The compiler can make up its own calling conventions, within limits

devblogs.microsoft.com/oldnewthing/20150128-00

January 28, 2015

Raymond Chen

A customer was confused by what they were seeing when debugging.

> It is our understanding that the Windows x86-64 calling convention passes the first four parameters in registers `rcx`, `rdx`, `r8`, and `r9`. But we're seeing the parameters being passed some other way. Given the function prototype
>
> ```
> int LogFile::Open(wchar_t *path, LogFileInfo *info, bool verbose);
> ```
>
> we would expect to see the parameters passed as
>
> - `rcx` = `this`
> - `rdx` = `path`
> - `r8` = `info`
> - `r9` = `verbose`
>
> but instead we're seeing this:
>
> ```
> rax=0000000001399020 rbx=0000000003baf238 rcx=00000000013c3260
> rdx=0000000003baf158 rsi=000000000139abf0 rdi=00000000013c3260
> rip=00007ffd69b71724 rsp=0000000003baf038 rbp=0000000003baf0d1
> r8=0000000001377870  r9=0000000000000000 r10=000000007fffffb9
> r11=00007ffd69af08e8 r12=00000000013a3b80 r13=0000000000000000
> r14=0000000001399010 r15=00000000013a3b90
> contoso!LogFile::Open:
> 00007ffd`69b71724 fff3        push rbx
> 0:001> du @rdx          // path should be in rdx
> 00000000`03baf158  "`"
> 0:001> du @r8           // but instead it's in r8
> 00000000`01377870  "C:\Logs\Contoso.txt"
> ```
>
> Is our understanding of the calling convention incomplete?

There are three parties to a calling convention.

1. The function doing the calling.

2. The function being called.
3. The operating system.

The operating system needs to get involved if something unusual occurs, like an exception, and it needs to go walking up the stack looking for a handler.

The catch is that if a compiler knows that it controls all the callers of a function, then it can modify the calling convention as long as the modified convention still observes the operating system rules. After all, the operating system doesn't see your source code. As long as the object code satisfies the calling convention rules, everything is fine. (This typically means that the modification needs to respect unwind codes and stack usage.)

For example, suppose you had code like this:

```
extern void bar(int b, int a);

static void foo(int a, int b)
{
  return bar(b + 1, a);
}

int __cdecl main(int argc, char **argv)
{
 foo(10, 20);
 foo(30, 40);
 return 0;
}
```

A clever compiler could make the following analysis: Since `foo` is a static function, it can be called only from this file. And in this file, the address of the function is never taken, so the compiler knows that it controls all the callers. Therefore, it optimizes the function `foo` by rewriting it as

```
static void foo(int b, int a)
{
  return bar(b + 1, a);
}
```

It makes corresponding changes to main:

```
int __cdecl main(int argc, char *argv)
{
 foo(20, 10); // flip the parameters
 foo(40, 30); // flip the parameters
 return 0;
}
```

By doing this, the compiler can generate the code for `foo` like this:

```
foo:
        inc     ecx
        jmp     bar
```

rather than the more conventional

```
foo:
        mov     eax, edx
        inc     eax
        mov     ecx, edx
        mov     edx, eax
        jmp     bar
```

You can look at this transformation in one of two ways. You can say, “The compiler rewrote my function prototype to be more efficient.” Or you can say, “The compiler is using a custom calling convention for `foo` which passes the parameters in reverse order.”

Both interpretations are just two ways of viewing the same thing.

Raymond Chen

Follow