Middleware Development technology

中间件开发技术

Chapter 2 Fundament of Middleware technology

- interoperation in procedure-oriented

-CSEE Hunan university -

Jin-Min Yang
(杨金民)

2016.02

Contents

- 互操作的基本含义；
- 互操作中的问题；
- 问题的解决思路；
  - 虚拟内存；
  - 相对寻址，间接寻址；
  - Compiling， linking，loading；
  - Stack， thread；
- .函数调用规范；
- 同一机器内进程之间的互操作；
- 不同机器之间的互操作；

Software history

A Computer run a application:

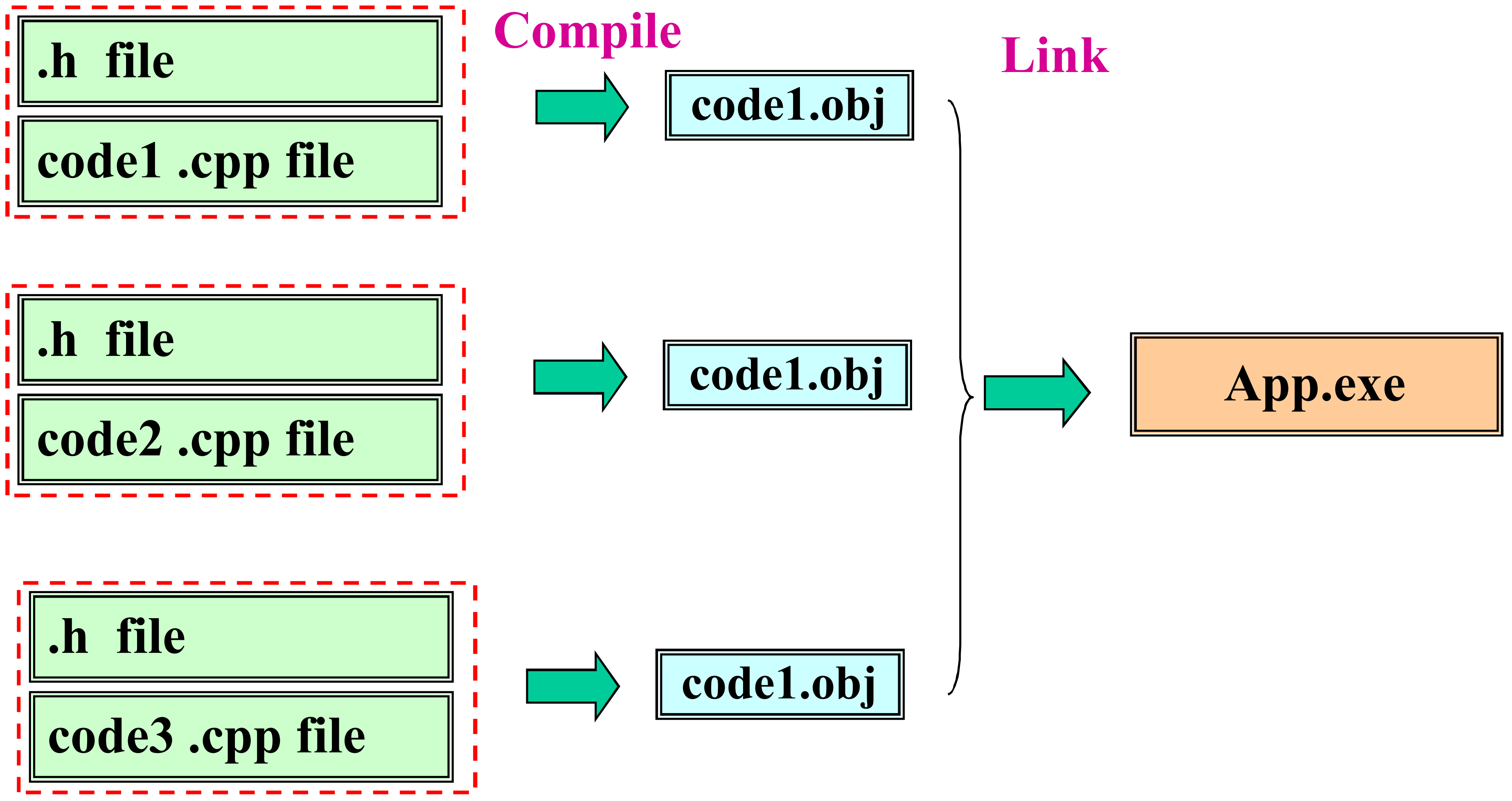
- A group developed an application;
- Integrate multiple modules to form a applications;

A computer run concurrently multiple applications;

二进制级互操作实现技术

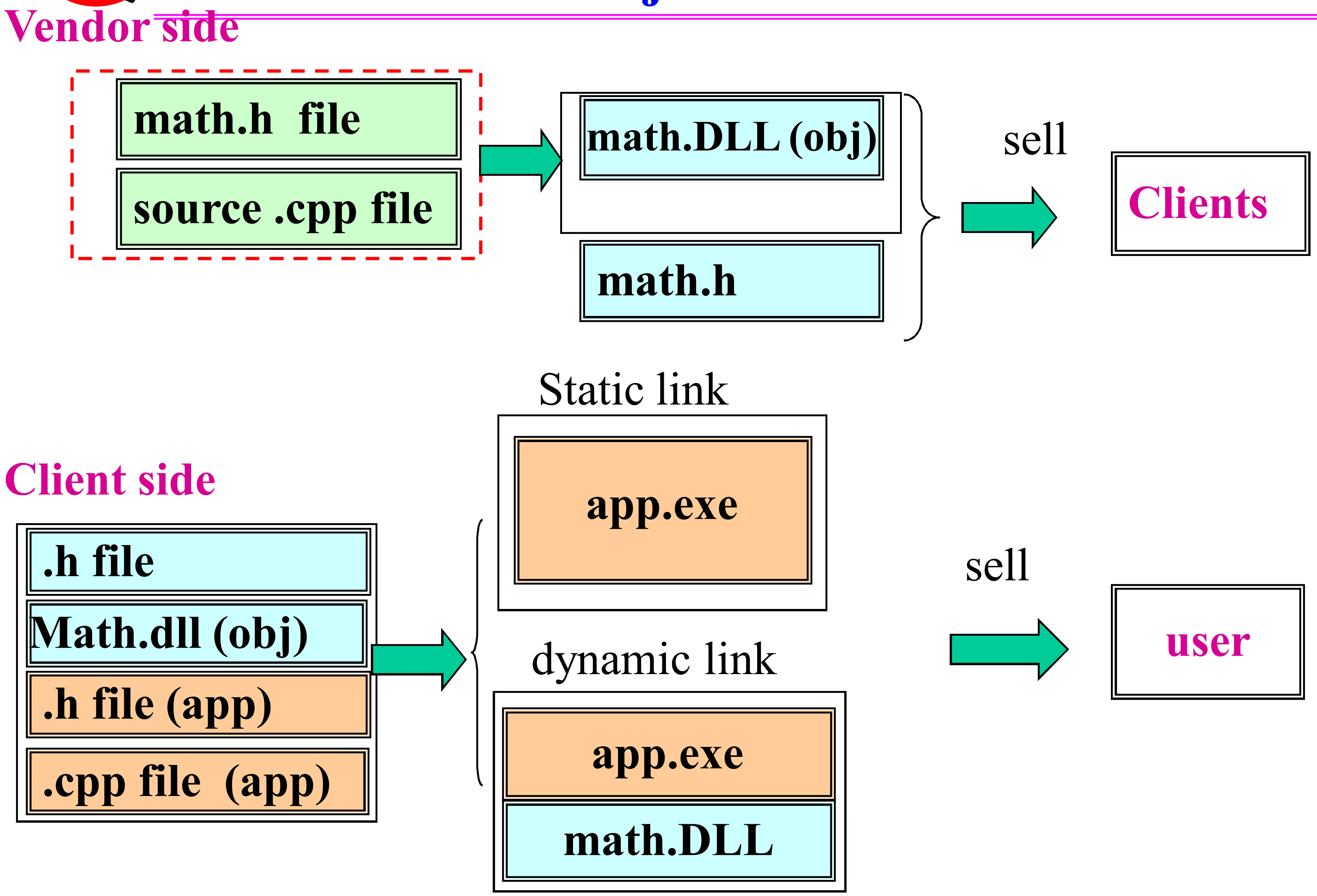
- binary interoperation:
  - process；
  - Compiling, compiler;
  - Linking, linker;
  - Loading, loader；
  - code re-entry;
  - function recursive calling;
  - Stack;
  - Thread;
  - Positioning of function and data；
  - Dynamic Linking,and static linking；
  - Local， variable， static， new, variable;

Development aspect figure in interoperation in object-oriented



Problem: any modification to source files must re-compile and re-link;

Development aspect figure in interoperation in object-oriented



服务方程序：一个函数的实现

printf.h 文件:

```
bool printf (char * p， int i, double j, double k);
```

服务方程序：一个函数的实现

真实的实现函数：
printf.c:

```
bool printf (char * p，  int i, double j, double k)
{
   int factor = atoi(p);
   if (factor *j > I*k)
       return true;
  else
       return false;
}
```

编译链接生成：./original/printf.dll

代理printf_proxy.c

```
include <MiddlewareEnvironment.h>
bool printf(char * p, int length, double width, double area)   {
    Pipeline = CreatePipeline("to server");
    Pipeline.AddDllName("printf.dll");
    Pipeline.AddFunctionName("printf_stub");
    Pipeline.AddString(p );
    Pipeline.AddIntParameter(length );
    Pipeline.AddDoubleParameter(width);
    Pipeline.AddDoubleParameter(area );
    Pipeline.Call( );
   Pipeline.WaitForResult( );
   bool ret = Pipeline.FetchBoolValue( );
   Pipeline.Close( );
   Return ret;
}
```

编译链接生成./proxy/prinf.dll

存根printf_stub.c

```
include <stdio.h>
bool printf_stub(char * parameterpackage)   {
    char *p;
    int length;
   double width;
   double area;
    p = AbstractString[parameterpackage);
    length = AbstractInt(parameterpackage) ;
    width = AbstractDouble(parameterpackage) ;
    area =   AbstractDouble(parameterpackage) ;
   bool ret = printf(p, length,width,area);
   return ret;
}
```

编译链接生成./stub/prinf.dll

真实的函数：

```
bool printf (char * p，  int i, double j, double k)   { ……}
```

服务方守护进程(Damon)的实现

```
void _ServiceThread(    )   {
   unsigned char szDllName[128], szFunName[128], pBuffer[4096]
   Bool   (*pfn)( unsigned char *) = 0;
   Pipeline = CreatePipeline("port");
   while (1)   {
          Pipeline.WaitForPortRequest( );
          Pipeline.FetchDllName(szDllName);
          Pipeline. FetchFunctionName(szFunName);
          Pipeline. FetchParameters(pBuffer);
           HINSTANCE h = LoadLibrary(szDllName);
           if (h)  *(FARPROC* )&pfn = GetProcAddress(h, szFunName);
           If (pfn)  int result = pfn(pBuffer);
           Pipeline.AddIntValue(result );
           TriggerEvent(hReturnEvent);
   }   //while
}
```

编译链接生成./server.exe

客户程序

client.c

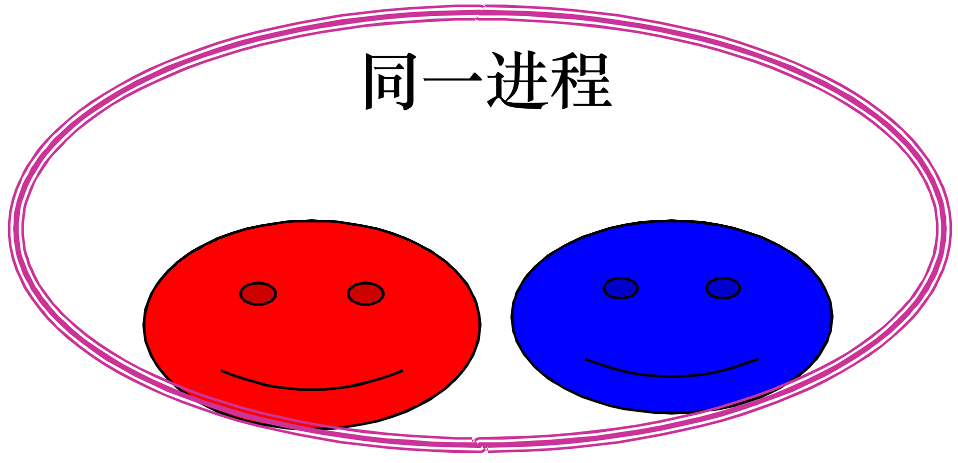
```
include <math.h>
include <stdio.h>
include  “printf.h”

main( )   {
  int i=12;
  double j =3.6;
  printf(“Area(%d,%f3.2)=%f3.2”,  i,  j, i*j);
  return;
}
```

