0
        ret
```

If we read this remembering that **[esp+4]** is referencing **h**, and **[esp+8]** is referencing **val**, then the code is quite clear. We can actually persuade gcc to make the assembly language a little clearer to read by using the switch **–fverbose-asm.** Using this switch, the references to parameters are commented in the assembly language, so for example the first few lines of find with that switch set look like:

```
_find:
L2:
        cmp     DWORD PTR [esp+4], 0     # h,
        je      L3       #,
        mov     eax, DWORD PTR [esp+4]   # h, h
```

The character # starts a comment, so the assembler itself ignores the # and all text up to the end of the line. Note that it is a copy of the value of the argument that is passed, not the argument itself, so the function cannot change its arguments. When we see the line:

```
    mov     DWORD PTR [esp+4], eax   # h, <variable>.next
```

which is part of the code generated for h=h->next, the stack location in which we stored the copy of the value of the first argument is being modified, not the argument itself.

Calling a Function with Arguments

We have looked at how the function itself handles the parameters, but now we need to look at how the function gets called. A call in C might look like:

```
struct node *myh;
find (myh, 17);
```

One way to place the parameters on the stack would be to use normal mov instructions, something like:

```
        sub     esp, 8                  # make room for args
        mov     eax, DWORD PTR myh      # get value of
        mov     DWORD PTR [esp], eax    # set argument h
        mov     DWORD PTR [esp+4], 17 # set argument val
        call    _find                   # call find with args
```

The above code would indeed work fine. Note that the **ESP** references are four bytes different from the references in the function itself, for example **val** is stored at **[esp+4]** but in the function is referenced as **[esp+8]**. That's because at the time we are storing the

arguments, we have not yet executed the call instruction, so the return point is not on the stack yet, and **ESP** is four bytes higher.

So let's see if gcc generates something like that for the call. We are going to add the switch **-mpreferred-stack-boundary=2** for now, to simplify this output. We will explain this switch in a moment. With this switch, the call generates:cc n

```
push    17
push    DWORD PTR [esp+4]
call    _find
add     esp, 8
```

Hello! That's not what we expected at all. Actually it is far simpler, and introduces the **push** instruction, which is exactly intended for this function. The effect of the **push** instruction is to subtract four from the stack pointer (without changing the flags), and then to store the source (second operand), which can be a constant, or a memory location or a register, at **[esp]**. That's exactly what we need for passing parameters, and of course the name of the instruction **push** makes perfect sense, given that **ESP** is the stack pointer. The operation of this instruction corresponds exactly to the abstract notion of pushing a value onto the stack. You will notice that we push the arguments in reverse order. That's required by the ABI and for now we just accept it as a (somewhat peculiar) rule. Later on we will see that there is method to this apparent idiosyncratic decision.

There is one more unexplained instruction in the calling sequence generated by gcc, which is the **add esp,8** instruction after the call. A little thought indicates why this is necessary. After we get back from the call, the function has removed the return point, but not the parameters, so, using the example earlier in this chapter, the stack after the call looks like:

```
Location 00100000  copy of value of argument val
Location 000FFFFC  copy of value of argument h      ← ESP
```

These stack locations are of no further interest to the caller, especially since the values stored there may have been clobbered by the function. But we can't just leave them there. If we don't remove them they waste eight bytes on the stack. That's a bad idea. If the call appeared in a loop, then each time through the loop, we would accumulate eight bytes of junk on the stack, eventually blowing the stack space. Furthermore, we depend on the value of **ESP** for addressing our own arguments, so we can't deal with it changing. The add instruction restores the status quo, returning **ESP** to its value before we started the call, and so execution can continue with this original **ESP** value.

We actually have to be a little careful that the push instruction modifies the value of ESP. Consider the following function definition in C, where we have one function calling another and both functions have arguments:

```
void add3 (unsigned a, unsigned b, unsigned c);
void adder (unsigned c) {
  add2 (c,c,c);
```

```
}
```

Concentrating our attention on the function adder, it has one parameter, c, which we would expect to be addressed at [esp+4], but the actual code generated by gcc for this function is:

