编译器对栈的使用分析

实验环境：Visual Studio 2022，MASM

问题 1：系统程序内存空间分布情况，如代码部分、全局变量部分、堆区、栈区等排列顺序；

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

int global = 10;

int uninitialized;// BSS

void print_address()
{
    // heap
    int* heap = (int*)malloc(sizeof(int));
    *heap = 30;
    // stack
    int local = 20;

    printf("code segment: %p\n", (void*)print_address);
    printf("gloabal var: %p\n", (void*)&global);
    printf("BSS segment: %p\n", (void*)&uninitialized);
    printf("stack segment: %p\n", (void*)&local);
    printf("heap segment: %p\n", (void*)heap);
    free(heap);
}
int main()
{
 print_address();
 return 0;
}
```

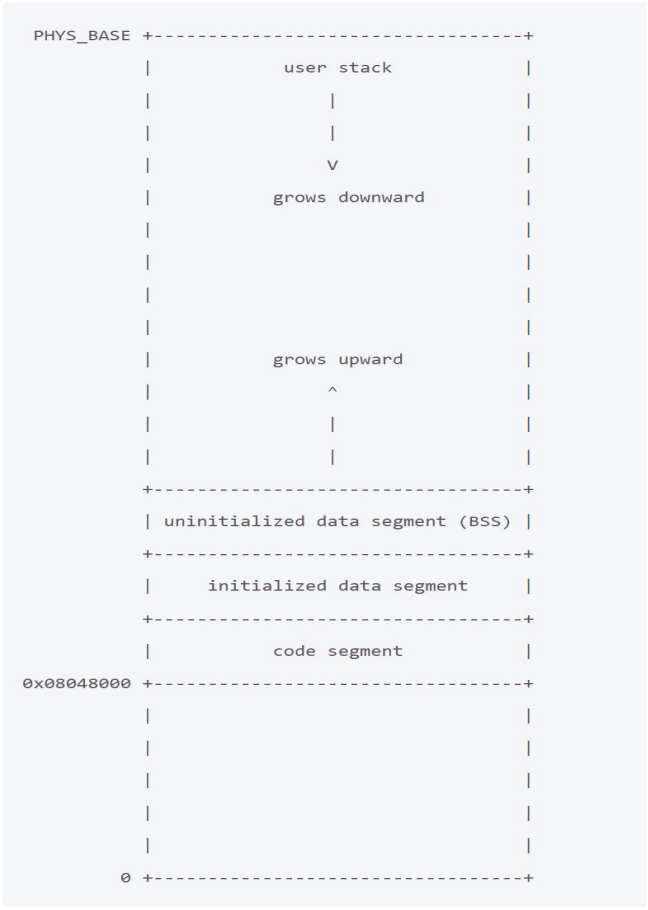
运行结果（除栈区地址，其他地址打印时默认省略了前面的 0）：

```
code segment: 0x401146
gloabal var: 0x404040
BSS segment: 0x404048
stack segment: 0x7ffddb4a92e4
heap segment: 0x20492a0
```

分析如下：

从操作系统课程中我们学习了 Virtual Memory Space 的相关知识，因此此处打印的不是真实的物理地址，而是虚拟内存地址，每一个 process 之间互不干扰。而在这段内存空间中，各部分的位置从低位开始是 process 的代码段空间，全局变量空间，未初始化变量空间（BSS），堆（向上增长，并留出增长空间），最后是栈（从 PHYS_BASE 向下增长）。PHYS_BASE 一般默认是 0xC0000000，其上方空间为 kernel memory。

这个是示意图：



而运行结果的显示也和我们的分析相吻合（将结果的打印句以从高位到低位的方式重新排列）：

栈区地址: 0x7ffddb4a92e4

堆区地址: 0x0000020492a0（补足前面 5 位的 0）

未初始化全局变量地址: 0x000000404048（补足前面 6 位的 0）

全局变量地址: 0x000000404040（补足前面 6 位的 0）

代码段地址: 0x000000401146（补足前面 6 位的 0）

问题 2：系统栈空间默认多大？如何修改？