编译、和printf.dll 链接生成client.exe

部署－同一进程中的互操作

把client.exe
把./original/printf.dll

拷贝到用户的运行目录:
C:\app\



client.exe　　　　printf.dll

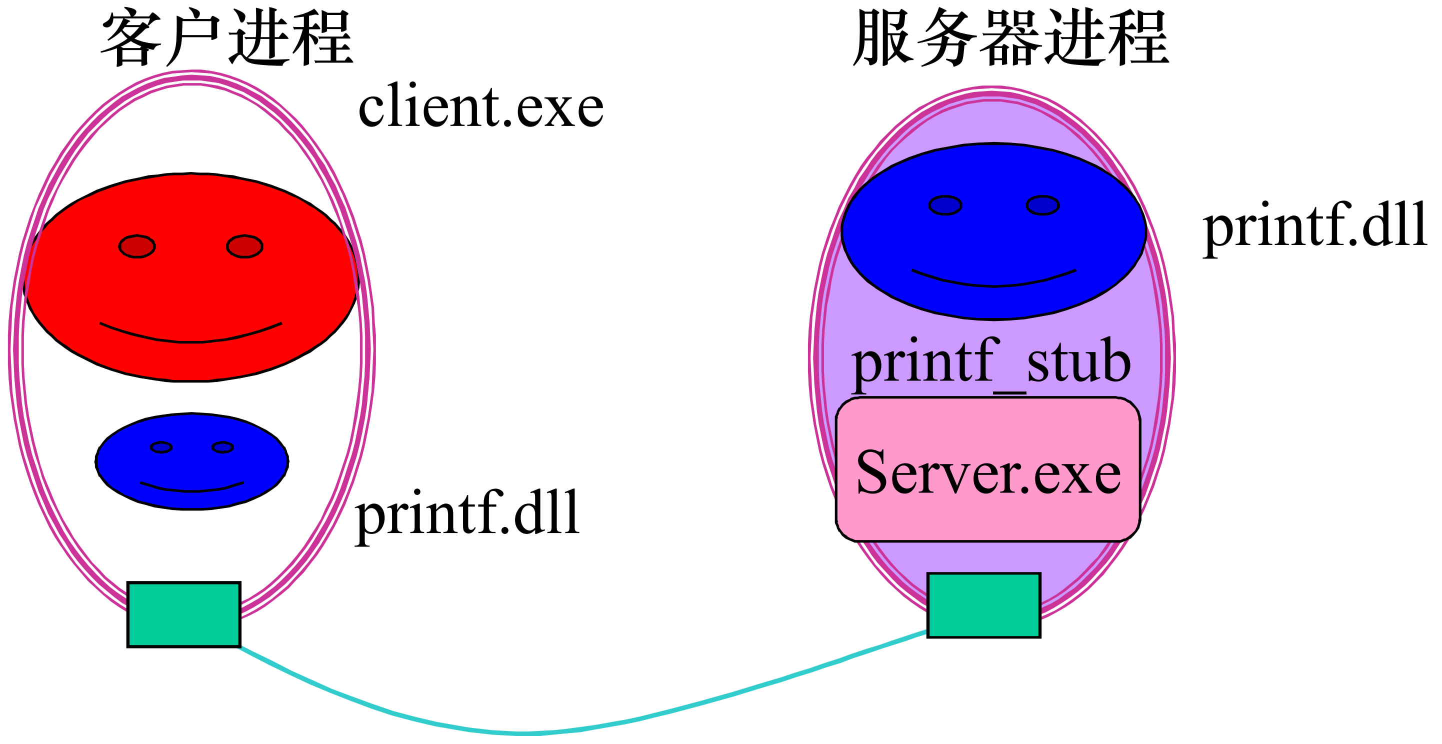
部署—不同进程中的互操作

把client.exe
把./proxy/printf.dll

拷贝到用户机器的运行目录：
C:\app\

把server.exe
把./stub/printf.dll

拷贝到服务器的运行目录：
C:\app\



计算模型

- 运算的两个基本要素：**代码**和**数据**:

什么运算?

该运算涉及什么数据?

运算指令（代码）放在哪里?

要运算的数据放在哪里?

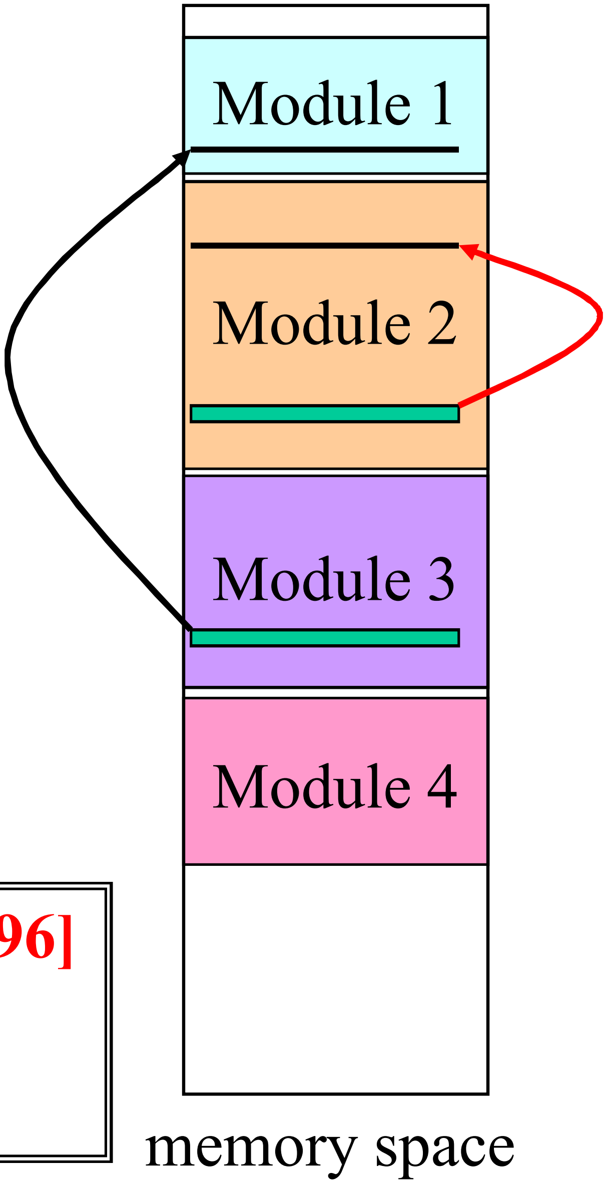
运算的结果放在哪里?

谁负责来执行这个运算?

```
int k = mv1+ 66;
void mf2( ) ;
```

IP

```
MOV   EAX,  ptr [4096]
ADD   EAX, 66;
CALL  8192;
```



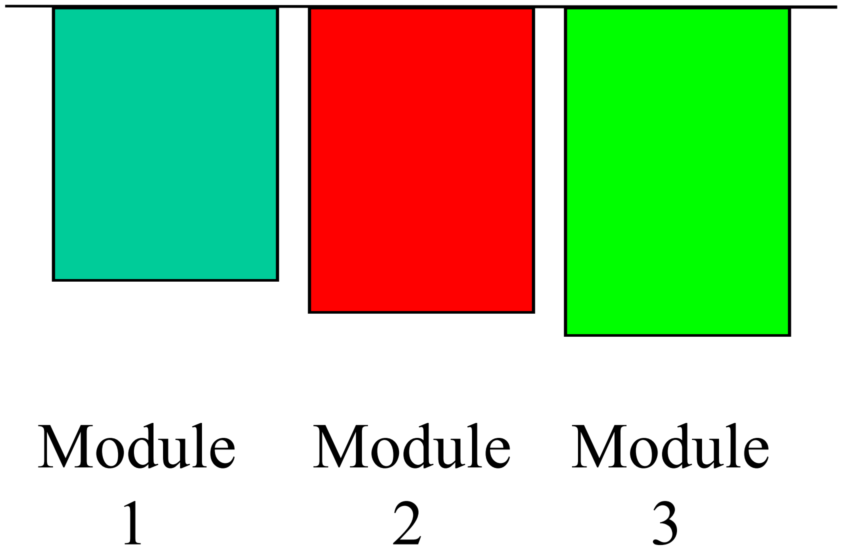
计算模型(cont.)

- 计算机: **CPU** plus **内存**;
- **程序: Code** plus **data**;
- **线性执行模型:**
  - 指令流水线：指令一个一个地依次排列，构成队列;
- **数据和代码都存存储在内存中，它们的定位问题:**
  - 内存寻址;
  - 最小存储单元为字节;
  - 就寻址而言，所有的单元具有对等性;
  - 所有存储单元的地址在长度上是一样的，而且是事先定下不变的;
- **寄存器(Register)**是一种与内存相比，具有更高访问速度的存储器，只存储一个数据，用名字来标识寄存器。.

编译器: 把源代码翻译成可执行代码

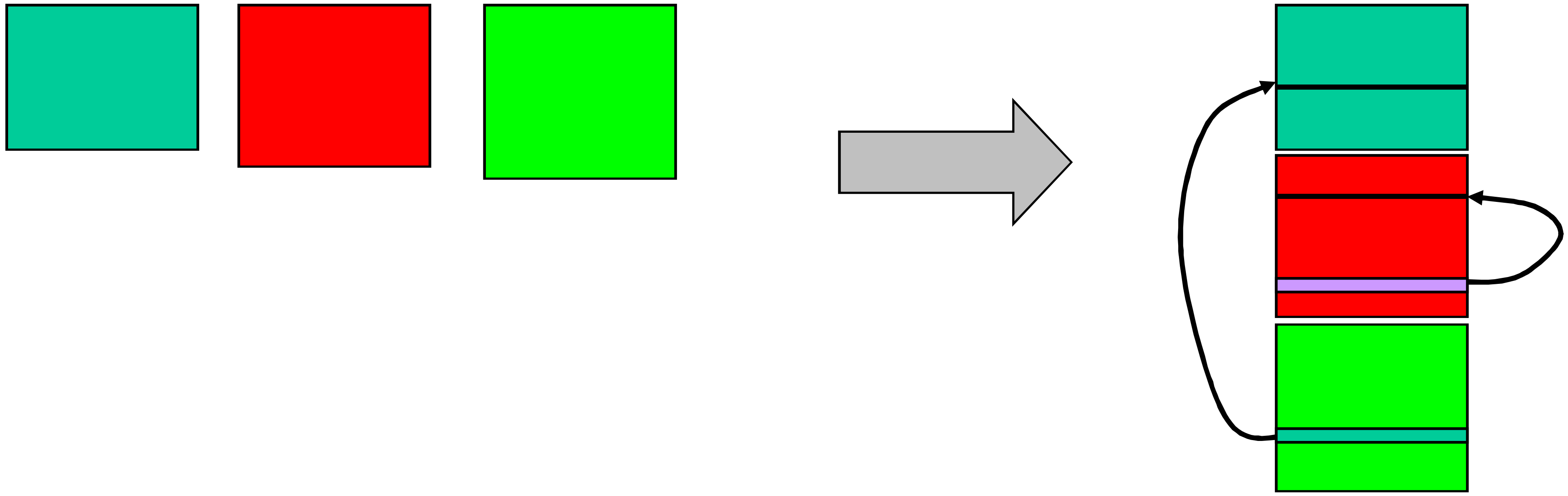
- 翻译时，假定所生成的模块，在运行时被加载到内存的0地址执行；

```
Void mf2( int k);
Global variable  mv1;
--------------------------
int k = mv1+ 66;
mf2(k ) ;
```



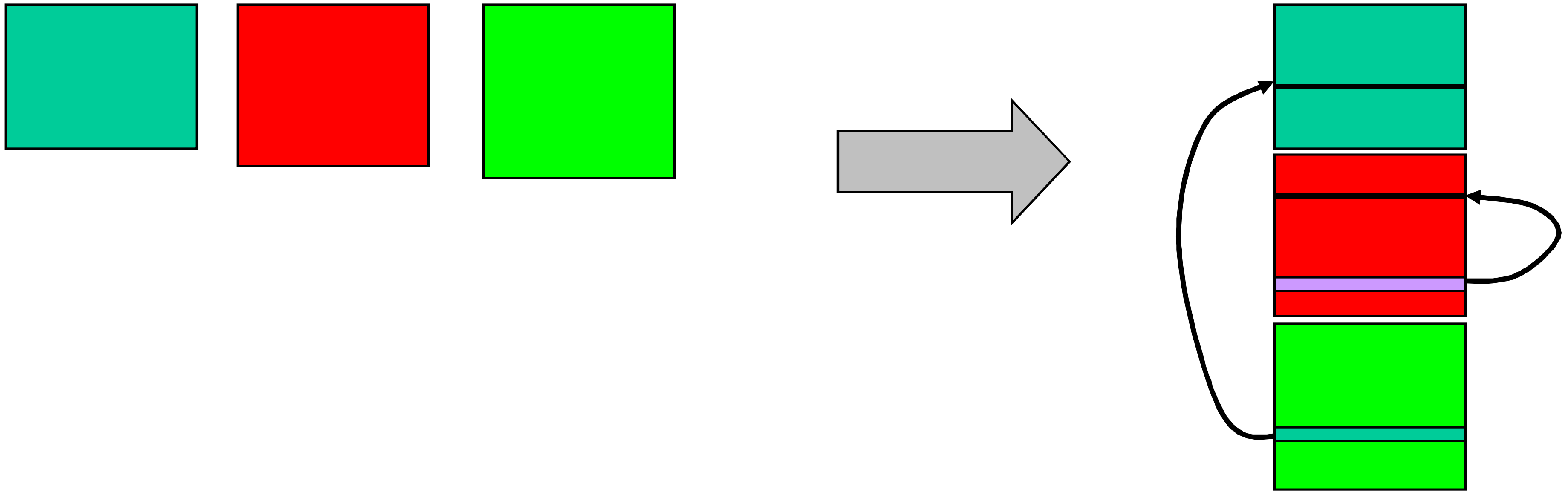
链接器：把多个已编译的模块组合成一个大模块

- 其工作主要是重新确定代码和数据在内存中的位置，对“引用代码和数据”的代码做相应的修改。使得引用与实际相一致.



