南京航空航天大學
NANJING UNIVERSITY OF AERONAUTICS AND ASTRONAUTICS
计算机科学与技术学院
/人工智能学院

# 操作系统课程作业

162140222 黄钰轩

2024 年 4 月 11 日

**题目 1.** 一般操作系统中，进程的每个段内部地址均连续，但段与段的相对次序可能不同，请你用 C/CPP 语言写一个探测程序，探测 Windows、Linux 操作系统中进程的各段的相对位置 (输出次序即可).

**解答.** 具体实现思路请查看 README.md，程序代码如下：

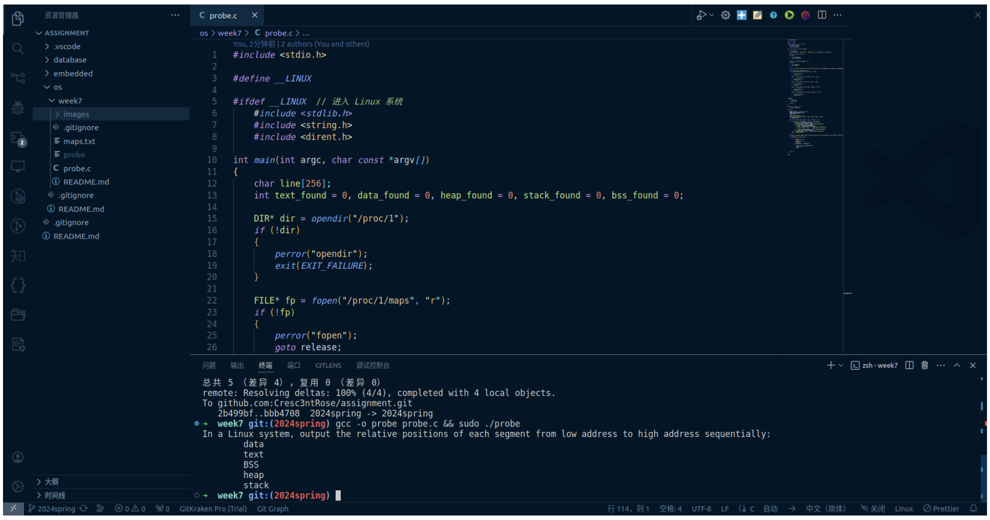
Listing 1: **probe.c**

```c
#include <stdio.h>

#ifndef  _WIN32 // 进入 Linux 系统
    #include <stdlib.h>
    #include <string.h>
    #include <dirent.h>

int main(int argc, char const *argv[])
{
    char line[256];
    int text_found = 0, data_found = 0, heap_found = 0, stack_found = 0, bss_found = 0;

    DIR* dir = opendir("/proc/1");
    if (!dir)
    {
        perror("opendir");
        exit(EXIT_FAILURE);
    }

    FILE* fp = fopen("/proc/1/maps", "r");
    if (!fp)
    {
        perror("fopen");
        goto release;
    }

    printf("In a Linux system, output the relative positions of each segment from low address to high address sequentially:\n");

    while (fgets(line, sizeof(line), fp))
        if (!text_found && strstr(line, "r-xp") != NULL)
        {
            printf("\ttext\n");
            text_found = 1;
        }
        else if (!data_found && strstr(line, "r--p") != NULL)
        {
            printf("\tdata\n");
            data_found = 1;
        }
        else if (!bss_found && strstr(line, "rw-p") != NULL)
        {
            printf("\tBSS\n");
            bss_found = 1;
        }
        else if (!heap_found && strstr(line, "[heap]") != NULL)
        {
            printf("\theap\n");
            heap_found = 1;
        }
        else if (!stack_found && strstr(line, "[stack]") != NULL)
        {
            printf("\tstack\n");
            stack_found = 1;
        }

release:
    if (fp)
        fclose(fp);
    closedir(dir);

    return 0;
}

#else // 进入 Windows 系统
```

```
    #include <windows.h>

enum SEG {
    TEXT,
    DATA,
    BSS,
    HEAP,
    STACK
};

typedef struct segment_info_t
{
    char* name;
    DWORD start_address;
    int SEG;
} segment_info_t;

int main() {
    segment_info_t segments[] = {
        {"text", 0, TEXT},
        {"data", 0, DATA},
        {"BSS", 0, BSS},
        {"heap", 0, HEAP},
        {"stack", 0, STACK}
    };

    int text_found = 0, data_found = 0, heap_found = 0, bss_found = 0;
    HANDLE hProcess = GetCurrentProcess();
    MEMORY_BASIC_INFORMATION mbi;
    DWORD_PTR addr = 0;

    SYSTEM_INFO sysinfo;
    GetSystemInfo(&sysinfo);
