保护模式篇——段寄存器

写在前面

此系列是本人一个字一个字码出来的，包括示例和实验截图。**由于系统内核的复杂性，故可能有错误或者不全面的地方，如有错误，欢迎批评指正，本教程将会长期更新。**如有好的建议，欢迎反馈。码字不易，如果本篇文章有帮助你的，如有闲钱，可以打赏支持我的创作。如想转载，请把我的转载信息附在文章后面，并声明我的个人信息和本人博客地址即可，但**必须事先通知我。**

> 你如果是从中间插过来看的，请仔细阅读 **羽夏看Win系统内核——简述** ，方便学习本教程。

看此教程之前，问几个问题，**基础知识储备好了吗？搭建好环境了吗？没有的话就不要继续了。**

🔒 华丽的分割线 🔒

什么是段寄存器

当我们用汇编读写某一个地址时，比如用下面的代码：

```
1 mov dword ptr ds:[0x123456], eax
```

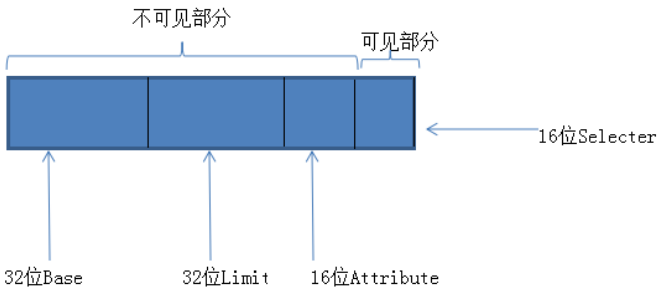
其实我们真正读写的地址是：`ds.base` + `0x123456` 。并不是 `0x123456` ，不过正好的是 `ds` 段寄存器的基址是 `0` 而已。

一些段寄存器

段寄存器有这几个：**ES、CS、SS、DS、FS、GS、LDTR、TR**，它们各有自己特殊的用途。

段寄存器的结构

段寄存器的结构可用下图表示：



段寄存器具有96位，但我们可见的只有16位。我们可以用OD随意加载一个程序，如下图所示：

段寄存器的读写

既然是寄存器了，那就可以进行读写操作，如下将介绍读写段寄存器的操作：

- Mov指令： `MOV AX,ES` ，但只能读16位的可见部分； `MOV DS,AX` 写段寄存器，写的是96位。

- 读写 `LDTR` 的指令为： `SLDT` / `LLDT`
- 读写 `TR` 的指令为： `STR` / `LTR`

段寄存器属性探测

我介绍过段寄存器有 `96位` ，但我们只能看见 `16位` ，那如果证明 `Attribute` 、 `Base` 、 `Limit` 的存在呢？我们将在下面进行初步探测。

段寄存器成员简介

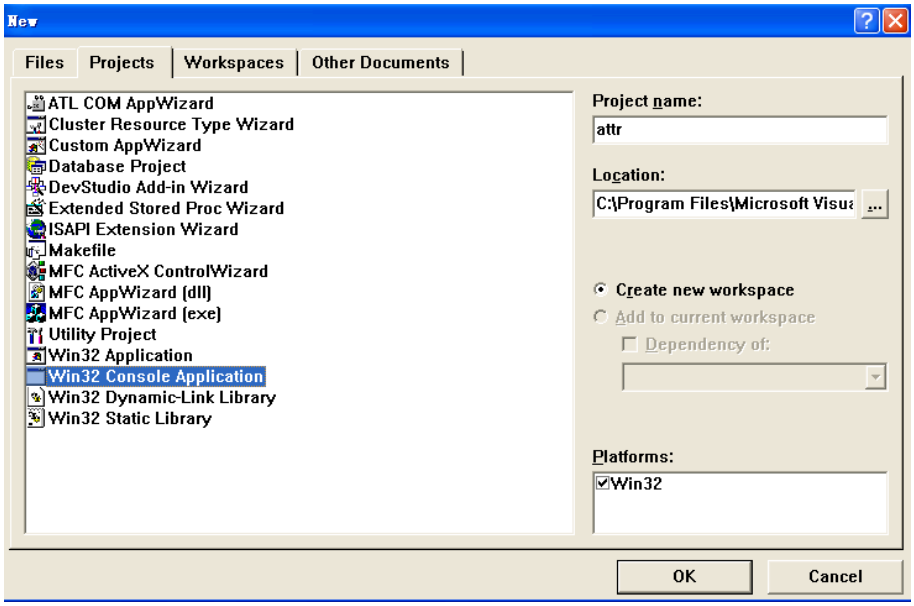
既然讨论段寄存器属性，首先要知道它们存着啥，下面表格的内容是我从虚拟机里查询到的值，可能和我的不一样，但无所谓。它们的属性我已查询并把它们的权限写到表格中，之所以为什么我之后将会介绍。