链接器：汇聚多个子模块成一个大模块

- 保持各个子模块在文件上的独立性；好处是方便替换；替换的原因是有bug，或者进行改进；
- 将多个子模块凝结成一个大模块；

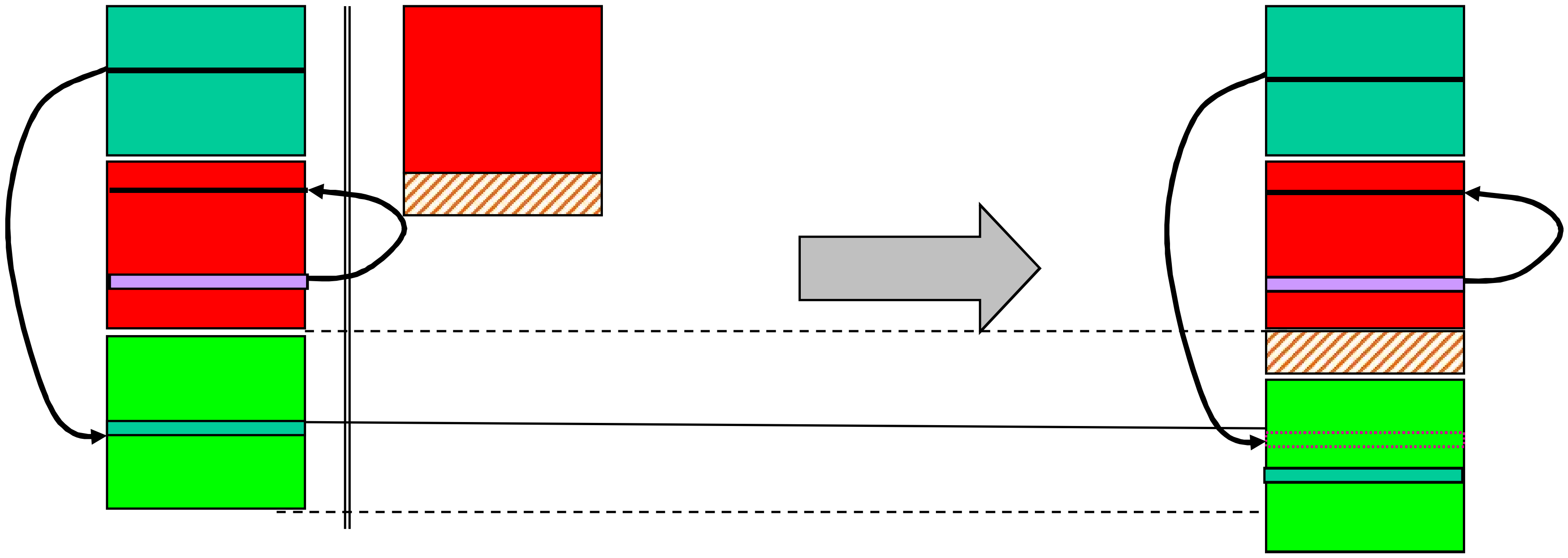


各个子模块在文件上的独立性带来了问题：

引用和实际不再一致

模块升级导致引用和实际不再一致

假设一个程序由多个模块构成，如果其中的某个模块因为版本升级而出现尺寸大小改变时，面临着每个模块在内存中不能加载到链接器在链接生成可执行代码时所期望的内存位置，导致错位;



因此，每次有模块升级时，使用它的应用程序开发商都不得不重新链接，然后重新分发给用户；

解决办法：相对寻址和间接寻址

- 相对寻址解决了同一模块内部的引用，不再受该模块在内存中的加载位置的影响;

```
                 Body
 MOV   EAX,  ptr [04008192]
ADD   EAX, 66;                    //mv1
 CALL  04008164;                  //mf2
```

绝对地址：内存地址

```
                 Body
MOV EAX, ptr [ R -172 ]
ADD  EAX, 66                 //mv1
CALL  ptr [R -188]           //mf2
```

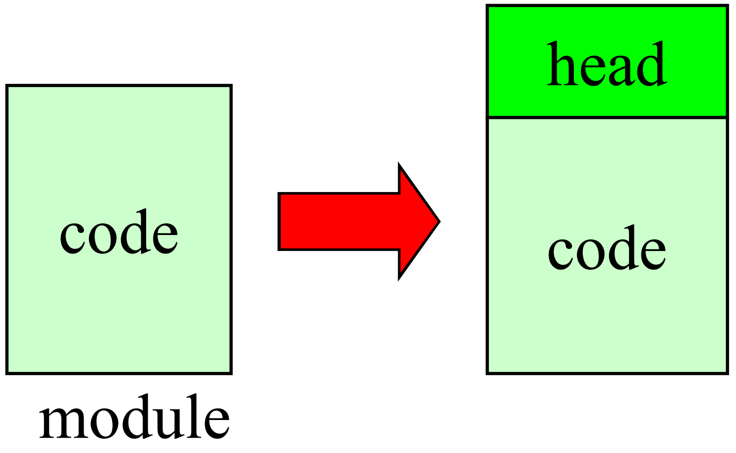
相对地址：
相对该条指令代码的
内存地址

How to calculate the absolute address? IP+相对地址

加载器: 解决外部引用问题

- 对于跨模块边界的引用，由于在链接时无法知道要引用的模块在内存中的加载位置，因此在链接时无法固定下来，只好进一步划分到程序执行的加载时来解决;
- When the program is loaded to execute, the **loader** must **determine the references across module boundary** ;
- 间接寻址;

Indirect addressing: solve external references determination



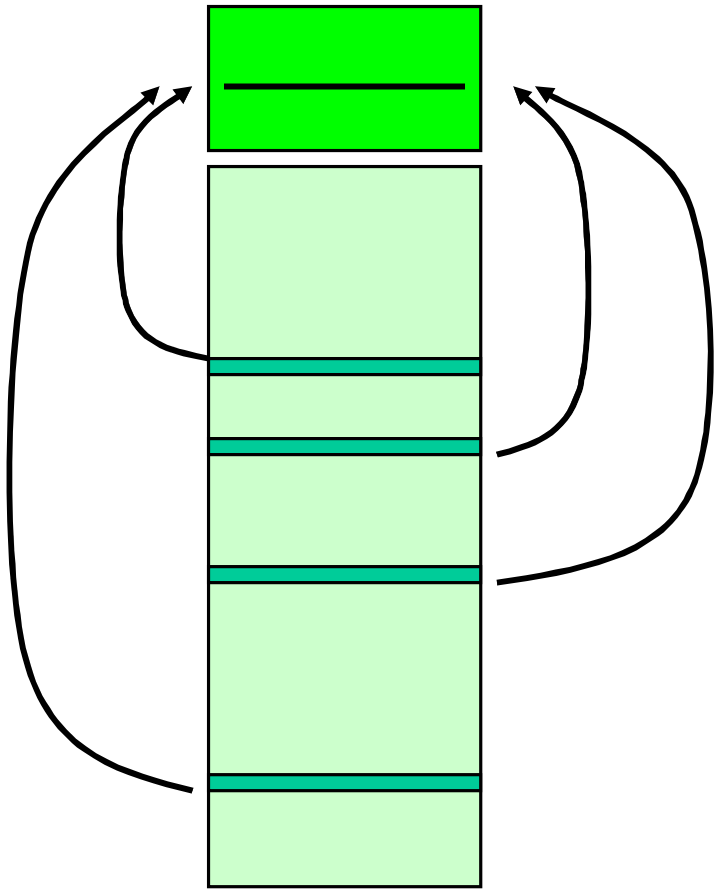
```
MOV   EAX,  ptr [xxxx]
ADD   EAX, 66;              //mv1
 CALL  xxxx;                //mf2
```

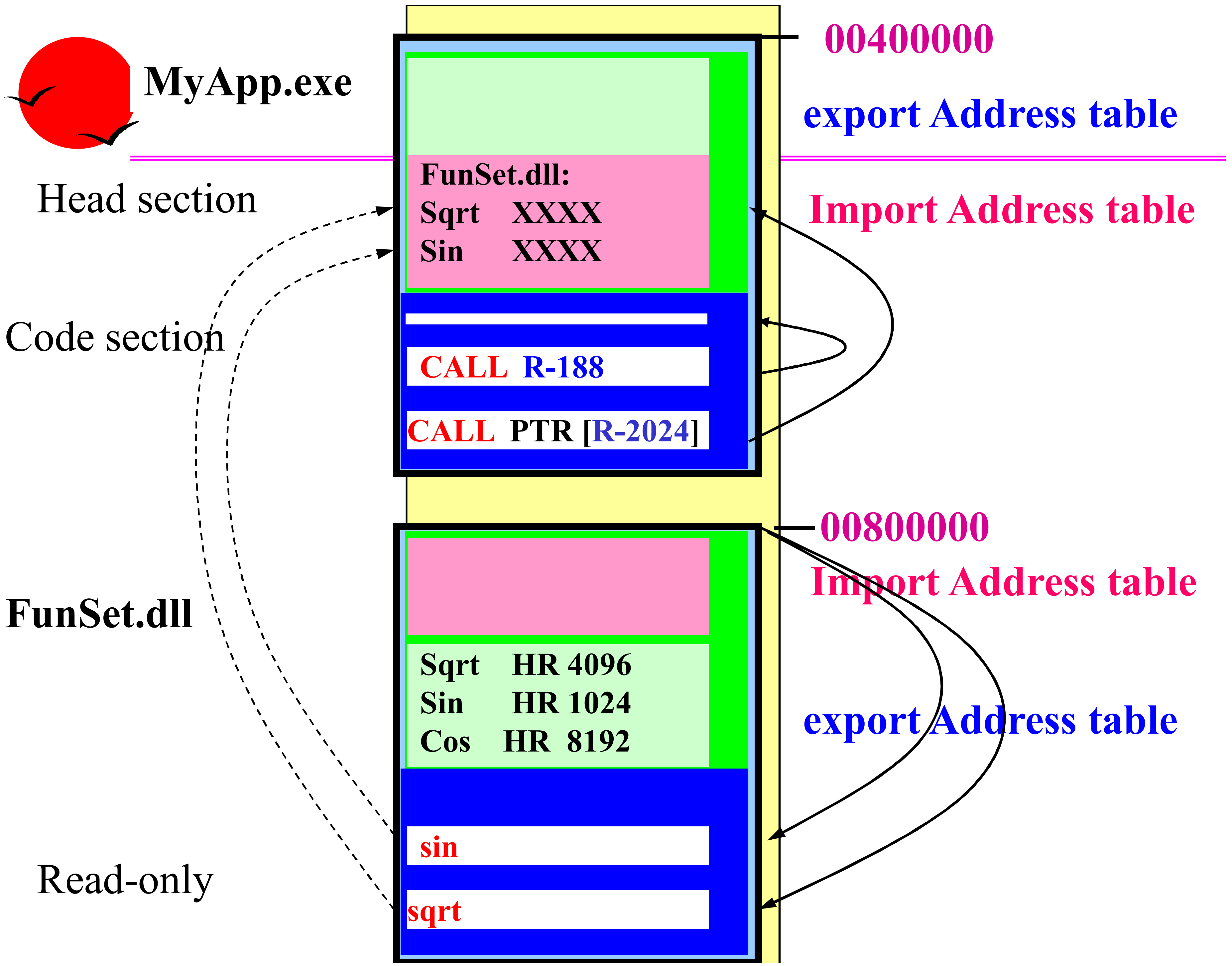
```
MOV EAX, ptr [ptr [R-172] ]
ADD  EAX, 66                 //mv1
CALL  ptr [R-188]            //mf2
```

