How do 16-bit programs start up?

devblogs.microsoft.com/oldnewthing/20071203-00

December 3, 2007

Raymond Chen

Back in 16-bit Windows, MS-DOS cast a long and dark shadow. The really ugly low-level munging was very much in the MS-DOS spirit. You opened files by setting up registers and issuing an `int 21h`, just like in MS-DOS. Although the interrupt went to Windows instead, Windows maintained the MS-DOS calling convention. Process startup followed the same "real men write in assembly language" philosophy.

All the parameters to a 16-bit program were passed in registers. The entry point to a 16-bit process received the following parameters on Windows 3.1:

AX	zero (used to contain even geekier information in Windows 2)
BX	stack size
CX	heap size
DX	unused (reserved)
SI	previous instance handle
DI	instance handle
BP	zero (for stack walking)
DS	application data segment
ES	selector of program segment prefix
SS	application data segment (SS=DS)
SP	top of stack

Hey, nobody said that 16-bit Windows was designed for portability.

The first thing a 16-bit program did was call the `InitTask` function. This function receives its parameters in registers, precisely in the format that they are received by the program entry point. The `InitTask` function initializes the stack, the data segment, the heap, retrieves and prepares the command line, recovers the `nCmdShow` parameter that was passed to `WinExec`, all the normal startup stuff. It even edits the stack of the caller so that real-mode stack walking works (critical for memory management in real-mode). When `InitTask` is all finished, it returns with the registers set for the next phase:

AX	selector of program segment prefix (or 0 on error)
BX	offset of command line
CX	stack limit
DX	`nCmdShow`
SI	previous instance handle
DI	instance handle
BP	top of stack (for stack walking)
DS	application data segment
ES	selector of command line
SS	application data segment (SS=DS)
SP	edited top of stack

Once `InitTask` returns, the stack, heap, and data segment are "ready to run," and if you have no other preparations to do, you can head right for the application's `WinMain` function. Minimal startup code therefore would go like this:

```
    call    far InitTask
    test    ax, ax
    jz      exit
    push    di      ; hInstance
    push    si      ; hPrevInstance
    push    es      ; lpszCmdLine selector
    push    bx      ; lpszCmdLine offset
    push    dx      ; nCmdShow
    ... some lines of code that aren't important to the discussion ...
    call    far WinMain ; call the application's WinMain function
    ; return value from WinMain is in the AL register,
    ; conveniently positioned for the exit process coming up next
exit:
    mov     ah, 4Ch ; exit process function code
    int     21h     ; do it
```

Why wasn't the application entry point called `main`? Well, for one thing, the name `main` was already taken, and Windows didn't have the authority to reserve an alternate definition. There was no C language standardization committee back then; C was what Dennis said it was, and it was hardly guaranteed that Dennis would take any special steps to preserve Windows source code compatibility in any future version of the C language. Since K&R didn't specify that implementations could extend the acceptable forms of the `main` function, it was entirely possible that there was a legal C compiler that rejected programs that declared `main` incorrectly. The current C language standard explicitly permits implementation-specific alternate definitions for `main`, but requiring all compilers to support this new Windows-specific version in order to compile Windows programs would gratuitously restrict the set of compilers you could use for writing Windows programs.

If you managed to overcome that obstacle, you'd have the problem that the Windows version of `main` would have to be something like this:

```
int main(int argc, char *argv[], HINSTANCE hinst,
         HINSTANCE hinstPrev, int nCmdShow);
```

Due to the way C linkage was performed, all variations of a function had to agree on the parameters they had in common. This means that the Windows version would have to add its parameters onto the end of the longest existing version of `main`, and then you'd have to cross your fingers and hope that the C language never added another alternate version of main. If you went this route, your crossed fingers failed you, because it turns out that a third parameter was added to `main` some time later, and it conflicted with your Windows-friendly version.

Suppose you managed to convince Dennis not to allow that three-parameter version of `main`. You still have to come up with those first two parameters, which means that every program's startup code needs to contain a command line parser. Back in the 16-bit days, people scrimped to save every byte. Telling them, "Oh, and all your programs are going to be 2KB bigger" probably wouldn't make you a lot of friends. I mean, that's four sectors of I/O off a floppy disk!

But probably the reason why the Windows entry point was given a different name is to emphasize that it's a different execution environment. If it were called `main`, people would take C programs designed for a console environment, throw them into their Windows compiler, and then run them, with disastrous results.

Raymond Chen

Follow