> Windows操作系统并不会使用 `GS` 寄存器，故用 `-` 表示。

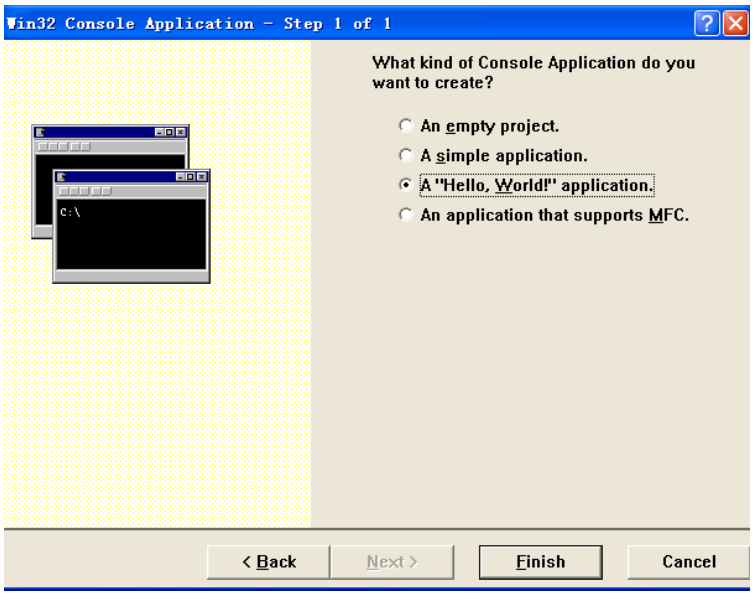
段寄存器	Selector	Attribute	Base	Limit
ES	0023	可读、可写	0	0xFFFFFFFF
CS	001B	可读、可执行	0	0xFFFFFFFF
SS	0023	可读、可写	0	0xFFFFFFFF
DS	0023	可读、可写	0	0xFFFFFFFF
FS	003B	可读、可写	0x7FFDE000	0xFFF
GS	-	-	-	-

探测Attribute

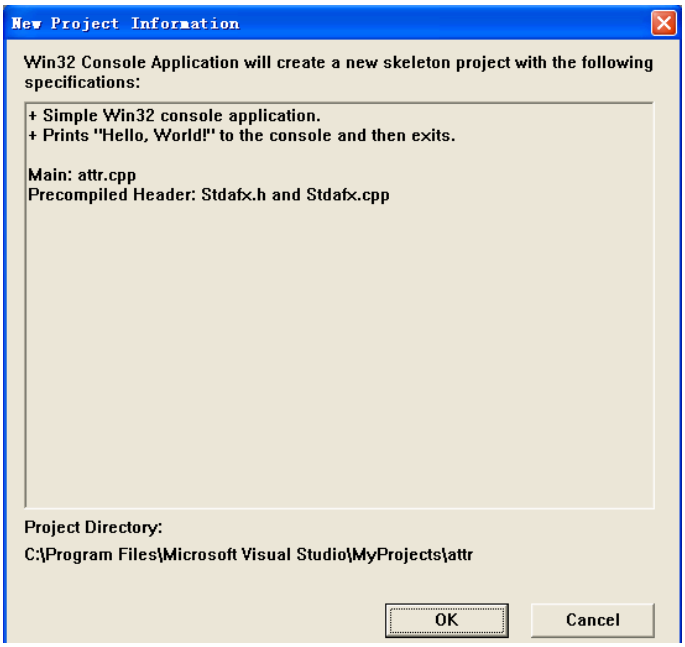
如果你没有使用 `Visual C++6.0` 的话，我先将如何建立工程并写代码简单介绍一下，但只介绍一遍。打开 `Visual C++6.0` ，通过 `File` -> `New` 打开新建项目，选择 `Win32 Console Application` ，输入你的 `Project name` ，如下图所示：