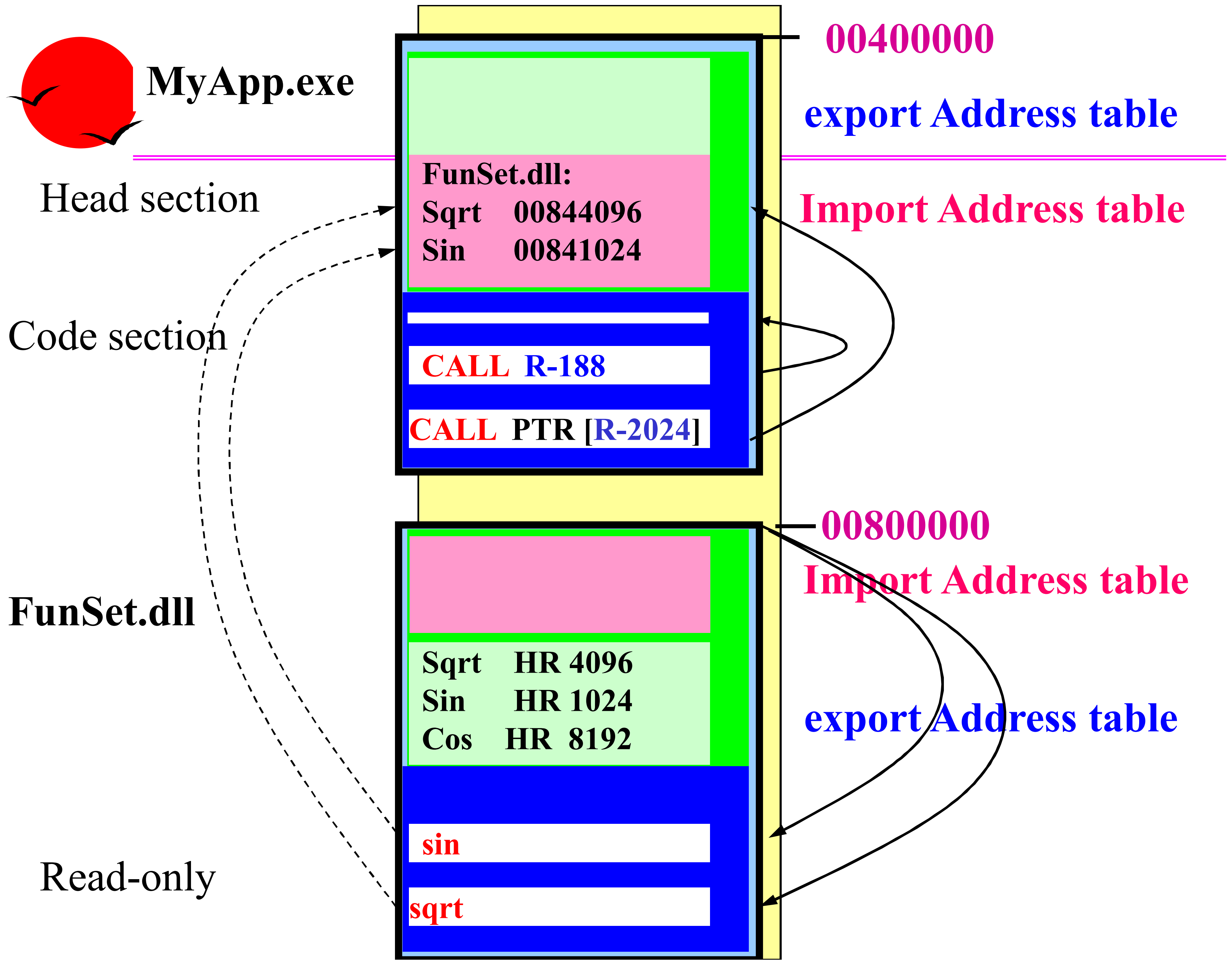
```
Start address = xxxx;
Exported section:
   Function:
      f1,  R 1024;
   Data:
      v1: R 2048;
Imported section:
    module 1:
         Function:
            mf1, xxxx;
         Data:
            mv1: xxxx;
```

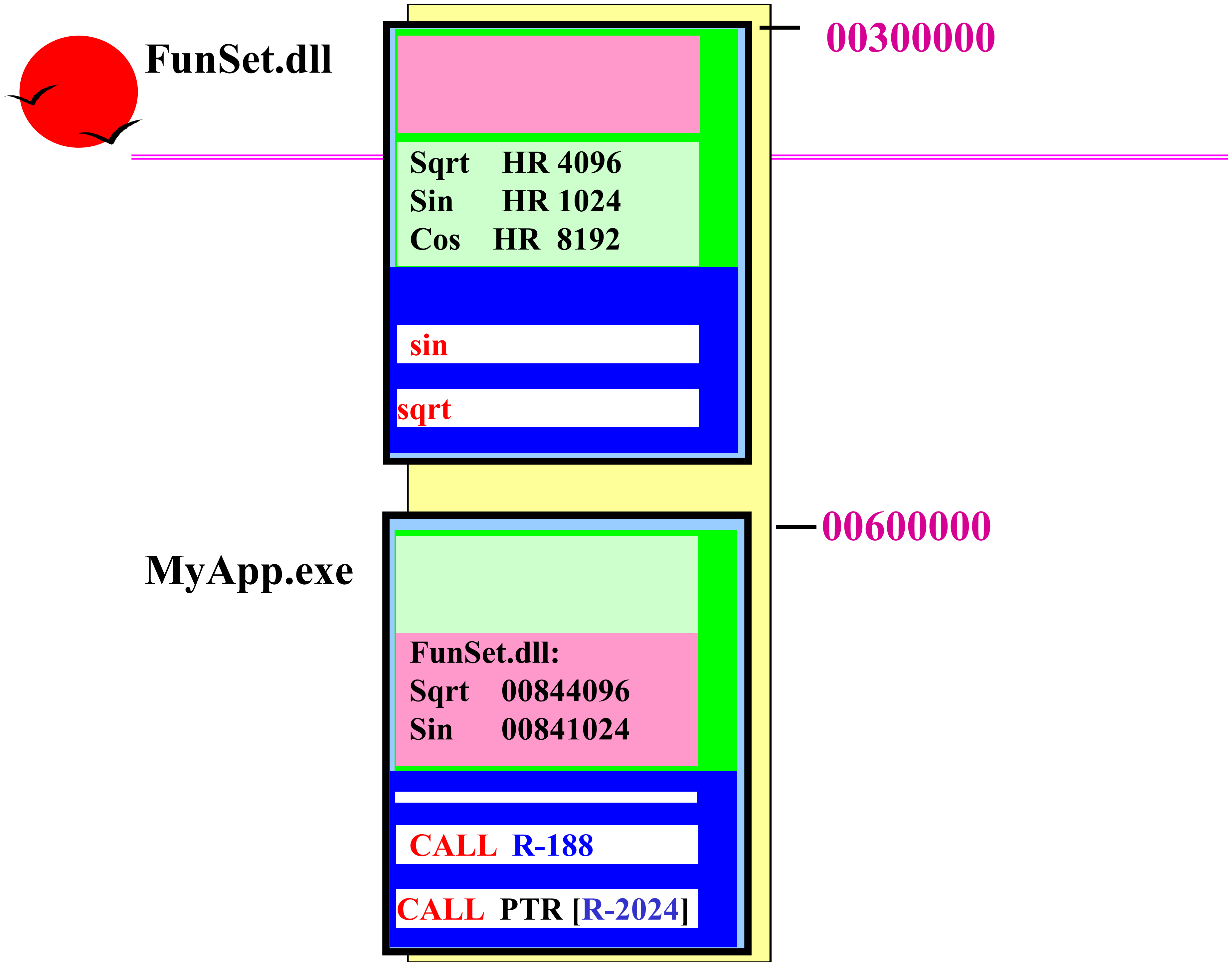
The importance of the head

- A external object might be referenced many times in a module.
- If without head, **code should be modified**, and **multiple places need to change** in code for a external reference.

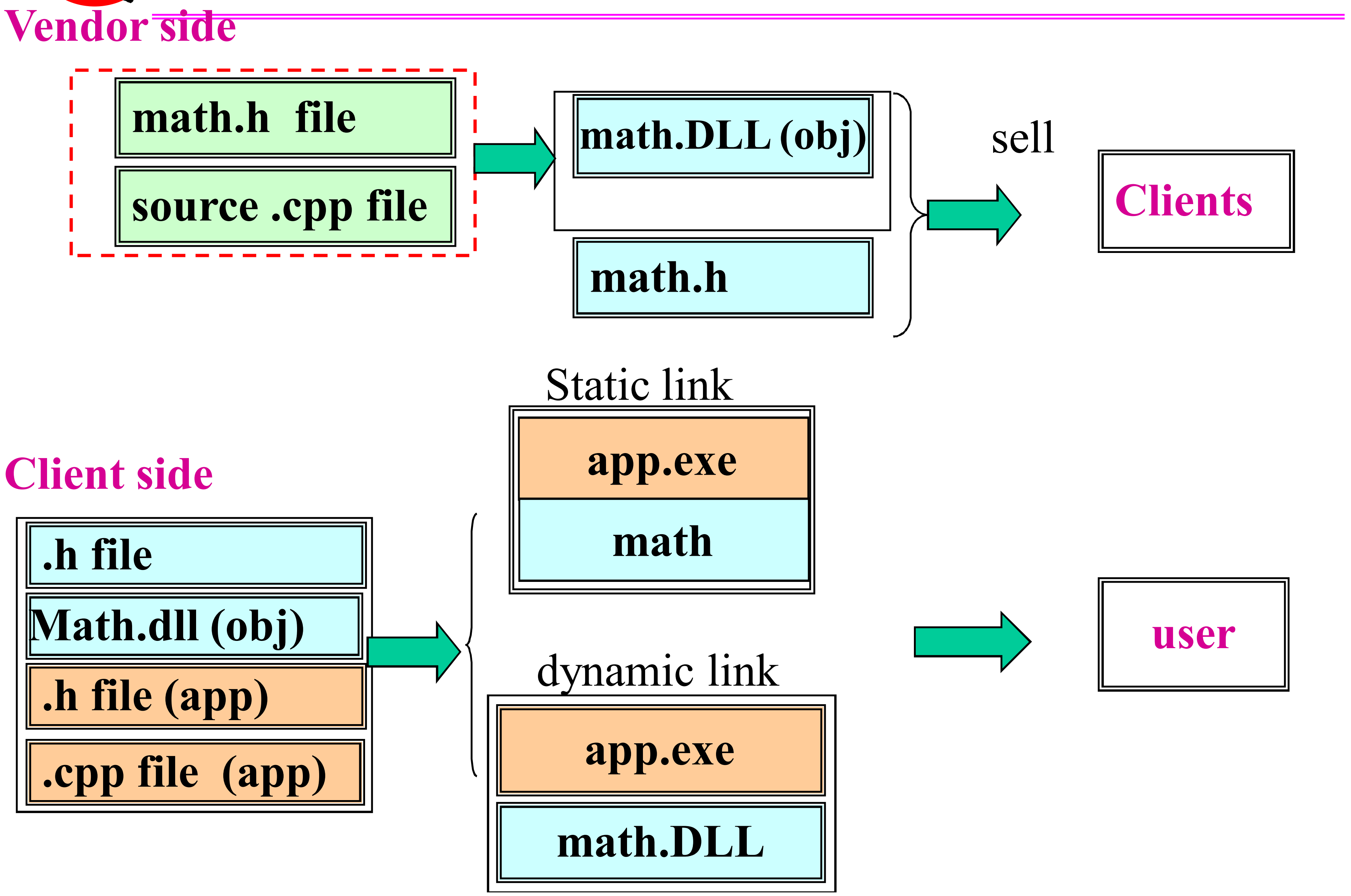



MyApp.exe
00400000
export Address table
Head section
FunSet.dll:
Sqrt XXXX
Sin XXXX
Import Address table
Code section
CALL R-188
CALL PTR [R-2024]
00800000
Import Address table
FunSet.dll
Sqrt HR 4096
Sin HR 1024
Cos HR 8192
export Address table
sin
sqrt
Read-only