```c
#include <stdio.h>
void recursion(int depth)
{
 int stack_var = depth;

 if (depth < 10 || depth % 1000 == 0)
 {
    printf("depth: %d, stack_address: %p\n", depth, (void
        *)&stack_var);
 }

 if (depth < 10000)
 {
     recursion(depth + 1);
 }
}
int main()
{
    recursion(1);
    return 0;
}
```

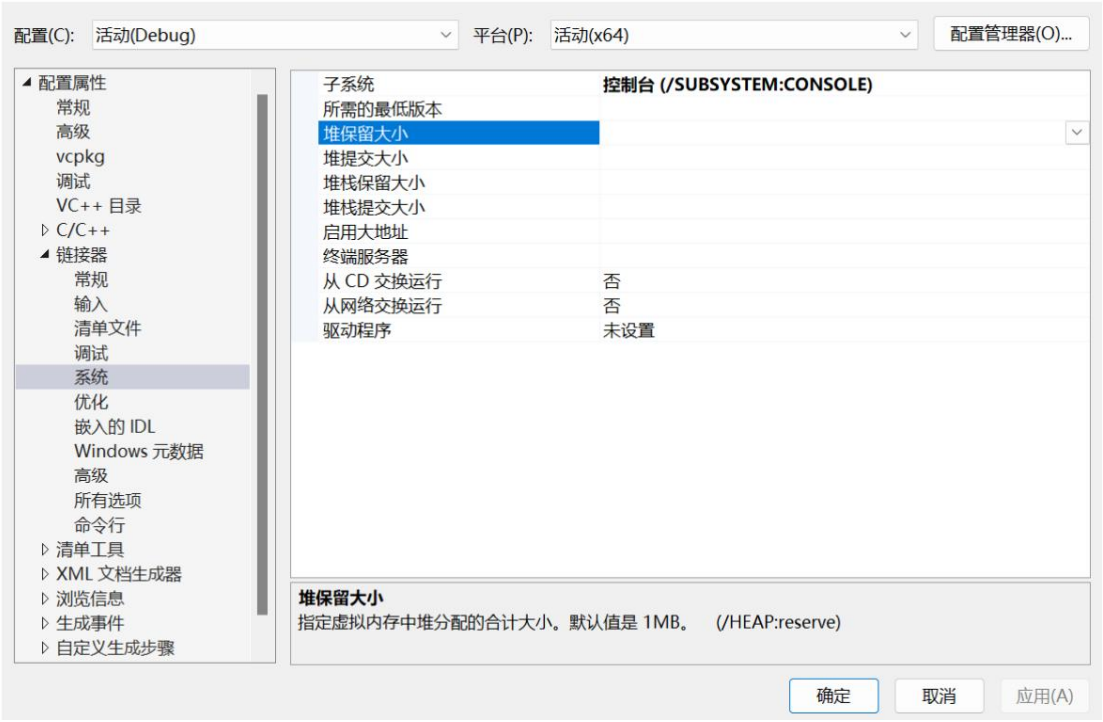
我通过递归函数不断压栈来探索系统给程序默认分配的栈空间大小，并设置了打印条件筛选，每 1000 深度进行打印，以及最大深度限制：