    segments[STACK].start_address = (DWORD_PTR)sysinfo.lpMaximumApplicationAddress;

    while (VirtualQuery((LPCVOID)addr, &mbi, sizeof(mbi)) != 0)
    {
        if (mbi.State == MEM_COMMIT && mbi.Type == MEM_PRIVATE)
            if (mbi.Protect & PAGE_EXECUTE_READ)
                for (int i = 0; i < 5; i ++ )
                    if (segments[i].SEG == TEXT && !text_found)
                    {
                        segments[i].start_address = (DWORD_PTR)mbi.BaseAddress;
                        text_found = 1;
                        break;
                    }
            else if (mbi.Protect & PAGE_READWRITE)
                if (mbi.AllocationBase == mbi.BaseAddress)
                    for (int i = 0; i < 5; i ++ )
114                         if (segments[i].SEG == BSS && !bss_found)
115                         {
116                             segments[i].start_address = (DWORD_PTR)mbi.BaseAddress;
117                             bss_found = 1;
118                             break;
119                         }
120                 else
121                     for (int i = 0; i < 5; i ++ )
122                         if (segments[i].SEG == HEAP && !heap_found)
123                         {
124                             segments[i].start_address = (DWORD_PTR)mbi.BaseAddress;
125                             heap_found = 1;
126                             break;
127                         }
128             else if (mbi.Protect & PAGE_READONLY)
129                 for (int i = 0; i < 5; i ++ )
130                     if (segments[i].SEG == DATA && !data_found)
131                     {
132                         segments[i].start_address = (DWORD_PTR)mbi.BaseAddress;
133                         data_found = 1;
134                         break;
135                     }
136         addr += mbi.RegionSize;
137     }
138 
139     printf("In a Windows system, output the relative positions of each segment from low address to
             high address sequentially:\n");
140     for (int i = 0; i < 5; i ++ )
141         for (int j = 0; j < 4 - i; j ++ )
142             if (segments[j].start_address > segments[j + 1].start_address)
143             {
144                 segment_info_t temp = segments[j];
145                 segments[j] = segments[j + 1];
146                 segments[j + 1] = temp;
147             }
148 
149     for (int i = 0; i < 5; i++)
150         printf("\t%s\n", segments[i].name);
151 
152     return 0;
153 }
154 
155 #endif
```