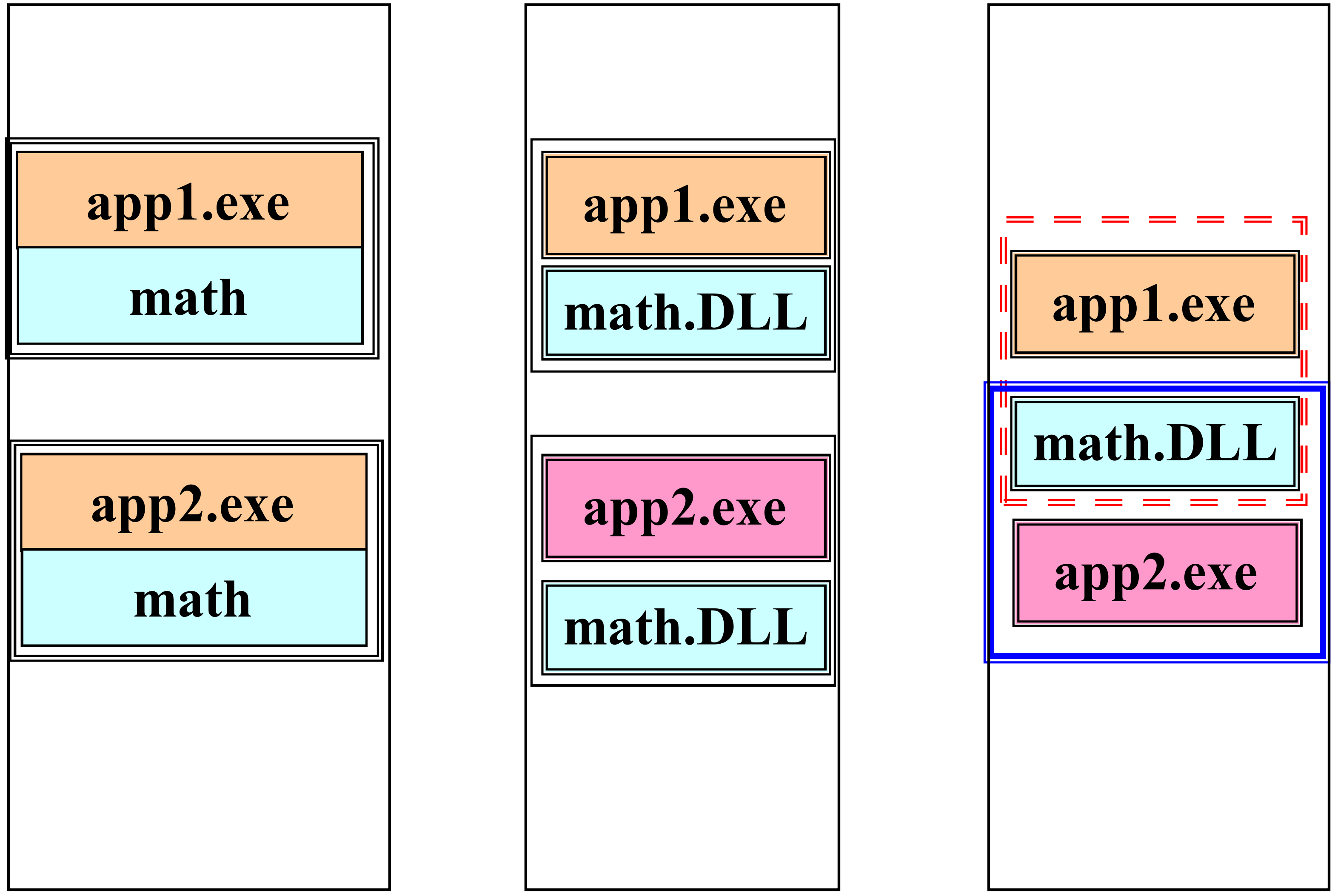

MyApp.exe
00400000
export Address table
Head section
FunSet.dll:
Sqrt 00844096
Sin 00841024
Import Address table
Code section
CALL R-188
CALL PTR [R-2024]
00800000
Import Address table
FunSet.dll
Sqrt HR 4096
Sin HR 1024
Cos HR 8192
export Address table
sin
sqrt
Read-only


FunSet.dll
00300000
Sqrt HR 4096
Sin HR 1024
Cos HR 8192
sin
sqrt
00600000
MyApp.exe
FunSet.dll:
Sqrt 00844096
Sin 00841024
CALL R-188
CALL PTR [R-2024]

A computer runs multiple programs

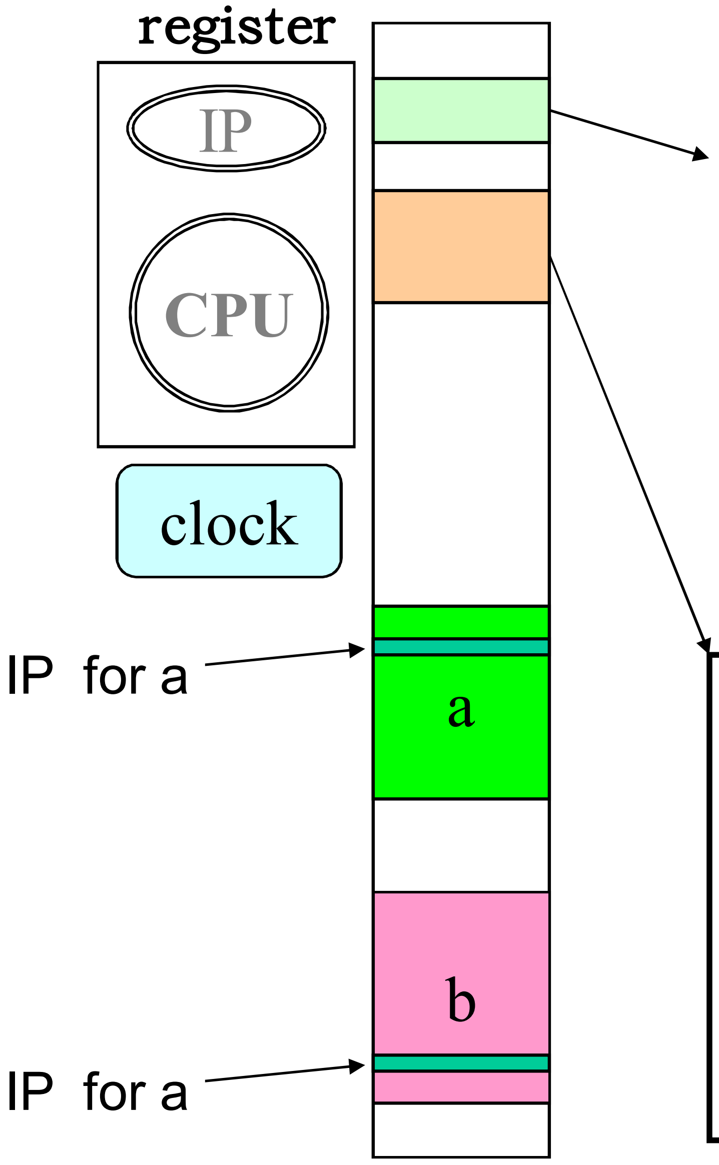


A computer runs multiple programs



Case 1 Case 2: multiple copies Case 3: **desired**

在一个计算机上多个程序的并发执行



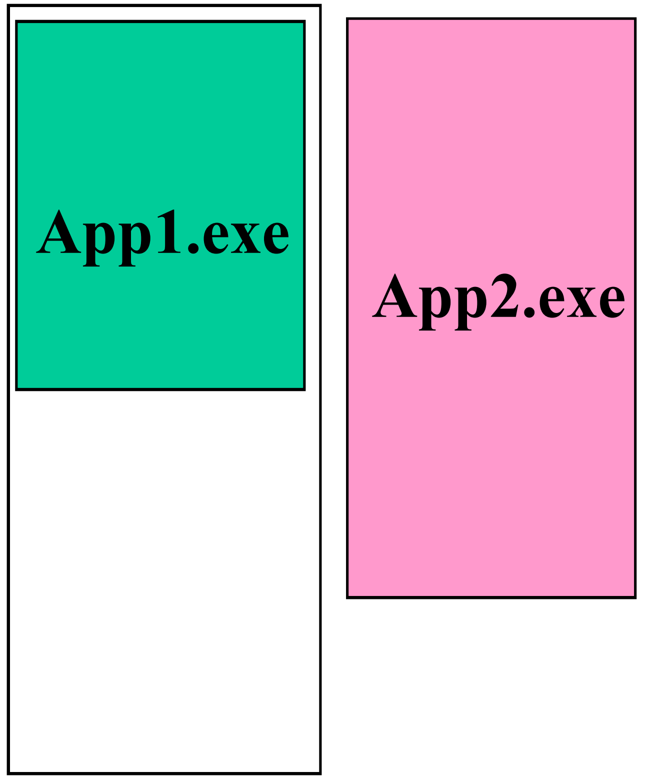
Interrupt Descriptor Table:

1	10240
2	20280
3	40960

Interrupt handler:

```
If current program = 'a'  then
   save the value of IP to IP_Var[1];
   SET  IP, IP_Var[2];
Else
   save the value of IP to IP_Var[2];
   SET  IP, IP_Var[1];
```

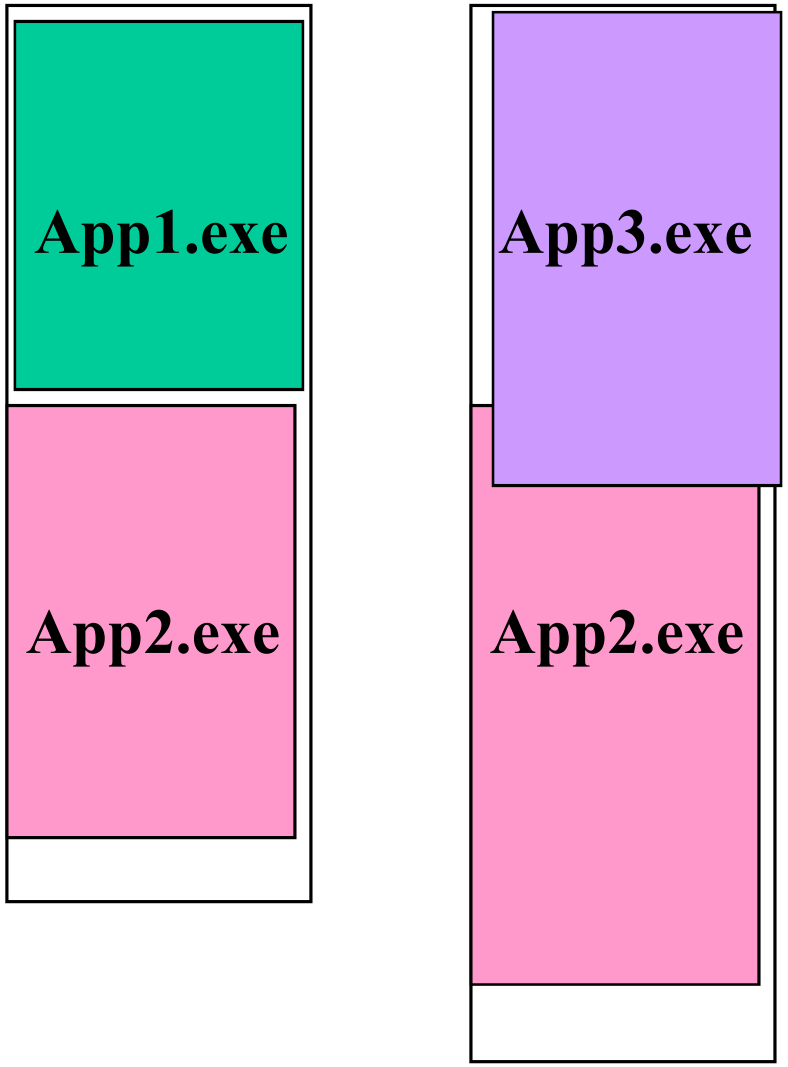
多程序并发运行的问题1：冲突



如何解决冲突?

- 两个程序在编译链接时，都被设定要求加载到内存的同一起始位置；
- 因为两个程序由不同的开发者独立地使用链接程序链接而成，都以0地址作为自己的开始地址；

解决方案1：重新链接

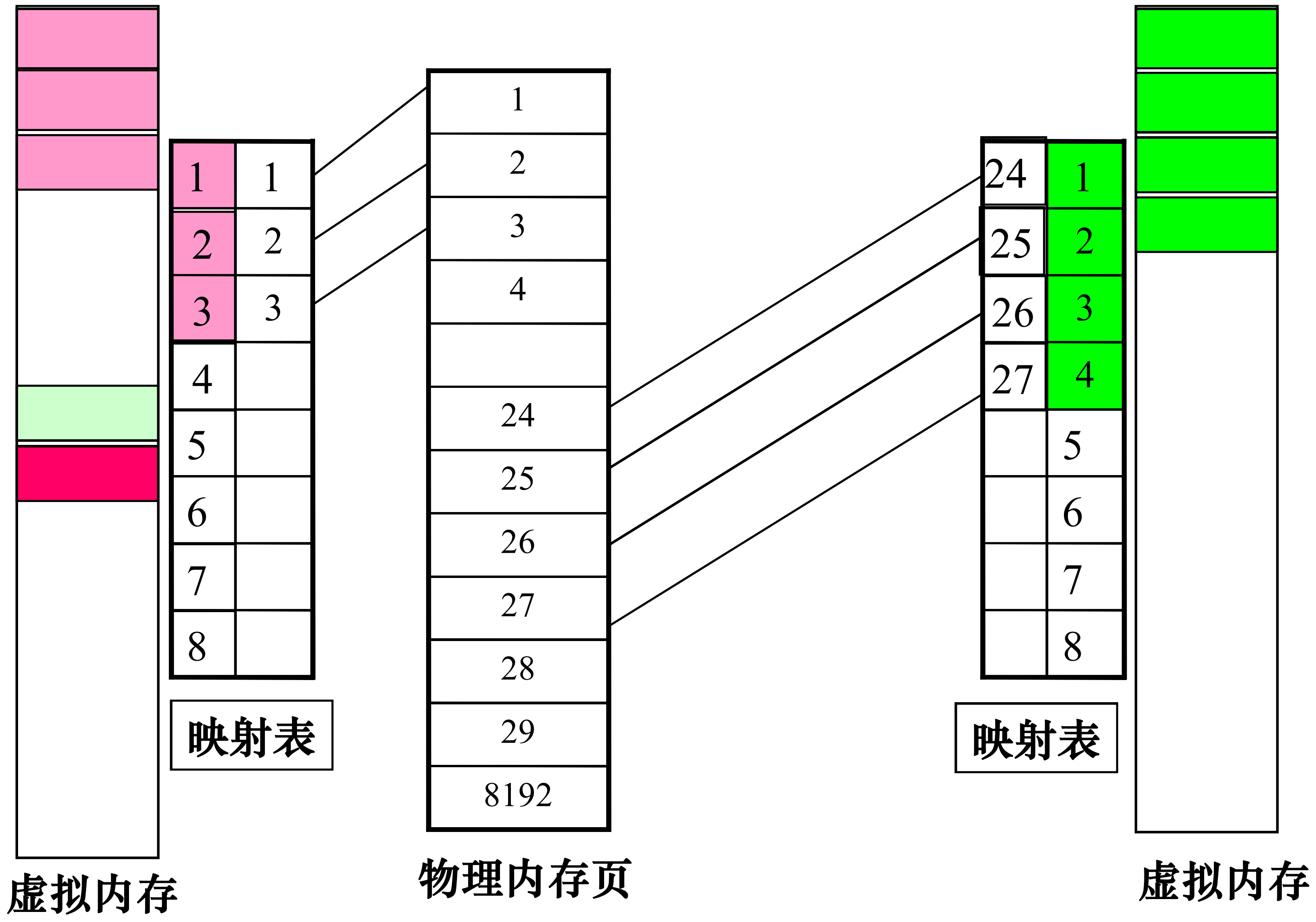


这种解决方法不具备组合能力；

App3.exe is conflicted with app2.exe;