```
_adder:
        push    DWORD PTR [esp+4]
        push    DWORD PTR [esp+8]
        push    DWORD PTR [esp+12]
        call    _add2
        add     esp, 12
        ret
```

As we push the parameters onto the stack, **ESP** is getting modified, and our argument is getting further away. That's not a problem as long as the compiler (or human programmer in the same situation) takes care to keep track of how many items have been pushed onto the stack and adjusts the references to **ESP** appropriately.

Local Variables in Functions

The next topic to address is the method of handling local variables. These are variables defined within a function. The idea is to allocate storage for these variables on entry to the function, and release the storage on return.

Let's rewrite our example a bit to use a local variable:

```
struct node { unsigned val; struct node *next; };

unsigned result;

void find (struct node *h, unsigned val) {
  struct node *p;
  p = h;
  while (p) {
    if (p->val == val) result = 1;
    else p = p->next;
  }
  result = 0;
}
```

We need to find a place to store the local variable p. The way this is handled is to create a stack frame immediately under the return point on the stack. This stack frame is a fixed length area that holds our local variables. When we leave the function, the stack frame will be removed. For this particular function, we want to set up the stack so that by the time we start to execute the code (starting at p=h), it will look like:

```
Location 00100000  copy of value of argument val
```

```
Location 000FFFFC  copy of value of argument h
Location 000FFFF8  return point past call of find
Location 000FFFF4  value of local variable p        ← ESP
```

Note that this means that the arguments are now four bytes higher than they used to be, so now **h** will be addressed as **[esp+8]** and **val** will be addressed as **[esp+12]**. The return point is at **[esp+4]**. The local variable **p** is addressed as **[esp]**. Here is the code generated by gcc for the modified function:

```
_find:
        sub     esp, 4
        mov     eax, DWORD PTR [esp+8]
        mov     DWORD PTR [esp], eax
L2:
        cmp     DWORD PTR [esp], 0
        je      L3
        mov     eax, DWORD PTR [esp]
        mov     eax, DWORD PTR [eax]
        cmp     eax, DWORD PTR [esp+12]
        jne     L4
        mov     DWORD PTR _result, 1
        jmp     L5
L4:
        mov     eax, DWORD PTR [esp]
        mov     eax, DWORD PTR [eax+4]
        mov     DWORD PTR [esp], eax
L5:
        jmp     L2
L3:
        mov     DWORD PTR _result, 0
        add     esp, 4
        ret
```

The instruction **sub esp,4** at the start is allocating the stack frame by moving **ESP** down to make room for the variable **p**. From then on the code of the function is straightforward if we remember how the stack is being addressed (keeping a picture in mind of the stack layout is very helpful). For example the two instructions:

```
        mov     eax, DWORD PTR [esp+8]
        mov     DWORD PTR [esp], eax
```

Load the value of parameter **h** and stores the result into the local variable **p**. If there were more than one local variable, the size of the stack frame would be adjusted appropriately. For example, if we had seven unsigned variables, requiring 28 bytes, then the initial instruction would subtract 28 from **ESP**.

The only other new instruction is the **add** instruction just before the return. That's undoing the allocation of the stack frame by removing it from the stack. The constant appearing in

this final **add** will of course be the same as the constant that appeared in the initial **sub**, so that everything stays consistent.

Aligning the Stack Boundary

Since the push and call instructions always subtract four from the stack pointer, the ESP value stays aligned on a four byte boundary if we start it out that way. That's good because although in the ia32 architecture it is permissible to access multi-byte unaligned values in most cases, it is not efficient. So if we execute an instruction like:

```
mov     eax, DWORD PTR [esp+4]
```

it will work fine no matter what value is in ESP, but it will work more efficiently if ESP is kept on a four byte boundary, which will always be the case.

However, there are other instructions in the instruction set that require even more alignment for efficient operation, and even some instructions that require proper alignment. It turns out that execution will be most efficient if we keep the stack pointer aligned to a 16-byte boundary. That does not happen automatically, the compiler has to generate extra code to achieve this, but it is worthwhile in improved efficiency. The switch **-mpreferred-stack-boundary=x** specifies the required alignment. Here x is the power of 2 corresponding to the required alignment, so x=2 specifies four byte alignment. We used this setting in the previous code to simplify the output, since four byte alignment comes for free without any extra work. But the default alignment in the absence of this switch is 16 bytes, which is a reasonable choice given the efficiency requirements of the machine. If we remove the **–mpreferred-stack-boundary** switch and recompile the call to find in our earlier example, we get:

```
sub     esp, 4
push    17
push    DWORD PTR [esp+4]
call    _find
add     esp, 8
add     esp, 4
```

The purpose of the extra subtract of 4 from **ESP** before the call, and the addition of 4 to **ESP** after the call is to keep the stack sixteen byte aligned. Since we are going to push two arguments (that's eight bytes), and the call will push the return point (that's four bytes), we need four bytes extra to maintain the required sixteen byte alignment.

Similarly when allocating a stack frame for local variables, the compiler will if necessary allocate extra space to maintain the alignment. For our example in the previous section of a function with seven unsigned local variables, requiring 28 bytes, the stack frame would be set to be 32 bytes long, "wasting" four bytes, to maintain the required alignment.

For the rest of our examples in this book, we are going to continue to use the switch to set the preferred stack boundary alignment to four, just to keep things a bit simpler. We won't

be using any of the instructions which require higher alignment, and we don't care that much about efficiency anyway.

Returning Results from Functions

The easiest of these three topics is the returning of results. The procedure is simple. Results are returned in the **EAX** register. There is nothing special in the ia32 architecture that dictates that **EAX** has some special function. However, we do need some convention since obviously the caller and the function must agree. The rule that **EAX** is to be used for this purpose is part of the ABI, and so all functions follow this rule.

As an example, let's rewrite the linked list searcher one more time to use the more natural return statement instead of returning the result in a global variable.

```
struct node { unsigned val; struct node *next; };

unsigned find (struct node *h, unsigned val) {
  struct node *p;
  p = h;
  while (p) {
    if (p->val == val) return 1;
    else p = p->next;
  }
  return 0;
```

This example finally looks reasonable from a C point of view, so now we are at the stage of being able to understand the code for normal C functions. The only difference we expect in the generated code is that instead of storing the result of the function into the global variable result, it will be stored into the EAX register instead. Everything else should remain the same.

So let's see what gcc will generate for this final version:

```
_find:
        sub     esp, 8
        mov     eax, DWORD PTR [esp+12]
        mov     DWORD PTR [esp+4], eax
L2:
        cmp     DWORD PTR [esp+4], 0
        je      L3
        mov     eax, DWORD PTR [esp+4]
        mov     eax, DWORD PTR [eax]
        cmp     eax, DWORD PTR [esp+16]
        jne     L4
        mov     DWORD PTR [esp], 1
        jmp     L1
```

```
L4:
        mov     eax, DWORD PTR [esp+4]
        mov     eax, DWORD PTR [eax+4]
        mov     DWORD PTR [esp+4], eax
        jmp     L2
L3:
        mov     DWORD PTR [esp], 0
L1:
        mov     eax, DWORD PTR [esp]
        add     esp, 8
        ret
```

Hmm! Not quite what we expected. What's going on here? The stack frame has grown to 8 bytes. Why? The explanation is that in unoptimized mode, the compiler has introduced a hidden local variable to hold the result of the function. It is as though we had written the C code as:

```
struct node { unsigned val; struct node *next; };

unsigned find (struct node *h, unsigned val) {
  struct node *p;
  unsigned result;
  p = h;
  while (p) {
    if (p->val == val) result = 1;
    else p = p->next;
  }
  result = 0
  return result;
```

Once we understand this transformation, the code becomes clear again. The stack frame now contains an extra entry for this extra local variable that the compiler has added, so it looks like:

```
Location 00100000  copy of value of argument val   ← ESP+16
Location 000FFFFC  copy of value of argument h     ← ESP+12
Location 000FFFF8  return point past call of find  ← ESP+8
Location 000FFFF4  value of local variable p       ← ESP+4
Location 000FFFF4  local variable to hold result   ← ESP
```

Once we understand this stack frame layout, we can see that the code of the function is essentially unchanged. The only important difference is the final instruction:

```
        mov     eax, DWORD PTR [esp]
```

which copies the result value into the **EAX** register, leaving it there for the caller. If we look at the code for a call to this function:

```
void caller() {
  struct node *myh;
  unsigned result;
  result = find (myh, 17);
}
```

the corresponding code generated for this call by gcc is:

```
_caller:
        sub     esp, 8
        push    17
        push    DWORD PTR [esp+8]
        call    _find
        add     esp, 8
        mov     DWORD PTR [esp], eax
        add     esp, 8
        ret
```

As we can see, the instruction immediately after the call references the result in **EAX** and stores it where it belongs (in this case into the local variable **result** in the caller function).

Finally, let's look at the optimized code for this last version of our find function:

```
_find:
        mov     edx, DWORD PTR [esp+8]
        mov     eax, DWORD PTR [esp+4]
        jmp     L9
        .p2align 4,,7
L12:
        cmp     DWORD PTR [eax], edx
        je      L11
        mov     eax, DWORD PTR [eax+4]
L9:
        test    eax, eax
        jne     L12
        xor     eax, eax
        ret
L11:
        mov     eax, 1
        ret
```

Well optimization certainly helps, we have gone down from 19 instructions with 12 memory references to 12 instructions with only four memory references. That's quite a saving. Of particular interest is that there is no local stack frame in the optimized function. The compiler figured out that it could keep the local variable **p** in a register (**EAX**), so no stack location was required, and the result can be returned directly in **EAX**. Let's trace through the instructions of this final optimized code:

```
_find:
        mov     edx, DWORD PTR [esp+8]
        mov     eax, DWORD PTR [esp+4]
```

What is going on here is that the generated code is loading the argument values into registers. Once loaded, they will stay in registers, and we never need to reference the stack locations again. It sure would be nice if they had been passed in registers in the first place, but the law is the law, and the ABI dictates how we pass parameters.

```
        jmp     L9
```

Here we are jumping into the middle of the loop. The compiler has rearranged the loop so that the test is at the end, so we only need one jump instruction. That means the first time in we have to jump to this test (since the while semantics requires a test the first time through the loop).

```
        .p2align 4,,7
```

That's the incantation to make sure the following label is aligned, improving efficiency of the loop, since we may be doing lots of jumps to that label, and jumps are more efficient if the jump target label is aligned.

```
L12:
        cmp     DWORD PTR [eax], edx
        je      L11
```

That's the head of the loop, and corresponds to the C code testing for equality of the **val** field. We jump out of the loop if equality is found, since it is a bit more efficient to avoid a jump in the normal case where the loop does not terminate. Figuring out the optimal rearrangement of instructions is not a simple matter. The timing details of modern implementations of the ia32 architecture are enormously complicated. That's another reason to leave the job of generating assembly language up to the compiler ☺

```
        mov     eax, DWORD PTR [eax+4]
```

This instruction moves the pointer along the list. The **EAX** register at this stage holds the value of the local variable that was called **p** in the source program.

```
        test    eax, eax
        jne     L12
```

This is the test for p being zero, and it introduces us to one additional instruction in the ia32 architecture. The **test** instruction performs exactly the same operation as a logical **and** instruction except that no result is stored. However, the **ZF** flag will be set to indicate if the result of the and instruction was non-zero. The relation of the **test** instruction to the **and** instruction is thus similar to the relation between the **cmp** instruction and the **sub** instruction. In this particular case, we are and'ing something with itself, and the result can only be zero if the value being and'ed is zero. So this is just a way of testing for zero. The

following instruction does a **je** (equivalent to **jz**) which will jump only if the result was zero. The two instruction sequence is thus equivalent to:

```
        cmp     eax, 0
        je      L8
```

However, the test instruction is slightly preferable, since it does not require an immediate value in the instruction. The test instruction also has many other uses. For example, the following sequence jumps only if the value in **EBX** is odd, and does not modify the value in **EBX**:

```
        test    ebx, 1
        jnz     odd
```

Now that we understand the test instruction, we can go back to the code of our example:

```
        xor     eax, eax
        ret
```

This is the code executed if the loop falls through. It sets a value of zero in **EAX** (xor'ing something with itself gives a zero result, and this is an efficient way of clearing a register to zero), and returns leaving the result in **EAX**.