这里使用了条件编译，判断宏 _WIN32 是否有定义，即可分别在不同的操作系统中进行验证. 当进入 Linux 系统时，保持代码不变，而当进入 Windows 系统时，只需将第三行注释掉，即可正常编译.

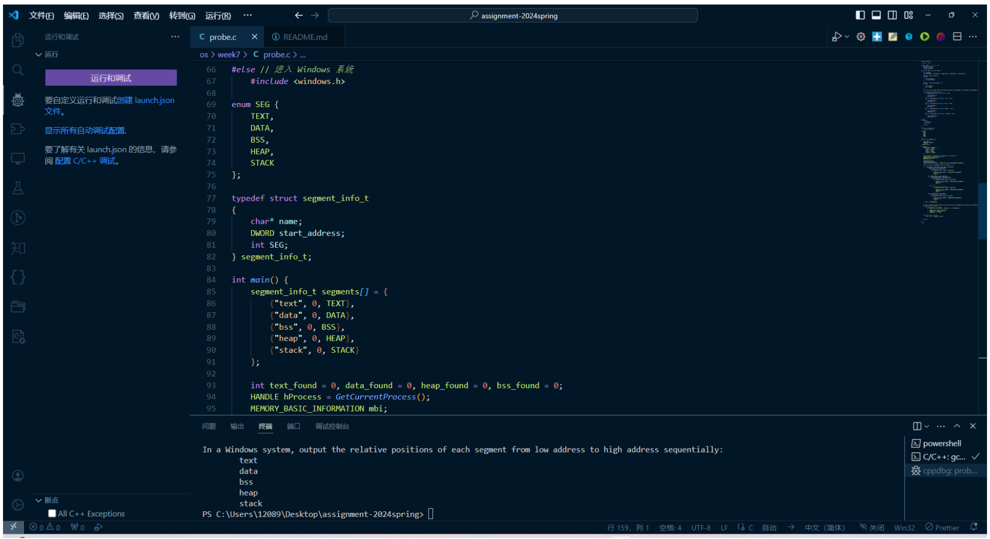
经测试，在两个系统中分别编译的结果如下：



```c
#include <stdio.h>

#define __LINUX

#ifdef __LINUX  // 进入 Linux 系统
    #include <stdlib.h>
    #include <string.h>
    #include <dirent.h>

int main(int argc, char const *argv[])
{
    char line[256];
    int text_found = 0, data_found = 0, heap_found = 0, stack_found = 0, bss_found = 0;

    DIR* dir = opendir("/proc/1");
    if (!dir)
    {
        perror("opendir");
        exit(EXIT_FAILURE);
    }

    FILE* fp = fopen("/proc/1/maps", "r");
    if (!fp)
    {
        perror("fopen");
        goto release;
```

```
总共 5 （差异 4），复用 0 （差异 0）
remote: Resolving deltas: 100% (4/4), completed with 4 local objects.
To github.com:Cresc3ntRose/assignment.git
   2b499bf..bbb4708  2024spring -> 2024spring
➜  week7 git:(2024spring) gcc -o probe probe.c && sudo ./probe
In a Linux system, output the relative positions of each segment from low address to high address sequentially:
        data
        text
        BSS
        heap
        stack
➜  week7 git:(2024spring) 
```

图 1: Linux 系统下编译代码



```c
#else // 进入 Windows 系统
    #include <windows.h>

enum SEG {
    TEXT,
    DATA,
    BSS,
    HEAP,
    STACK
};

typedef struct segment_info_t
{
    char* name;
    DWORD start_address;
    int SEG;
} segment_info_t;

int main() {
    segment_info_t segments[] = {
        {"text", 0, TEXT},
        {"data", 0, DATA},
        {"bss", 0, BSS},
        {"heap", 0, HEAP},
        {"stack", 0, STACK}
    };

    int text_found = 0, data_found = 0, heap_found = 0, bss_found = 0;
    HANDLE hProcess = GetCurrentProcess();
    MEMORY_BASIC_INFORMATION mbi;
```

```
In a Windows system, output the relative positions of each segment from low address to high address sequentially:
        text
        data
        bss
        heap
        stack
PS C:\Users\12089\Desktop\assignment-2024spring> 
```

图 2: Windows 系统下编译代码

**题目 1 的注记.** Linux 提供了 procfs，目录是 /proc，而 /proc/[pid]/maps 文件中就蕴含着完成这个作业所需的全部信息.

**题目 2.** 实现一程序，分别在 Windows、Linux 操作系统下验证：

(1) 栈、堆、数据区是否可读可写不可执行.

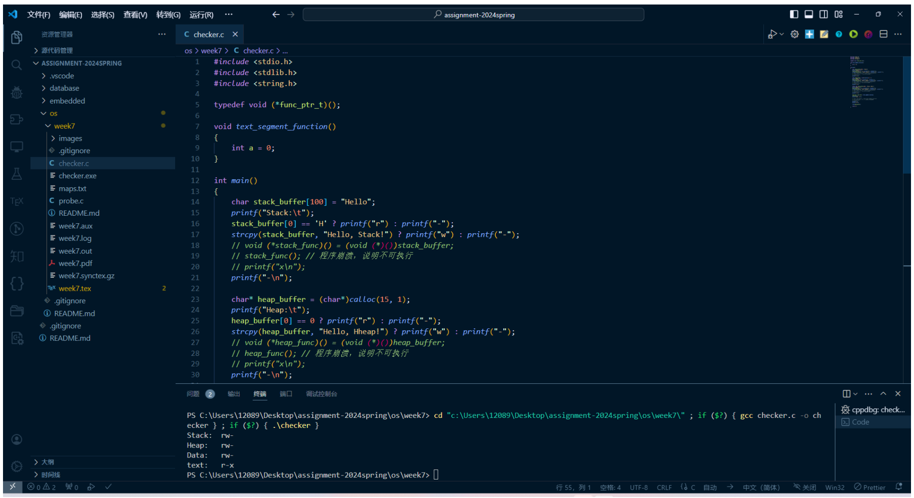
(2) 代码段是否可读不可写可执行.