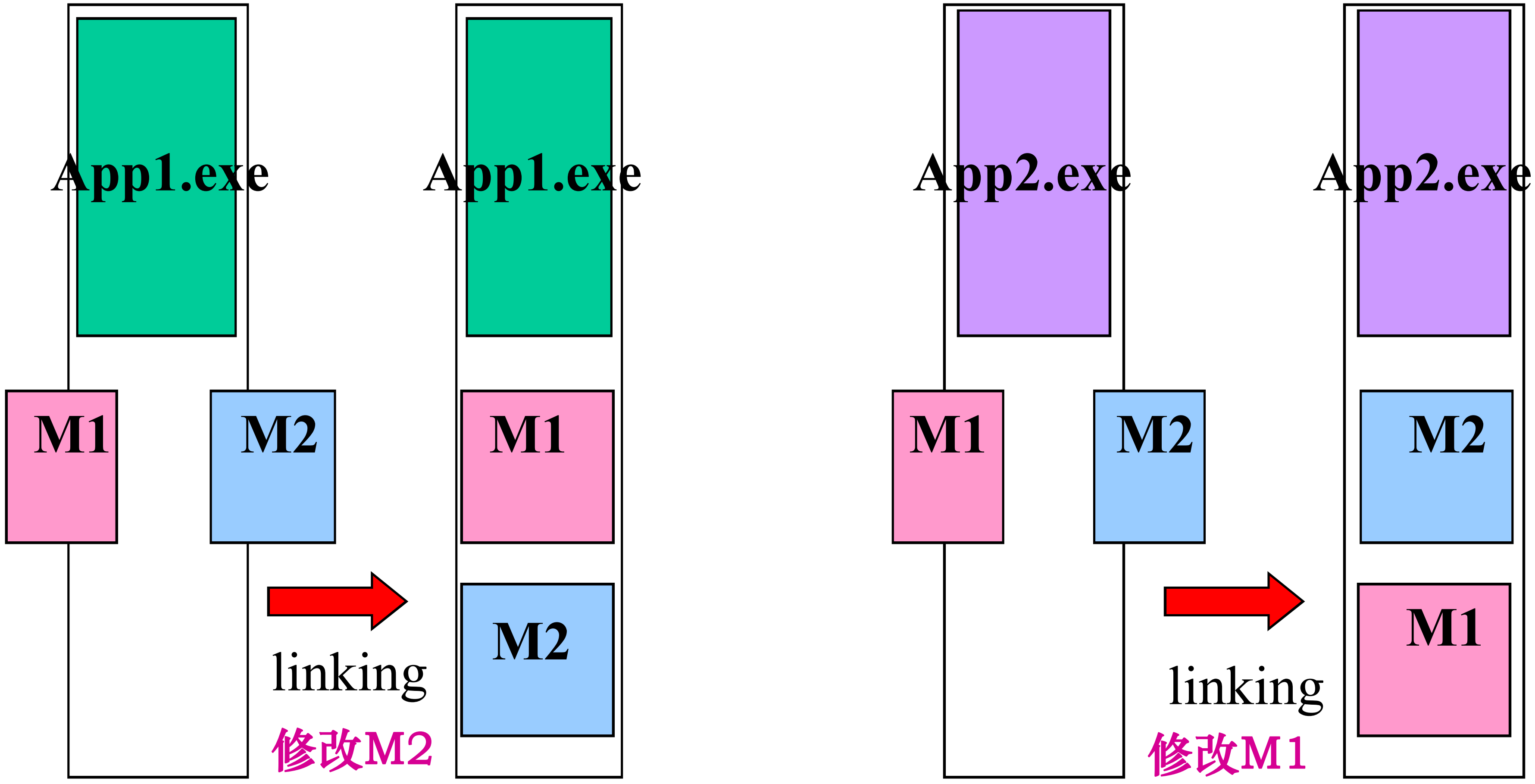
App3.exe is conflicted with app1.exe;

解决方案2：虚拟内存



1
2
3
4
5
6
7
8
映射表
虚拟内存
1
2
3
4
24
25
26
27
28
29
8192
物理内存页
24
25
26
27
映射表
虚拟内存

问题 2: 共享模块在内存中重复



每个程序都要有一个自己的一个拷贝，导致内存利用不高；

当两个应用程序运行在一台机器上时，**无法共享**

问题 2 的解决方法: 相对寻址

- 对于同一模块内的引用，相对寻址使得模块不管被加载在内存中的哪一个位置，模块内的定位都没有问题；
- 实现了代码可在编译链接时可固定下来，保持不变；

```
MOV   EAX,  ptr [04008192]
ADD   EAX, 66;                    //mv1
 CALL  04008164;                  //mf2
```

绝对地址

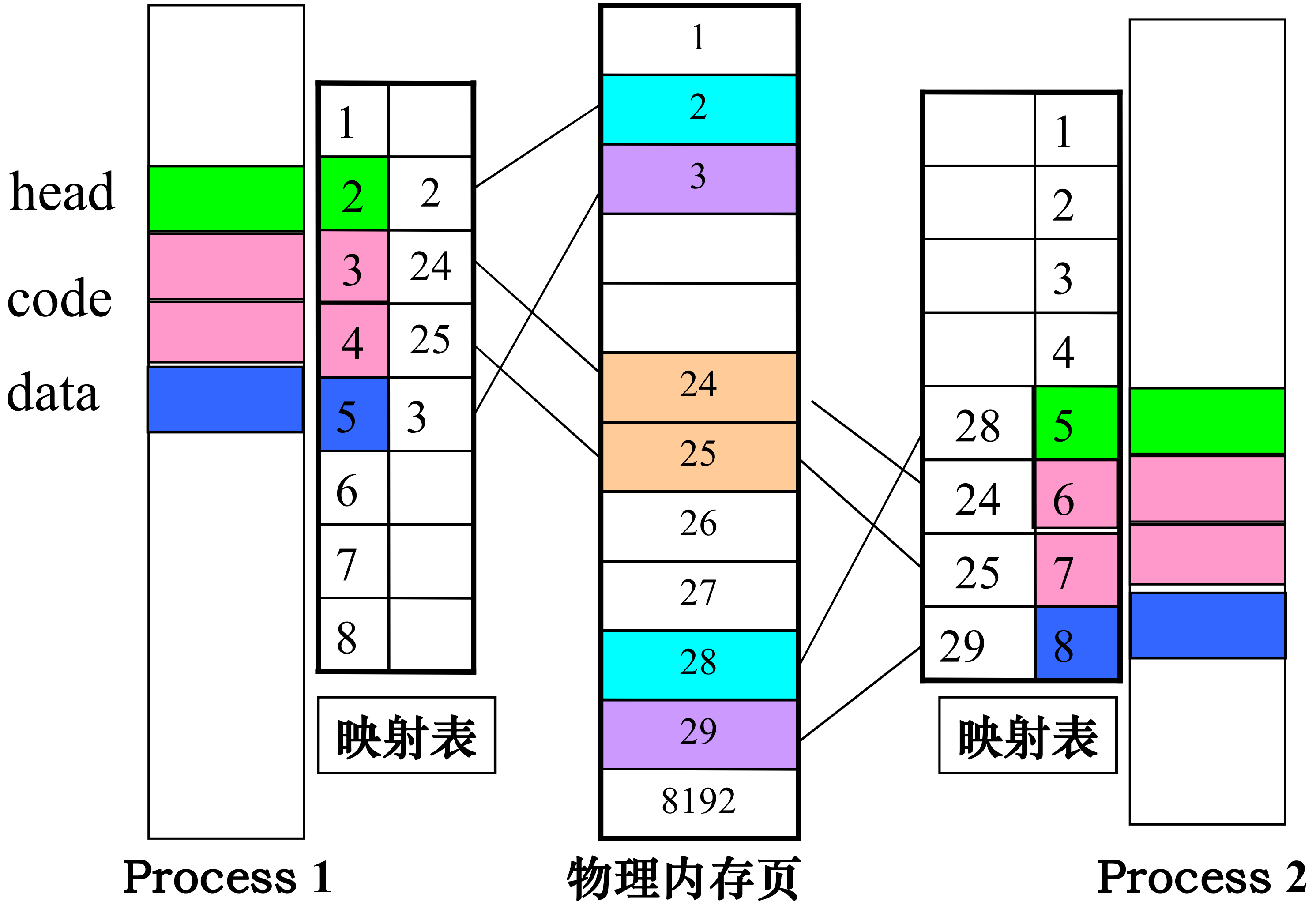
```
MOV EAX, ptr [ R -172 ]
ADD  EAX, 66                  //mv1
CALL  ptr [R -188]            //mf2
```

相对地址

如何从相对地址，推算出绝对地址？

对于共享模块，相对寻址取得了其代码部分可在内存中存一份就够（理想状态）

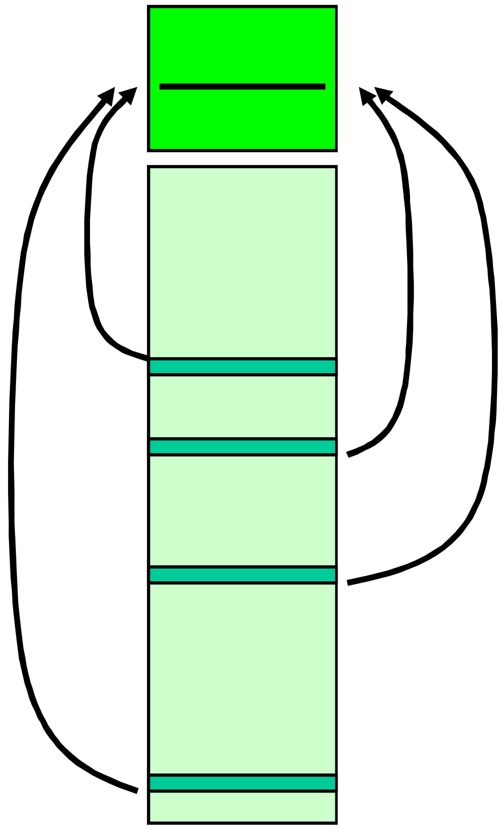
共享模块的head部分在内存中不能只一份，而是每个程序一份?