```
L11:
        mov     eax, 1
        ret
```

This is the branch taken if we find an equal match. It sets a value of 1 (true) in **EAX** and returns leaving this value in **EAX**. Note by the way that we do not bother to align the label **L11**. We only consider it worth wasting the space for aligning labels in the case of loop labels, where we expect multiple uses of the same label.

Functions with Variable Number of Arguments in C

C allows a function to take a variable number of arguments. The current C standard has a special macro VARARGS for this purpose, which allows this to be done in a relatively clean manner, and if you are writing a real C program and need this functionality, you should certainly use the VARARGS macro for the purpose.

However, in traditional C as it was first defined, it turned out that functions with a variable number of arguments could be constructed with virtually no special help from the compiler. Although this approach is deprecated, it is instructive in the context of this chapter, because it is only understandable if you understand the way arguments are passed and referenced.

As an example, let's write a function sum that will sum an arbitrary number of unsigned arguments. The first argument will be the number of remaining arguments to be summed. So for example, we expect

```
sum (4,5,6,1,2)
```

to return a result of 14. Here is C code for the function sum with a driver to show that it works

```
#include <stdio.h>
unsigned sum (unsigned count, unsigned args) {
  unsigned *argptr = &args;
  unsigned sum = 0;
  while (count--) sum += *argptr++;
  return sum;
}
void main () {
  printf ("%d\n", sum (4,5,6,1,2));
  printf ("%d\n", sum (2, 4, 3));
}
```

Running this program generates an output of 14 and 7 as expected. Well, expected by the description of what sum is supposed to do, but the code of the function is pretty obscure. What is going on here? First, a key point is that we pushed the arguments right to left. This means that the argument count is pushed last, no matter how many arguments are passed. For example, in the case of the second call above, the stack frame looks like:

```
Location 00100000  second value to sum = 3          ← ESP+20
Location 00100000  first value to sum = 4           ← ESP+16
Location 000FFFFC  count value = 2                  ← ESP+12
Location 000FFFF8  return point past call of sum    ← ESP+8
Location 000FFFF4  value of local variable argptr   ← ESP+4
Location 000FFFF4  value of local variable sum      ← ESP
```

Looking at this stack frame, we see that the **count** argument can always be referenced at **[esp+12]** regardless of how many arguments are passed. This is critical to the working of this program, and is indeed the fundamental motivation for pushing arguments onto the stack in reverse order. The argument **args** corresponds to the first argument to be summed (which is the last but one argument pushed). This is always at **[esp+16]** allowing it to be easily addressed. In particular, we set **argptr** to point to this first argument, so that **argptr** will initially have the value of **[esp+16]**. The first time through the loop**, *argptr** will access this location, yielding the value 4. Then **argptr** is incremented, which means that it points to the second value to sum (the value in **esp+20**).

Very clever! But very clever usually means rather horrible. And indeed there are two serious objections to this C code. First it is obscure. You really need to understand what is going on with argument passing at the assembly code level to understand it. That makes it a nice example in this chapter, but is a real drawback for real code, where we want to be clear. Second, and more serious, this is highly non-portable code. It makes assumptions about how arguments are passed, and in which direction the stack builds. For example, if arguments are passed in registers, this whole approach breaks down.

Indeed, gcc really does not like this program. If you feed this as a single file into the compiler it complains that the call has the wrong number of arguments. You can get gcc to ignore this, but it's a bit tricky. An easier way to fool gcc is to put the function main in a separate file with no prototype for sum. Then gcc has no way of knowing that it is generating calls to sum with the wrong number of arguments, and goes ahead and generates code that outputs the expected values of 14 and 6.

So it is indeed appropriate that this mechanism has been replaced by the well defined VARARGS language feature. Nevertheless it is rather amazing that this powerful feature was accessible in the original C compilers with no help from the compiler other than the promise to push arguments in reverse order onto the stack.