**解答.** 具体实现思路请查看 README.md. 在 Windows 系统下，我选择使用 C 语言进行验证：

Listing 2: **checker.c**

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef void (*func_ptr_t)();

void text_segment_function()
{
    int a = 0;
}

int main()
{
    char stack_buffer[100] = "Hello";
    printf("Stack:\t");
    stack_buffer[0] == 'H' ? printf("r") : printf("-");
    strcpy(stack_buffer, "Hello, Stack!") ? printf("w") : printf("-");
    // void (*stack_func)() = (void (*)())stack_buffer;
    // stack_func(); // 程序崩溃，说明不可执行
    // printf("x\n");
    printf("-\n");

    char* heap_buffer = (char*)calloc(15, 1);
    printf("Heap:\t");
    heap_buffer[0] == 0 ? printf("r") : printf("-");
    strcpy(heap_buffer, "Hello, Hheap!") ? printf("w") : printf("-");
    // void (*heap_func)() = (void (*)())heap_buffer;
    // heap_func(); // 程序崩溃，说明不可执行
    // printf("x\n");
    printf("-\n");

    static char data_buffer[100] = "Hello, Data!";
    printf("Data:\t");
    data_buffer[0] == 'H' ? printf("r") : printf("-");
    strcpy(data_buffer, "Data changed!") ? printf("w") : printf("-");
    // void (*data_func)() = (void (*)())data_buffer;
    // data_func(); // 程序崩溃，说明不可执行
    // printf("x\n");
    printf("-\n");

    func_ptr_t text_func = text_segment_function;
    text_func();
    printf("text: \tr");
    // char* text_buffer = (char*)text_segment_function;
    // text_buffer[0] = 'X'; // 段错误,说明不可写
    // printf("w");
    printf("-");
    printf("x\n");

    free(heap_buffer);

    return 0;
}
```

在 Windows 系统中编译结果如下：

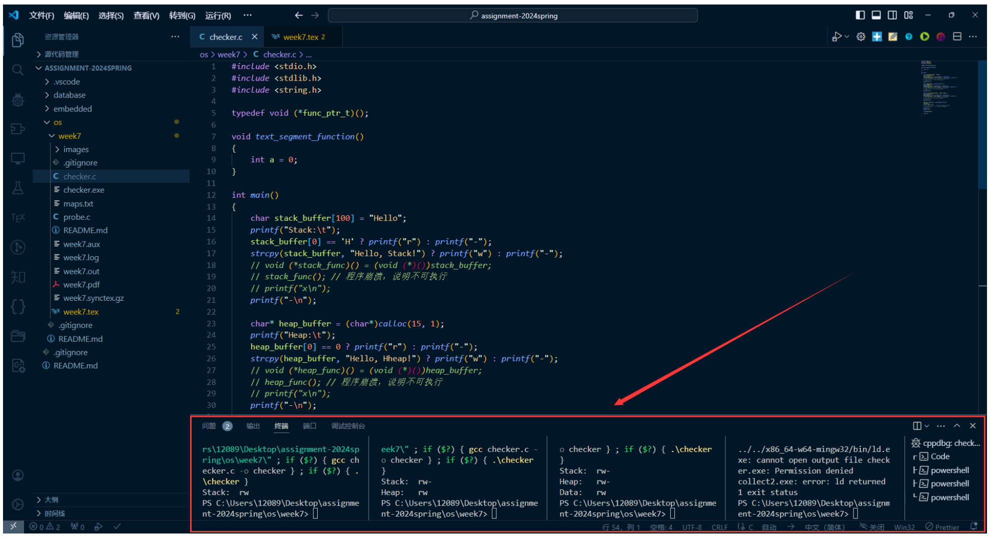


```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef void (*func_ptr_t)();

void text_segment_function()
{
    int a = 0;
}

int main()
{
    char stack_buffer[100] = "Hello";
    printf("Stack:\t");
    stack_buffer[0] == 'H' ? printf("r") : printf("-");
    strcpy(stack_buffer, "Hello, Stack!") ? printf("w") : printf("-");
    // void (*stack_func)() = (void (*)())stack_buffer;
    // stack_func(); // 程序崩溃，说明不可执行
    // printf("x\n");
    printf("-\n");

    char* heap_buffer = (char*)calloc(15, 1);
    printf("Heap:\t");
    heap_buffer[0] == 0 ? printf("r") : printf("-");
    strcpy(heap_buffer, "Hello, Hheap!") ? printf("w") : printf("-");
    // void (*heap_func)() = (void (*)())heap_buffer;
    // heap_func(); // 程序崩溃，说明不可执行
    // printf("x\n");
    printf("-\n");
```