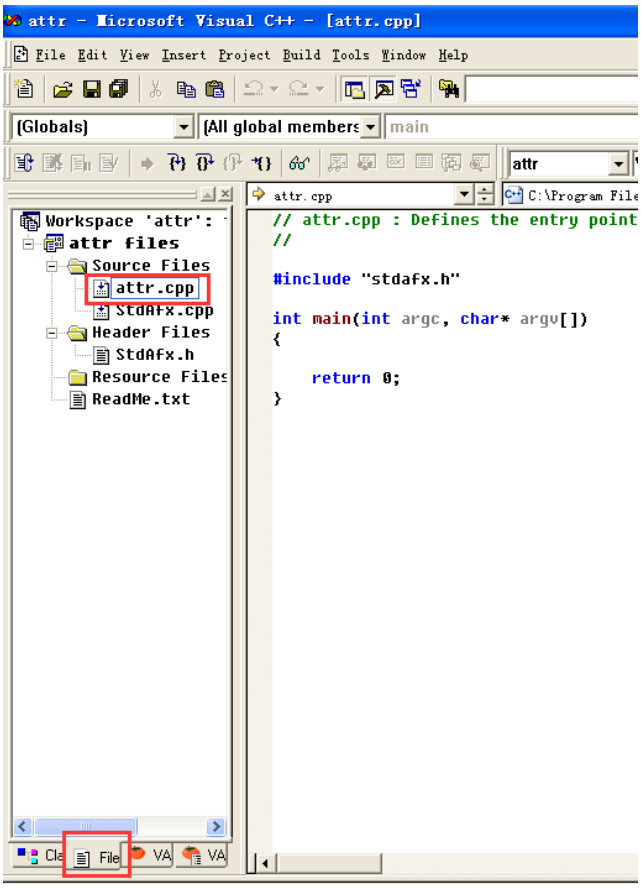
选中带有 `Hello World` 的工程，这样基本上IDE就帮我们建好要做实验的工程了。



然后IDE会展示帮我们新建的内容信息，如下图所示，直接点确定即可，工程新建完毕。



按照下图指示打开源代码文件，删掉用不到的 `printf("Hello World!\n");` 。就能开始做实验了。

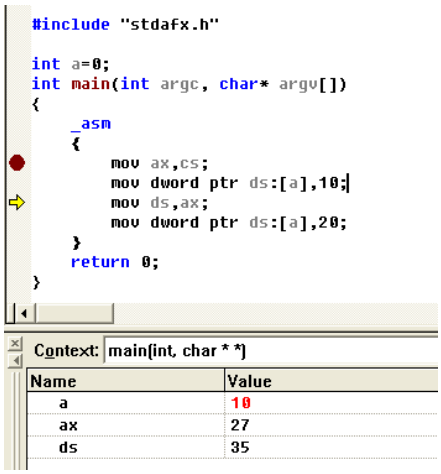


怎么建工程我已详细说明，那就正式进入正题，我们将用以下代码进行验证 `Attribute` ：

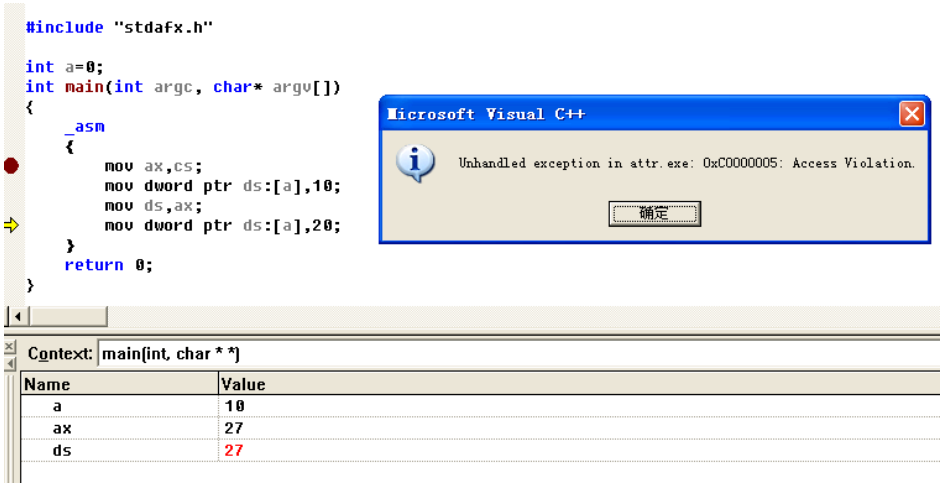
```
#include "stdafx.h"

int a=0;
int main(int argc, char* argv[])
{
    _asm
    {
        mov ax,cs;
        mov dword ptr ds:[a],10;
        mov ds,ax;
        mov dword ptr ds:[a],20;
    }
    return 0;
}
```

然后在 `main` 函数的第一句代码下断点，然后单步运行，运行过第一句给变量 `a` 赋值的汇编代码时，成功通过，如下图所示：



运行到第二个给变量 `a` 赋值的汇编代码时，弹出一个错误信息框，如下图所示



翻译过来就是地址访问冲突，这是什么原因呢？这就是由于cs段寄存器是可读的，而不是可写的。原来的ds是可读可写的，但将 `cs` 通过 `ax` 赋值给 `ds` 时候， `ds` 不再是原来的 `ds` ，而是 `cs` ，故会引发此错误。