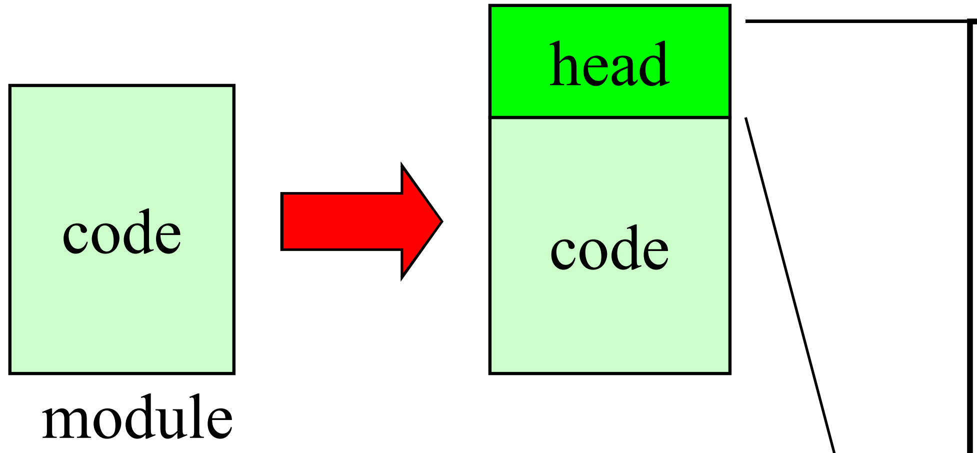
Head section 的重要性

在一个模块中，对其而言的一个外部对象，可能在该模块中被反复引用多次。如果**没有模块头**，那么在加载时，该模块的代码部分中所有的对外部对象的引用之处都需要修改，带来2个问题：

- 加载时，要对所有代码进行扫描解析，并进行修改，**加载效率低;**
- 代码的修改会导致代码部分不能共享，要求每个程序都要有自己专有的拷贝，**内存利用率低;**



间接寻址: 解决了外部对象引用的定位问题



```
Start address = xxxx;
Exported section:
   Function:
      f1,  R 1024;
   Data:
      v1: R 2048;
Imported section:
    module 1:
         Function:
            mf1, xxxx;
         Data:
            mv1: xxxx;
```

```
MOV   EAX,  ptr [xxxx]
ADD   EAX, 66;              //mv1
 CALL  xxxx;                //mf2
```

```
MOV EAX, ptr [ptr [R-172] ]
ADD  EAX, 66                   //mv1
CALL  ptr [R-188]              //mf2
```

提问

- 相对寻址需要硬件的支持，一些机器并不支持相对寻址，如何来使用软件方法解决?

- 当一个模块被多个程序共享时，如果没有相对寻址，共享模块就不得不每个程序都搞一个拷贝，而且加载时，要全扫描解析，对每处定位的地方都作修改；

  这就是为什么Dephi**启动时很慢**的原因。

- 一些处理器并不支持间接寻址，该如何处理?

多程序并发执行

相对寻址解决了模块内部的引用问题，间接寻址则解决了模块间的引用问题。虚拟内存则实现了一机变多机；

虚拟内存的好处：原有的单程序占有整个内存的编译和执行概念没有改变，过去的程序依然可执行，即实现了兼容性；

同时又支持多程序并发执行；

付出的代价：

在空间上，多了映射表，要消耗内存；

在时间上，要做映射转换，每个虚拟内存地址都要映射成物理内存地址；

相对寻址/间接寻址/虚拟内存三者关系

有些老处理器，或者简单处理器（便宜）**并不支持相对寻址，和间接寻址**，针对这类处理器：

编译/链接时施行**静态链接，**即链接时把一个应用程序的所有模块合成一个可执行模块。链接时就确定这个可执行模块在内存的加载地址，然后，对所有引用（包括数据和代码（if，else））都确定其内存**绝对地址。这种程序启动就很快，执行效率也高。（为什么？）。**

这种方式对嵌入式系统很有效，因为这样的处理器所执行的应用程序单一，功能也固定；

这种应用程序一般固化在处理器上，因功能单一，简单，**也很少要修改和升级。**

相对寻址/间接寻址/虚拟内存三者关系

采用绝对地址寻址方式编译/链接的可执行应用程序，在支持**虚拟内存**的机器上，并不需要做任何改动，照样可以执行，**实现了前向兼容性。程序不要修改，增添了多程序并发执行功能，**

付出的代价：

- 在空间上，多了映射表，要消耗内存；
- 在时间上，要做映射转换，每个虚拟内存地址都要映射成物理内存地址；

多程序并发执行

接下来的问题：

多程序并发执行实现后，接下来的主要矛盾是：**共享问题**；

多个程序共享一个模块：

策略：**代码和数据的分离**；

数据不共享，只有代码共享；

引入的**新概念**：**栈**，**堆**，**线程**；

代码重入, 函数递归调用, stack ,Thread

虚拟内存实现了每个应用程序在编译时，就可确定自己要执行时，在内存中的加载地址；多个程序并发执行不会冲突；

相对寻址解决了模块内部的引用问题，**间接寻址**则解决了模块间的引用问题。

虚拟内存与相对寻址，间接寻址是彼此**相互独立的概念**，但都是为了实现**代码部分的事先确定，执行时不需修改，即固定不变性；**

它们也解决了**共享性问题，提高了加载效率，和内存有效利用率；**

代码重入，函数递归调用，stack ,Thread

这些概念和技术，使得一个应用程序可由独立的模块文件组成，**实现了松耦合，大解放，模块的替换（升级，或者更好的替代者），非常容易简单**。也使得应用程序模块之间**原有的依赖关系被打破**，也就是说，**不再受某个模块（即某个厂家产品的产品）的绑定**，可以拿另一个替代品来替换，这种替换根本就不依赖与应用程序的厂家来完成，应用程序的使用者自己搞定。

我们知道，像宝马汽车，很多人买得起，但是养不起。如果也采用了软件这种技术，那么**人人都可开上宝马了**，正如我们**人人都使用电脑/手机一样。**

软件一旦被解放，就出现蓬勃发展，到处开花，正如福特所说：每个人都开上汽车。因此也成为了时代的主流，即信息时代。

代码/数据的访问的频繁程度不同

频繁访问的内容：

- 虚拟内存的映射表；
- 中断处理例程；
- 线程的上下文；

因此这些内容应该放在二级Cache内存中；

预测问题：

当前线程不再访问的数据，调出Cache内存，下一线程要访问的数据和代码调入cache内存；

代码重入, 函数递归调用, stack ,Thread

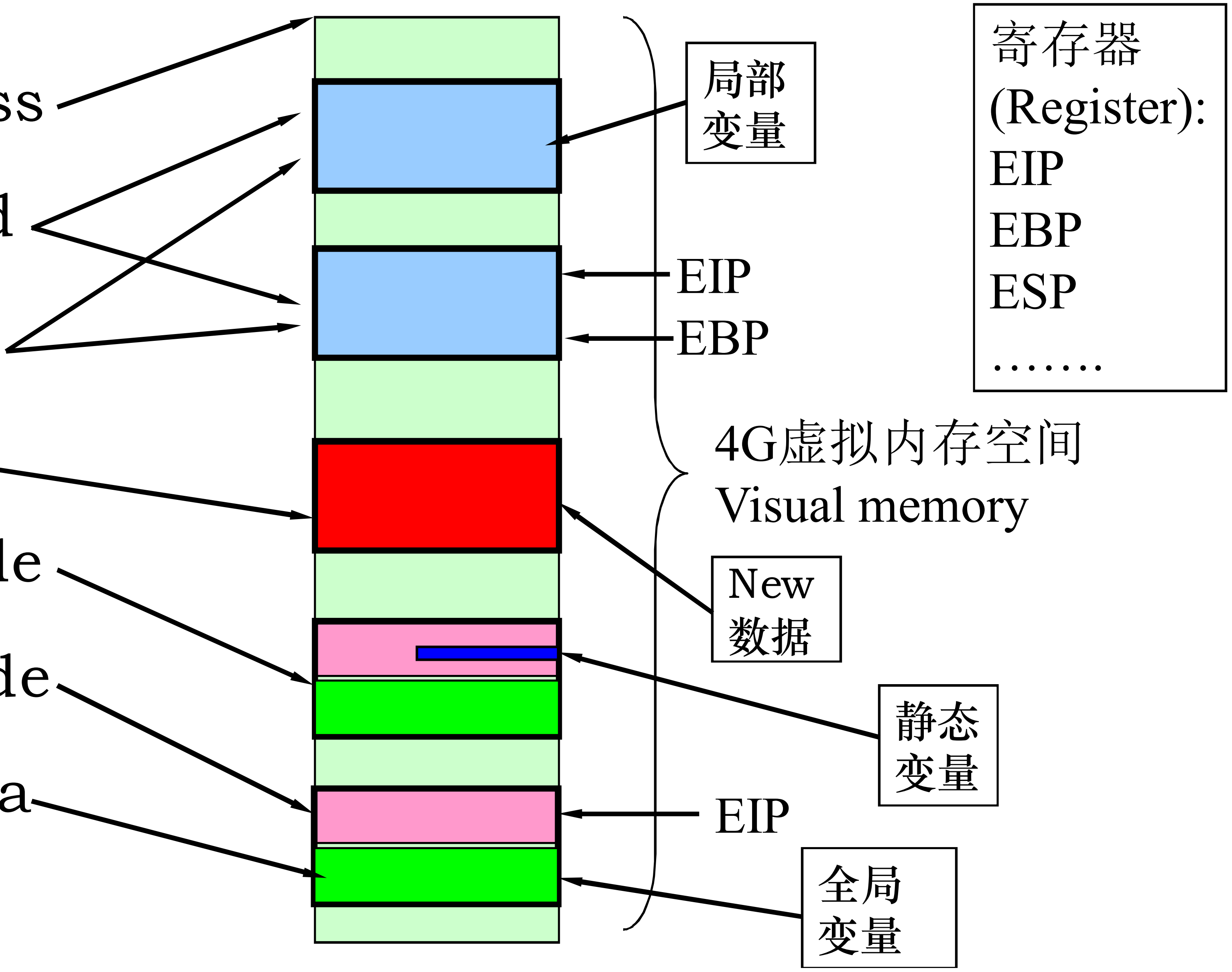
Share module improves memory efficiency, thread and stack further improve it;

面向过程中的基本概念:

- 代码：函数, 和函数的调用;
- 数据：局部变量，全局变量，静态变量，new数据;

程序执行机制－进程模型

- process
- thread
- Stack
- Heap
- module
  - Code
  - data


局部变量
寄存器 (Register): EIP EBP ESP ……..
EIP
EBP
4G虚拟内存空间 Visual memory
New 数据
静态变量
EIP
全局变量

Code re-entry, function recursive calling, stack ,Thread

```
int g_myGlobalVariable;
int main( int argc, char *argv[ ] )  {
   char szBuf [128];
   char *psz = "Hello";
   unsigned long   p = 2;

   g_myGlobalVariable = 0x12345678;
//Procedure invocation :
  MyFunction( p );
}
```

函数调用的原理

```
int g_myGlobalVariable;
int main( int argc, char *argv[ ] )  {
   char szBuf [128];
   char *psz = "Hello";
   unsigned long   p = 2;

   g_myGlobalVariable = 0x12345678;
//Procedure invocation :
  MyFunction( p );
}
```

函数调用的原理

```
int main( int argc, char *argv[] )     {
•   401000:   PUSH      EBP
•   401001:   MOV       EBP,  ESP
•   401003:   SUB       ESP, 00000088
•   401009:   PUSH      EDI
•   40100A:   MOV    DWORD PTR [EBP- 00000084],  00406030
•   401014:   MOV    DWORD PTR [EBP- 00000088],  00000002
•   401036:   MOV       DWORD PTR [004088E8], 12345678
•   4010E1:   MOV       EAX, DWORD PTR [EBP - 00000088]
•   4010F9:   CALL      DWORD PTR [0x00405030]
•   4010FE:   ADD       ESP, 4     //清除传递参数
}
```

We can see that no static space appears for local variable.

全局变量的弊端在哪里？

函数调用的原理

```
int MyFunction (int anArgument) {
    int aVariable;
    aVariable = anArgument;
    return aVariable;
}
```

