C语言进程的内存布局：

程序：就是我们写好的代码并编译完成的那个二进制文件，它被存放与磁盘中，它是死的。

进程：把磁盘中的二进制文件"拷贝"到内存中取执行它，让运行起来，它是活的。

所有的程序被执行起来之后，系统会为他分配各种资源内存，用来存放该进程中用到的各种变量、常量、代码等等。这些不容的内容将会被存放到内存中不同的位置 （区域），不同的内存区域他的特性是右差别。

每一个进程所拥有的内存都是一个虚拟的内存，所谓的虚拟内存是用物理内存中映射（投影）而来的，对于每一个进程而言所有的虚拟内存布局都是一样的。让每个进程都以为自己独自拥有了完整的内存空间。

物理内存（Physical Memory）

虚拟内存（Virtual Memory）

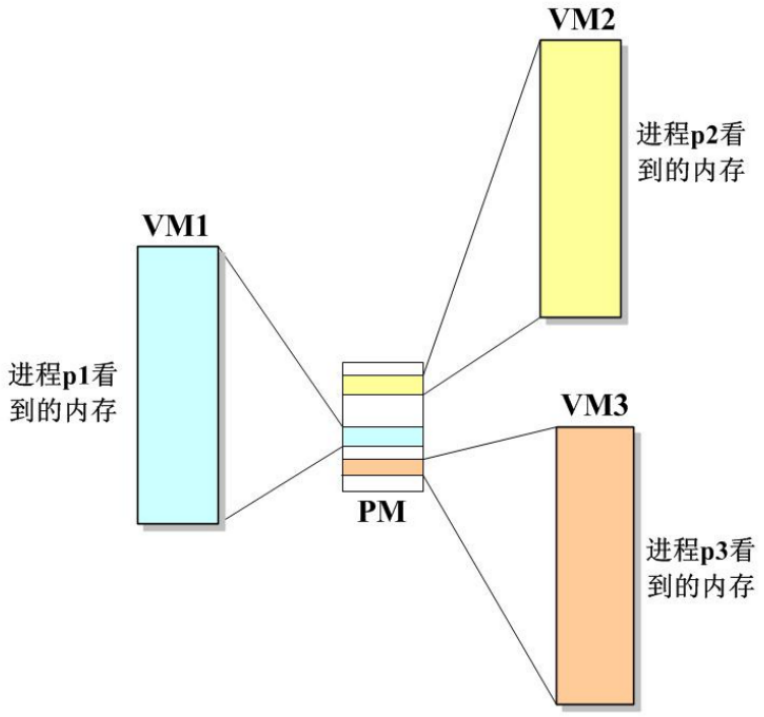


图 2-58 从物理空间映射到虚拟空间

虚拟内存的布局（区域）：

栈 （stack）

堆（heap）

数据段

代码段

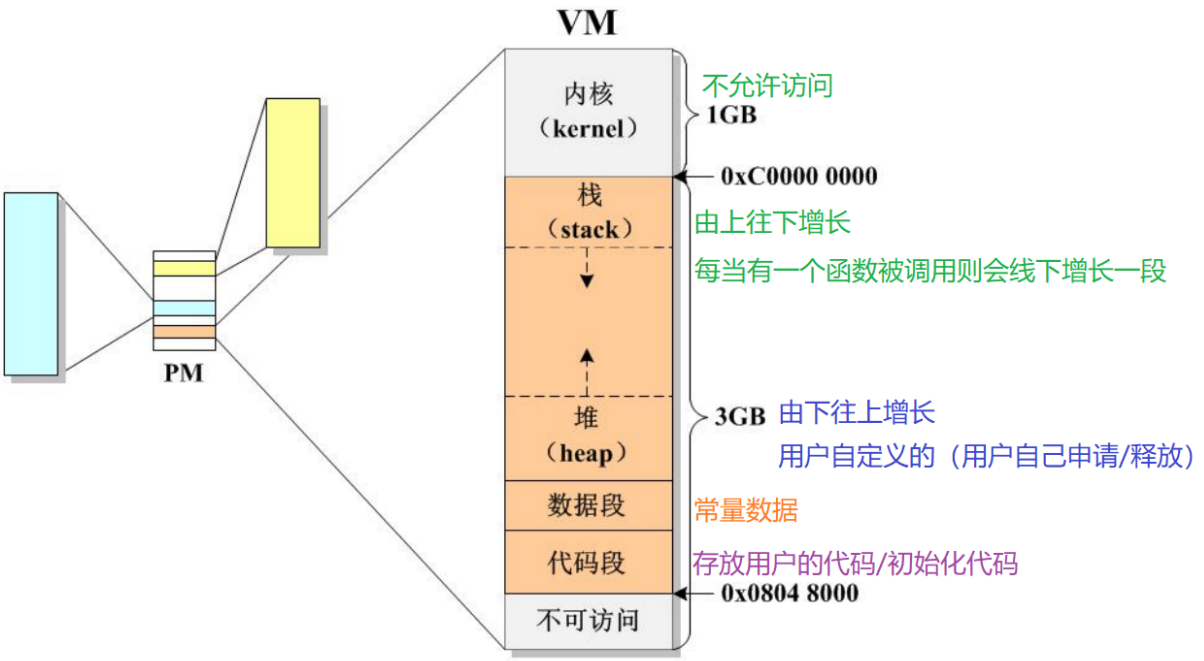


图 2-59 进程的虚拟空间

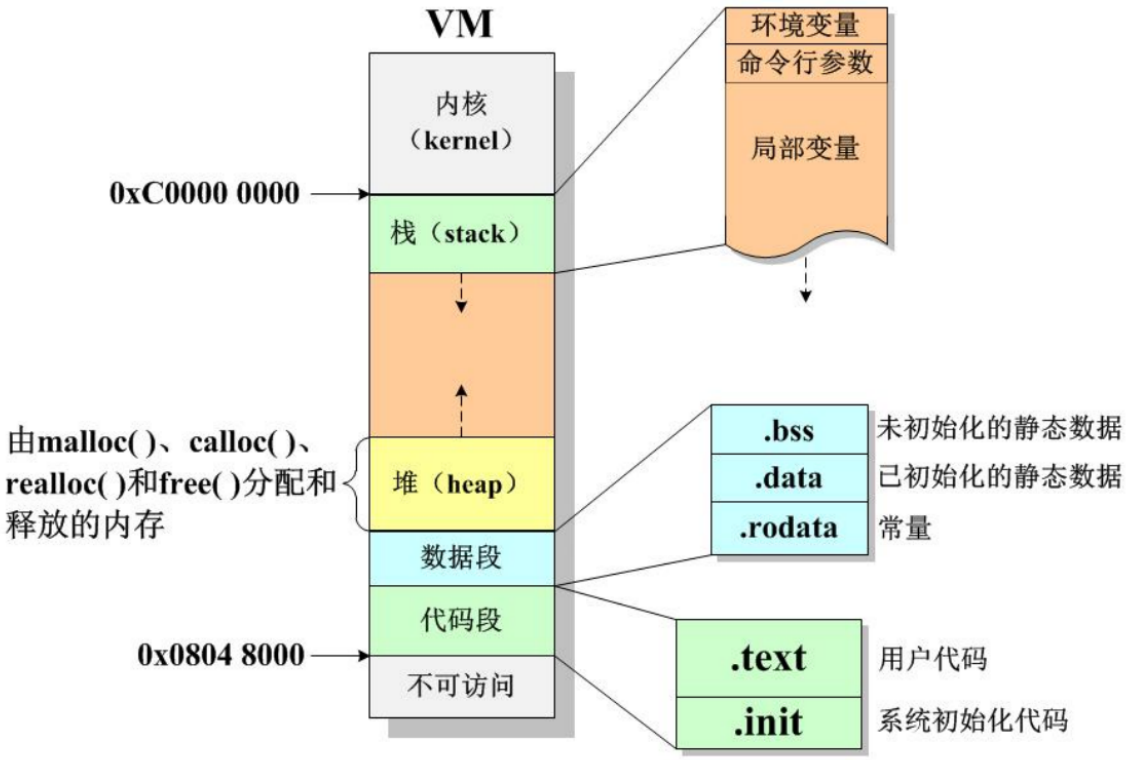
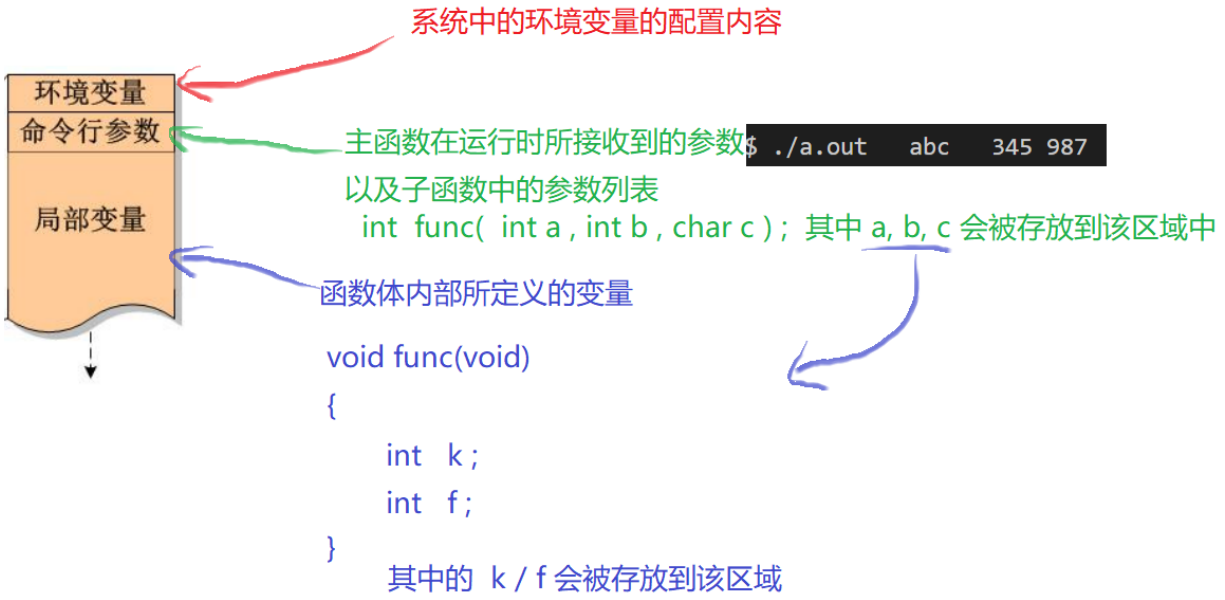