```
PS C:\Users\12089\Desktop\assignment-2024spring\os\week7> cd "c:\Users\12089\Desktop\assignment-2024spring\os\week7\" ; if ($?) { gcc checker.c -o ch
ecker } ; if ($?) { .\checker }
Stack:  rw-
Heap:   rw-
Data:   rw-
text:   r-x
PS C:\Users\12089\Desktop\assignment-2024spring\os\week7> 
```

图 3: Windows 系统下编译代码

如果将代码中被注释的部分取消注释，那么就会引发程序崩溃.



```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef void (*func_ptr_t)();

void text_segment_function()
{
    int a = 0;
}

int main()
{
    char stack_buffer[100] = "Hello";
    printf("Stack:\t");
    stack_buffer[0] == 'H' ? printf("r") : printf("-");
    strcpy(stack_buffer, "Hello, Stack!") ? printf("w") : printf("-");
    // void (*stack_func)() = (void (*)())stack_buffer;
    // stack_func(); // 程序崩溃，说明不可执行
    // printf("x\n");
    printf("-\n");

    char* heap_buffer = (char*)calloc(15, 1);
    printf("Heap:\t");
    heap_buffer[0] == 0 ? printf("r") : printf("-");
    strcpy(heap_buffer, "Hello, Hheap!") ? printf("w") : printf("-");
    // void (*heap_func)() = (void (*)())heap_buffer;
    // heap_func(); // 程序崩溃，说明不可执行
    // printf("x\n");
    printf("-\n");
```

```
rs\12089\Desktop\assignment-2024sp
ring\os\week7\" ; if ($?) { gcc ch
ecker.c -o checker } ; if ($?) { .
\checker }
Stack:  rw
PS C:\Users\12089\Desktop\assignme
nt-2024spring\os\week7> 
```

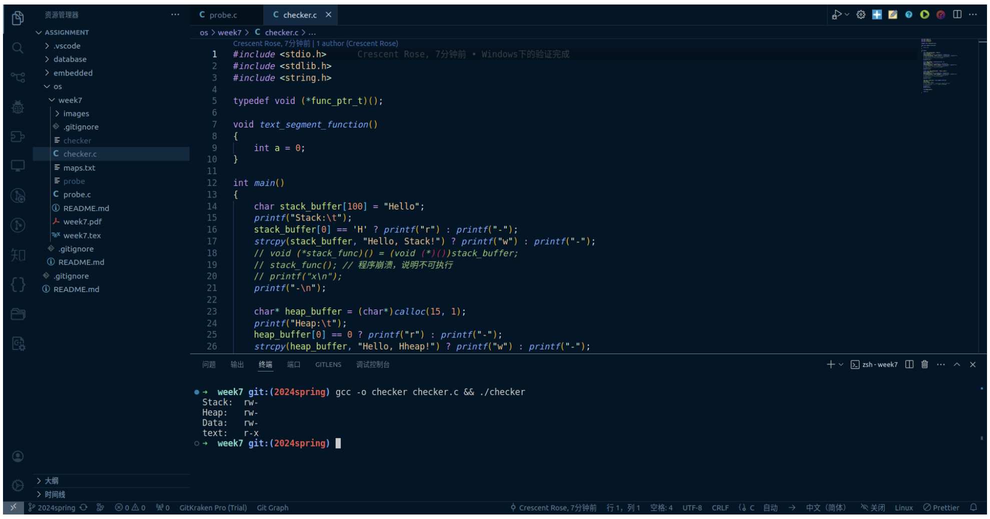
```
eek7\" ; if ($?) { gcc checker.c -
o checker } ; if ($?) { .\checker
}
Stack:  rw-
Heap:   rw
PS C:\Users\12089\Desktop\assignme
nt-2024spring\os\week7> 
```

```
o checker } ; if ($?) { .\checker
}
Stack:  rw-
Heap:   rw-
Data:   rw
PS C:\Users\12089\Desktop\assignme
nt-2024spring\os\week7> 
```