```
depth: 1, stack_address: 0x7ffcf662c65c
depth: 2, stack_address: 0x7ffcf662c62c
depth: 3, stack_address: 0x7ffcf662c5fc
depth: 4, stack_address: 0x7ffcf662c5cc
depth: 5, stack_address: 0x7ffcf662c59c
depth: 6, stack_address: 0x7ffcf662c56c
depth: 7, stack_address: 0x7ffcf662c53c
depth: 8, stack_address: 0x7ffcf662c50c
depth: 9, stack_address: 0x7ffcf662c4dc
depth: 1000, stack_address: 0x7ffcf6620b0c
depth: 2000, stack_address: 0x7ffcf6614f8c
depth: 3000, stack_address: 0x7ffcf660940c
depth: 4000, stack_address: 0x7ffcf65fd88c
depth: 5000, stack_address: 0x7ffcf65f1d0c
depth: 6000, stack_address: 0x7ffcf65e618c
depth: 7000, stack_address: 0x7ffcf65da60c
depth: 8000, stack_address: 0x7ffcf65cea8c
depth: 9000, stack_address: 0x7ffcf65c2f0c
depth: 10000, stack_address: 0x7ffcf65b738c
```

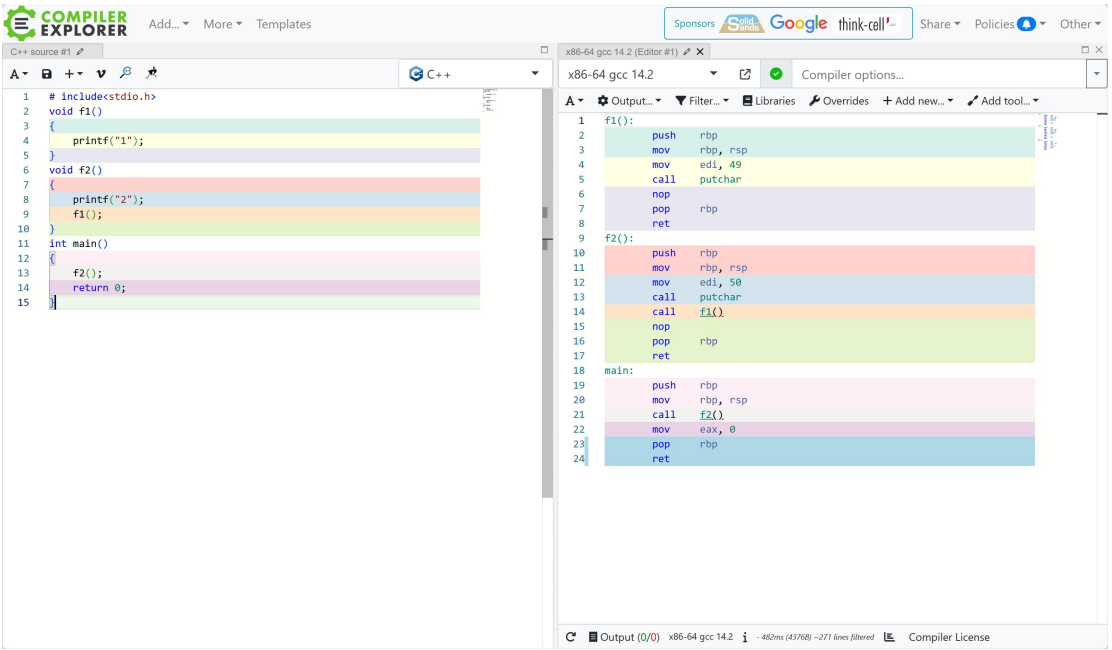
我发现可以达到最大深度而栈没有溢出，因此增加最大深度。最终在打印深度为 21000 后由于栈溢出报错。

分析：可以看出每次递归所占的栈空间大小为 0x65c-0x62c = 0x30 = 48 字节。而系统默认的程序栈空间，我们可以通过计算 21000 * 48 = 1008000 字节，22000*48 = 1056000 字节，而得到其约为 1MB（1024*1024 = 1048576 字节）

修改栈空间可以在 VS Studio 的项目属性中，选择配置属性，再选择链接器的系统，里面有对应的设置选项。



问题 3：课堂作业内容。



如图所示，左边为源程序代码，右边为源程序代码的汇编结果。

分析：

1. 函数调用关系：
   (1) Main 函数调用 f1
   (2) f1 调用 f2
2. 栈的使用规则

在汇编结果中，可以看到函数 main、f1 和 f2 都有进入函数和结束函数时的栈操作：

push rbp：将 rbp 推入栈中，保存调用者的基址寄存器（即其前者的栈帧）。

Mov rbp, rsp: 将 rsp（栈指针）赋值给 rbp，设置新的栈帧（即自己的栈帧）

pop rbp: 将栈上的第一个元素取出放入 rbp，恢复调用者的基址寄存器

ret：返回调用点，退出函数。

实际过程中，还有更详细的栈操作过程（举例假设）：

sub rsp, 16：为栈帧分配空间

lea rsp, ... : 释放栈帧空间

3. 调用过程的栈帧变化：每一个函数都对应一个 frame，里面含有该函数的相关信息，如参数，局部变量和返回地址等。而 frame 会在调用其他函数时保存，返回到原函数后恢复原来的 frame。当一个函数返回后（ret），其 frame 会销毁。
4. 栈的增长方向： 与第一问相同，栈从高位向低位增长。