图 2-60 虚拟空间的各个部分

栈空间：



栈空间的特点：

- 空间非常有限，尤其在嵌入式的环境下，因此我们应该尽可能少去使用栈空间内存，特别是要存放比较大的数据。

```
ulimit -a
stack size (kbytes, -s) 8192 当前64位系统 默认只有8M栈内存
$ ulimit -s 10240 // 临时修改为 10M 重启后会回到默认值

```

- 每当一个函数被调用的时候， 栈空间会向下增长一段，用来存放该函数的局部变量
- 当一个函数退出的时候， 栈空间会向上回缩一段，该空间的所有权将归还系统
- 栈空间的分配与释放，用户是无法干预的， 全部由系统来完成。

静态变量：

在C语言中有两种静态变量

全局变量： 定义在函数体之外的变量

静态的局部变量： 定义在函数体内部而且被 static 修饰的变量

```
int c ; // 在函数体之外，属于全局变量 --> 静态变量

```

```c
int func(int argc , char * argv[] ) // argc argv 属于main函数的局部变量
{
 int a ; // 局部变量
 static int b ; // 静态局部变量
 // 静态的局部变量 初始化语句只会被被执行一次

}
```

```c

int k = 1000 ;

void func(void)
{

    int a  = 250  ;
    static int b = 100 ;

     printf("a:%d , b:%d \n " , ++a , ++b );

}


int main(void)
{

    func();
    func();
    func();

    return 0;
}
```

为什么会有静态变量?

- 当我们需要把一个变量引用到不同的函数内部甚至不在同一个.c 文件中，可以全局变量来实现。
- 当我们需要一个局部变量用来记录某个值， 并希望这个值不会被重新初始化的情况下可以使用静态的局部变量。

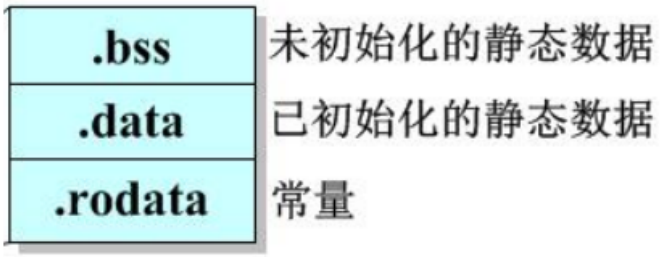
数据段与代码段:

数据段有哪些内容:

- .bss 未初始化的静态数据， 会被自动初始化为0
- .data 已初始化的静态数据
- .rodata 存放常量数据 "Hello Even" ，不允许修改的（只读）

代码段中有那些内容：

- 用户的代码 （比如我们自己写函数func.....main）
- 系统初始化代码，由编译器为我们添加的



数据段的特点：

- 没有初始化则自动初始化为0
- 初始化语句只会被执行一次（在程序被加载的过程中已经初始化结束）
- 静态数据的内存从程序运行之初就存在，直到程序退出才会被释放（与进程共生死）

堆内存：

堆内存，又称为动态内存、自由内存、简称堆。唯一一个由开发者随意分配与释放的内存空间。具体的申请大小，使用的时常都是由我们自己来决定。

堆内存空间的基本特性：

- 相对与栈空间来说， 堆空间大很多（堆的大小受限于物理内存），系统不会对对空间进行限制。
- 相对与栈空间来说， 堆内存是从下往上增长的。
- 堆空间的内存称为匿名内存， 不像栈空间那样有个名字，只能通过指针来访问
- 堆空间内存的申请与释放都是由用户自己完成，用户申请之后需要手动去释放，直到程序退出。