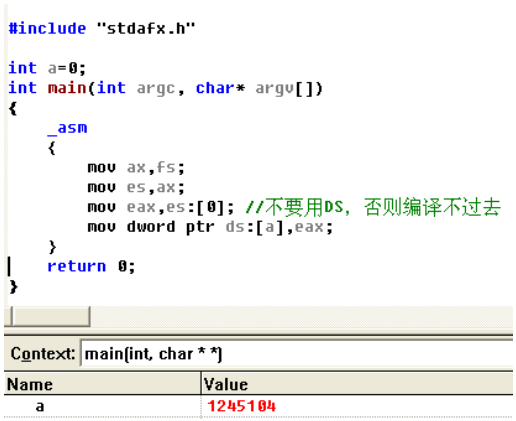
探测Base

老生常谈程序的零地址无法访问。但零地址一定是无法访问吗？我们将用以下代码进行验证 `Base` ：

```
#include "stdafx.h"

int a=0;
int main(int argc, char* argv[])
{
    _asm
    {
        mov ax,fs;
        mov es,ax;
        mov eax,es:[0];        //不要用DS，否则编译不过去
        mov dword ptr ds:[a],eax;
    }
    return 0;
}
```

编译运行通过，变量 `a` 被正常赋值，如下图所示：



探测Limit

我们将用以下代码进行验证 `Limit` ：

```
#include "stdafx.h"

int a=0;
int main(int argc, char* argv[])
{
    _asm
```

```
    {
        mov eax,fs:[0x1000];
        mov dword ptr ds:[a],eax;
    }
    return 0;
}
```

执行第一句汇编就发生内存访问冲突错误，就是因为 `0x1000` 超出了它的 `Limit` 的值 `0xFFF` ，如下图所示：

```
#include "stdafx.h"

int a=0;
int main(int argc, char* argv[])
{
    _asm
    {
        mov eax,fs:[0x1000];
        mov dword ptr ds:[a],eax;
    }
    return 0;
}
```

踩坑问题

里面有一些坑我还没让你踩，你踩一踩看看。请你思考出答案或者百思不得其解的时候再来看答案。

1. 在验证属性的时候，用下面的代码，结果运行 `mov dword ptr ds:[a],10` 正常通过，放开程序跑后内存访问冲突。

```
#include "stdafx.h"

int main(int argc, char* argv[])
{
    int a=0;
    _asm
    {
        mov ax,cs;
        mov ds,ax;
        mov dword ptr ds:[a],10;
    }
    return 0;
}
```

▼ 🎉 踩坑解决方案 🎉

为什么会出现这个问题呢？我明明代码很正常但就不行呢，难道只是因为局部变量的问题呢。但全局变量和局部变量在内存上根本没有区别。全局变量只是一个死地址，局部变量是该变量所在函数临时的地址，但访问上根本没有区别。我们看一看编译器到底把咱们的内联汇编到底翻译成了什么？为什么会出现这个问题呢？我明明代码很正常但就不行呢，难道只是因为局部变量的问题呢。但全局变量和局部变量在内存上根本没有区别。全局变量只是一个死地址，局部变量是该变量所在函数临时的地址，但访问上根本没有区别。我们看一看编译器到底把咱们的内联汇编到底翻译成了什么？

```
    mov ax,cs;
mov         ax,cs
    mov ds,ax;
mov         ds,ax
    mov dword ptr ds:[a],10;
mov         dword ptr [ebp-4],0Ah
}
return 0;
xor         eax,eax
```

前面的内敛汇编编译器给我一五一十的直接翻译了，但这个 `mov dword ptr ds:[a],10;` 内联汇编给我翻译成了啥，过分了！结果压根没有用 `ds` 的权限来访问，而是默认的 `ss` 来访问，这和预期结果一样才怪。

2.在探测Base属性的时候，使用gs作为试验寄存器，单步执行到 `mov eax,gs:[0]` ，出现内存访问冲突错误。

```
#include "stdafx.h"

int a=0;
int main(int argc, char* argv[])
{
    _asm
    {
```

```
        mov ax,fs;
        mov gs,ax;
        mov eax,gs:[0];
        mov dword ptr ds:[a],eax;
    }
    return 0;
}
```

▼ 🎉 踩坑解决方案 🎉

是因为每次单步调试，就会触发单步调试异常，进入内核，内核会把 `gs` 清零了，故导致实验无法成功。

下一篇

[保护模式篇——段描述符与段选择子]

分类: 羽夏看Win系统内核

标签: Win系统内核

posted @ 2021-09-19 18:20 寂静的羽夏 阅读(2987) 评论(0) 编辑 收藏 举报