```
../../x86_64-w64-mingw32/bin/ld.e
xe: cannot open output file check
er.exe: Permission denied
collect2.exe: error: ld returned
1 exit status
PS C:\Users\12089\Desktop\assignm
ent-2024spring\os\week7> 
```

图 4: Windows 系统下编译代码

该程序在 Linux 系统下一样可以得到验证：

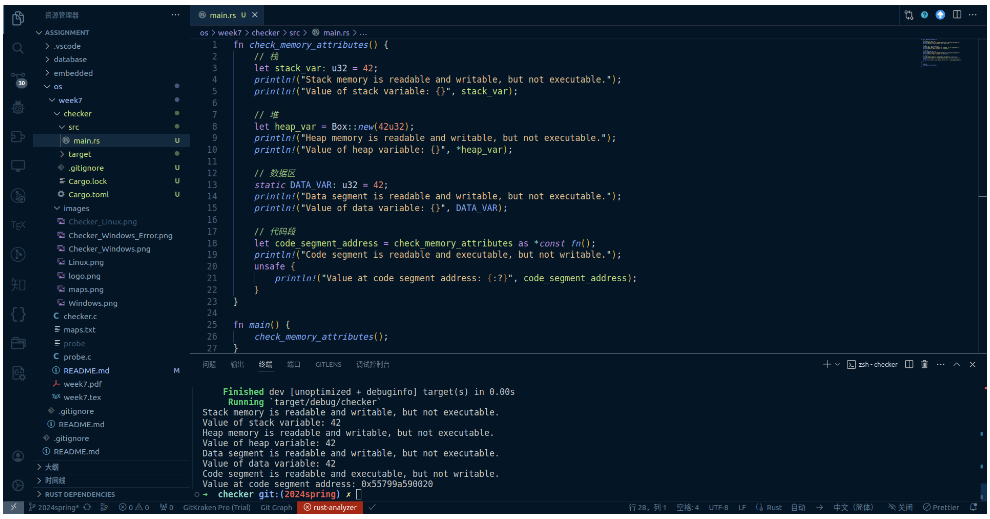


```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef void (*func_ptr_t)();

void text_segment_function()
{
    int a = 0;
}

int main()
{
    char stack_buffer[100] = "Hello";
    printf("Stack:\t");
    stack_buffer[0] == 'H' ? printf("r") : printf("-");
    strcpy(stack_buffer, "Hello, Stack!") ? printf("w") : printf("-");
    // void (*stack_func)() = (void (*)())stack_buffer;
    // stack_func(); // 程序崩溃，说明不可执行
    // printf("x\n");
    printf("-\n");

    char* heap_buffer = (char*)calloc(15, 1);
    printf("Heap:\t");
    heap_buffer[0] == 0 ? printf("r") : printf("-");
    strcpy(heap_buffer, "Hello, Hheap!") ? printf("w") : printf("-");
```

```
● → week7 git:(2024spring) gcc -o checker checker.c && ./checker
  Stack:  rw-
  Heap:   rw-
  Data:   rw-
  text:   r-x
○ → week7 git:(2024spring) █
```

图 5: Windows 系统下编译代码

同时，在 Linux 系统下，我尝试了使用 rust 语言构建程序进行验证：



```rust
fn check_memory_attributes() {
    // 栈
    let stack_var: u32 = 42;
    println!("Stack memory is readable and writable, but not executable.");
    println!("Value of stack variable: {}", stack_var);

    // 堆
    let heap_var = Box::new(42u32);
    println!("Heap memory is readable and writable, but not executable.");
    println!("Value of heap variable: {}", *heap_var);

    // 数据区
    static DATA_VAR: u32 = 42;
    println!("Data segment is readable and writable, but not executable.");
    println!("Value of data variable: {}", DATA_VAR);

    // 代码段
    let code_segment_address = check_memory_attributes as *const fn();
    println!("Code segment is readable and executable, but not writable.");
    unsafe {
        println!("Value at code segment address: {:?}", code_segment_address);
    }
}

fn main() {
    check_memory_attributes();
}
```

```
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
     Running `target/debug/checker`
Stack memory is readable and writable, but not executable.
Value of stack variable: 42
Heap memory is readable and writable, but not executable.
Value of heap variable: 42
Data segment is readable and writable, but not executable.
Value of data variable: 42
Code segment is readable and executable, but not writable.
Value at code segment address: 0x55799a590020
○ → checker git:(2024spring) ✗ 
```

图 6: Windows 系统下编译代码

最终同样完成了验证.