如何申请堆空间内存：

malloc （只是申请内存而已，并不会清空）

```
malloc (向系统申请内存)
 头文件：
 #include <stdlib.h>
 函数原型：
 void *malloc(size_t size);
 参数分析：
 size --> 需要申请的内存 （字节为单位）
 返回值：
 成功 返回一个指向成功申请到内存的指针（入口地址）
  失败 返回 NULL

```

calloc （会把内存进行清空）

```
calloc (向系统申请内存)
 头文件：
 #include <stdlib.h>
 函数原型：
 void *calloc(size_t nmemb, size_t size);
 参数分析：
 nmemb -- > N 块内存（连续的）
 size -- > 每一块内存的大小尺寸
 返回值：
  成功 返回一个指向成功申请到内存的指针（入口地址）
  失败 返回 NULL

```

realloc

```
realloc (重新申请空间)
 头文件：
 #include <stdlib.h>
 函数原型：
 void *realloc(void *ptr, size_t size);
 参数分析：
 ptr --> 需要 扩容/缩小 的内存的入口地址
```

```
size --> 目前需要的大小
返回值：
 成功 返回修改后的地址
 失败 NULL
```

清空：

bzero

memset

如何释放：

free

```
free(释放堆内存)
 头文件：
 #include <stdlib.h>
 函数原型：
 void free(void *ptr);
 参数分析：
 ptr --> 需要释放的内存的入口地址
 返回值；
 无
```

示例1 malloc：

```

    int * p = (int *)malloc( 10 * sizeof(int) );  // 申请一个可以存放10个整型的 堆空间

    for (int i = 0; i < 10 ; i++)
    {
        *(p+i) = i;
    }

    for (int i = 0; i < 10 ; i++)
     {
         printf("*(p+%d):%d\n" ,i,  *(p+i));
     }

     //   &p 打印的是栈空间地址      p中所存放的地址为堆空间地址
     printf("&p:%p--->%p\n" , &p , p );

```

```
17
18      //不再使用该堆空间时需要释放
19      free(p);
20
```

示例2 calloc:

```
1       //            申请10块连续的内存  ，每一块为 整型大小
2       int * p = calloc( 10 , sizeof(int));
3
4       for (size_t i = 0; i < 10; i++)
5       {
6           *(p+i) = i+ 998 ;
7       }
8
9       for (int i = 0; i < 10; i++)
10       {
11          printf("*(p+%d):%d\n" , i , *(p+i));
12       }
13
14      //不再使用该堆空间时需要释放
15      free(p);
16
```

示例 3 realloc

```
1       //            申请10块连续的内存  ，每一块为 整型大小
2       int * p = calloc( 10 , sizeof(int));
3
4       for (size_t i = 0; i < 10; i++)
5       {
6           *(p+i) = i+ 998 ;
7       }
8
9       for (int i = 0; i < 10; i++)
10       {
11          printf("*(p+%d):%d\n" , i , *(p+i));
12       }
13
14
15      printf("重新分配前p:%p\n" , p );
16
```

```c
    // 如果重新申请内存需要另找宝地 ，  那么 原本的地址会被释放掉
    int * p1 = realloc( p , 128);
    p = NULL ;// 让p不要称为野指针，  需要让它指向NULL

    printf("重新分配后p1:%p\n" , p1 );

    //realloc 会帮我们把原有的数进行搬运
    for (int i = 0; i < 10; i++)
    {
       printf("*(p1+%d):%d\n" , i , *(p1+i));
    }


    //不再使用该堆空间时需要释放
    free(p1);

```

总结：

- 使用malloc 申请内存时， 内存中的值时随机值， 可以使用bzero清空
- calloc 申请内存时， 内存中的值会被初始化为 0
- free 只能释放堆空间的内存，不能释放其它区域的内存

释放内存的含义：

- 释放内存仅仅意为着，当前内存的所有权交回给系统
- 释放内存并不会清空内存的内容
- 也不会改变指针的指向，需要手动把指针指向NULL， 不然就成野指针了

练习：

尝试申请一个堆空间。用来存放字符串

研究该字符串是否能被修改?

完成get_memory(1).c

拓展预习：

static 修饰的作用：

修饰局部变量： 使得局部变量的存放位置从栈空间改为数据段

修饰全局变量： 可以缩小全局变量的可见范围，由原本的拷文件可见变为本文可见，可以使得重名概率降低

修饰函数： 使得函数为本文将可见....

有没有其它的？？？

calloc如果把内存变小，小到存不下原有的数据会怎样么样？？

内存拷贝函数：

memcpy

strcpy

strncpy

sprontf

。。。。。。。