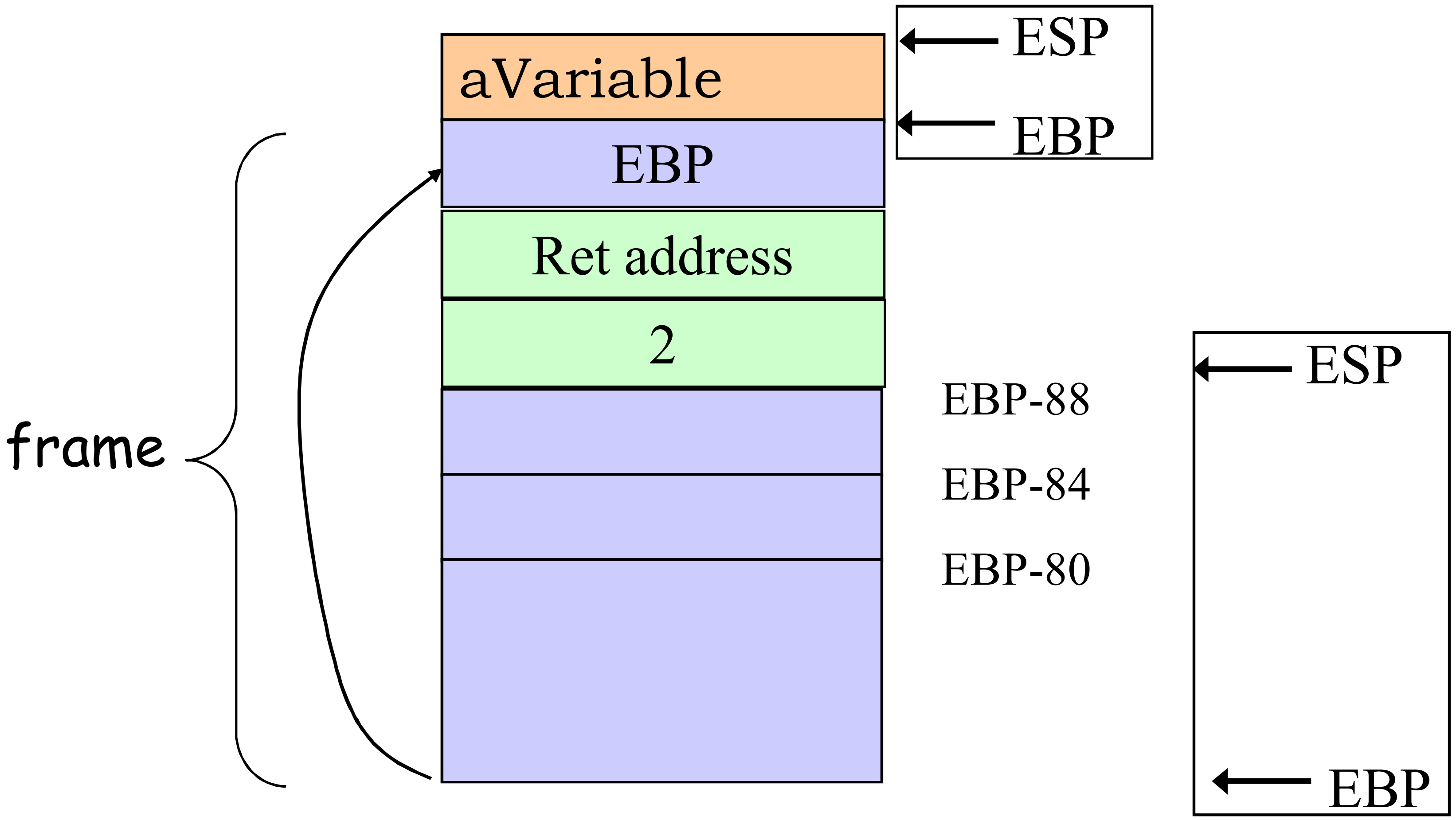
Assmebly:

```
PUSH EBP ;            //保存调用者的stack frame的顶
MOV EBP, ESP ;        //建立本函数的stack frame底
SUB ESP, 4 ;          //分配局部变量空间
MOV EAX, [EBP+8] ;    //取出传递参数
MOV [EBP-4], EAX ;    //写局部变量
MOV EAX,[EBP-4] ;     //写返回值
MOV ESP, EBP ;        //释放本函数的stack frame
POP EBP ;             //恢复调用者的stack frame底
RET ;                 //弹出pop返回地址到EIP;
```

Register使用频率非常高

Stack结构－链表结构

Stack frame(栈帧)

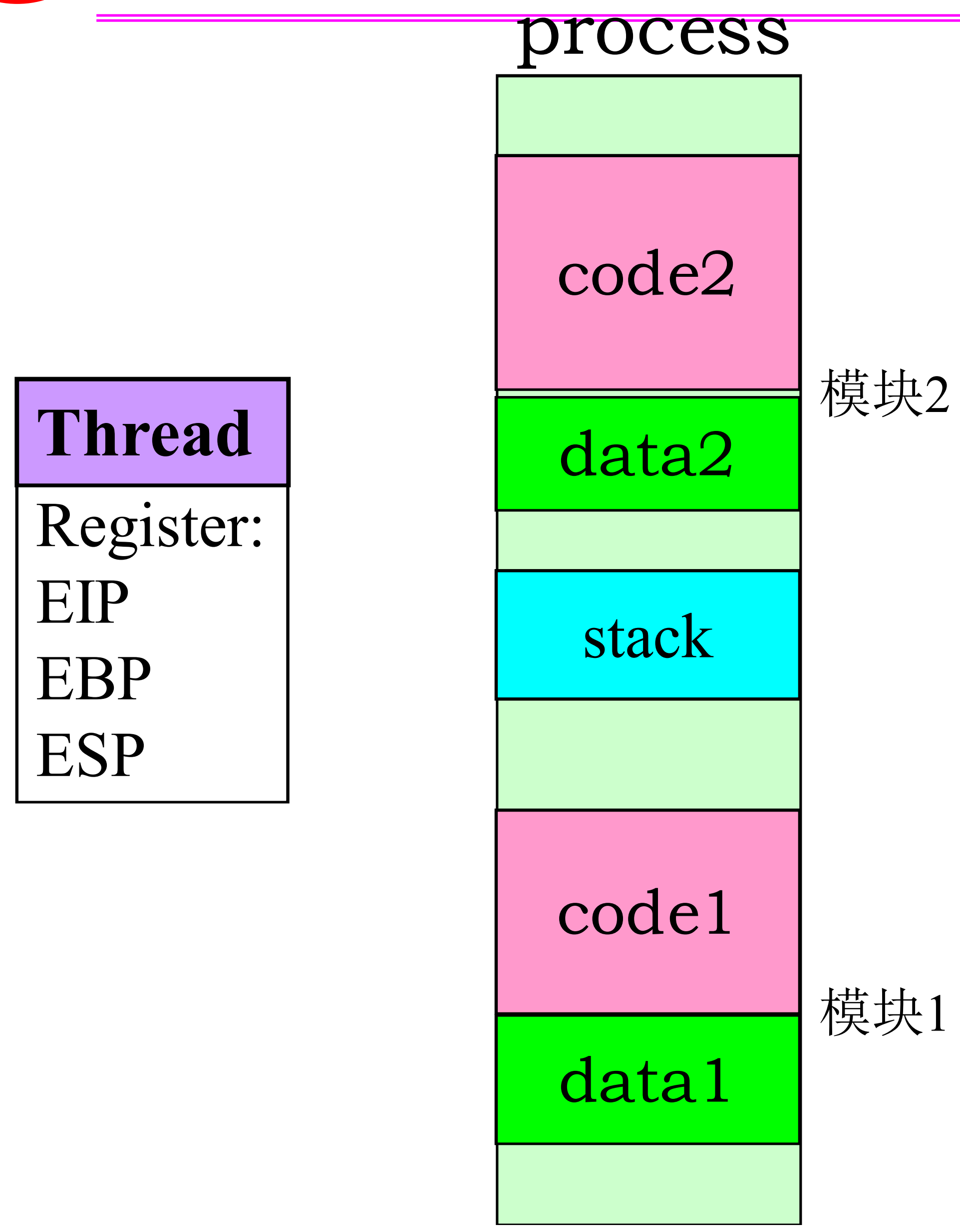


note

- We can see very high use frequency of registers;

- **Benefits from stack:**
  - Code re-entry;
  - function recursive calling;
  - Multiple threads;
  - Efficient memory utilization;

同一进程内的互操作



互操作的形势：**函数调用**

互操作中的问题：

1） **参数如何传递?**

2）代码和数据如何定位?

参数传递媒介；
参数位置;
参数顺序；

函数调用规范

A calling convention is a contract between the caller and the callee.

1) Arguments transfer agreement: argument length, order;
2) Where are the return values placed, and their length;
3) Who is responsible for cleaning up the stack;
4) procedure naming specification;
5) Saving and restoring the registers, if they are used in the procedure;

函数调用规范

Keyword	**Stack Cleanup**	**Parameter passing**
__cdecl	Caller	Pushes parameters on the stack in reverse order
__stdcall	Callee	Pushes parameters on the stack in reverse order
__fastcall	Callee	Stores parameters in registers, then pushed on stack
thiscall	Callee	Pushed on stack; **this** pointer stored in ECX

Other Calling Conventions: **__pascal**, **__fortran**, and **__syscall**

__cdecl

It is the default calling convention for C and C++ programs, and support **vararg** procedures.

For implementer (provider), there are two things:

int **__declspec(dllexport) __cdecl** MyFunc(char c, short s, int i, double f) {…….}

Offer 3 files to clients (.h file,.lib file,.dll file);

In .h file:

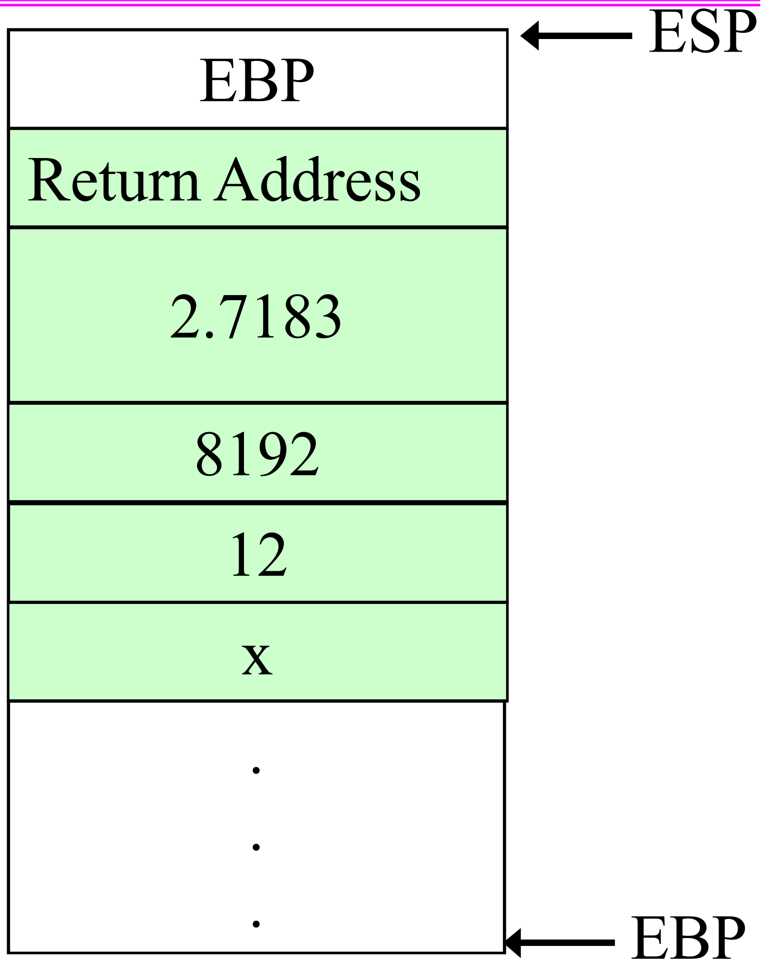
int **__declspec(dllimport) __cdecl** MyFunc(char c, short s, int i, double f);

For a client, just first include the .h file, then call it in the program:

MyFunc ('x', 12, 8192, 2.7183);

__cdecl

- Pushes parameters on the stack in reverse order;
- Return value is in EAX register;
- The caller cleans up the stack;
- The C decorated function name is "_MyFunc."

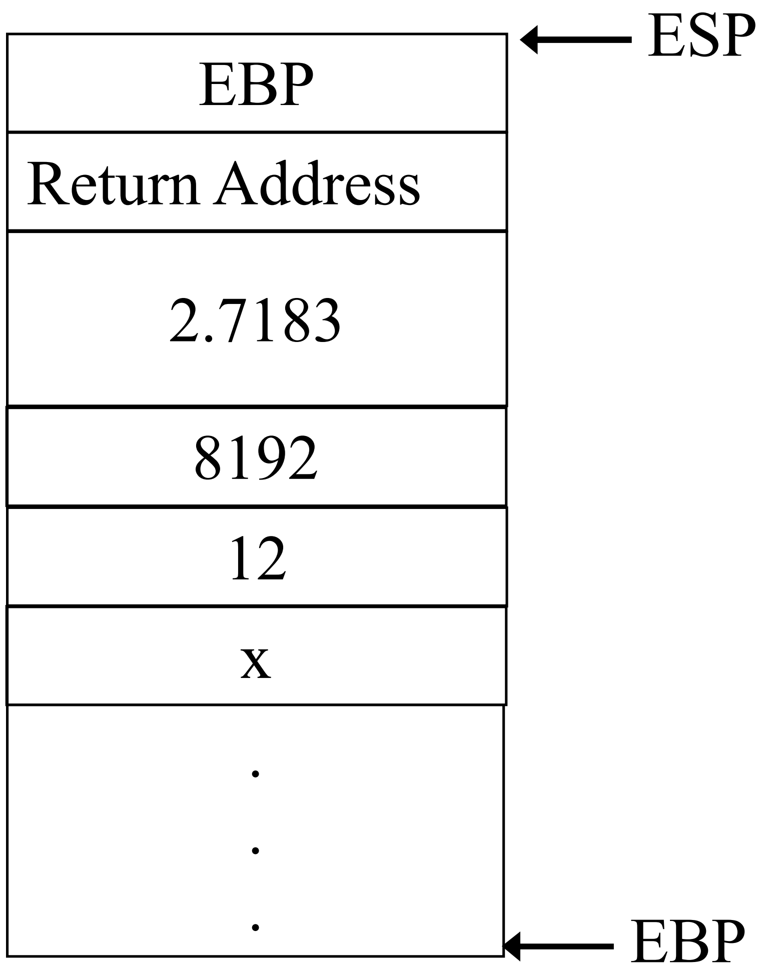


Stack frame

ECX and EDX are not used

__stdcall

- For Win32 API functions
- #define WINAPI __stdcall
- Pushes parameters on the stack in reverse order;
- Return value is in EAX register or EDX:EAX;
- The callee cleans up the stack;
- The C decorated function name is "_MyFunc@20."



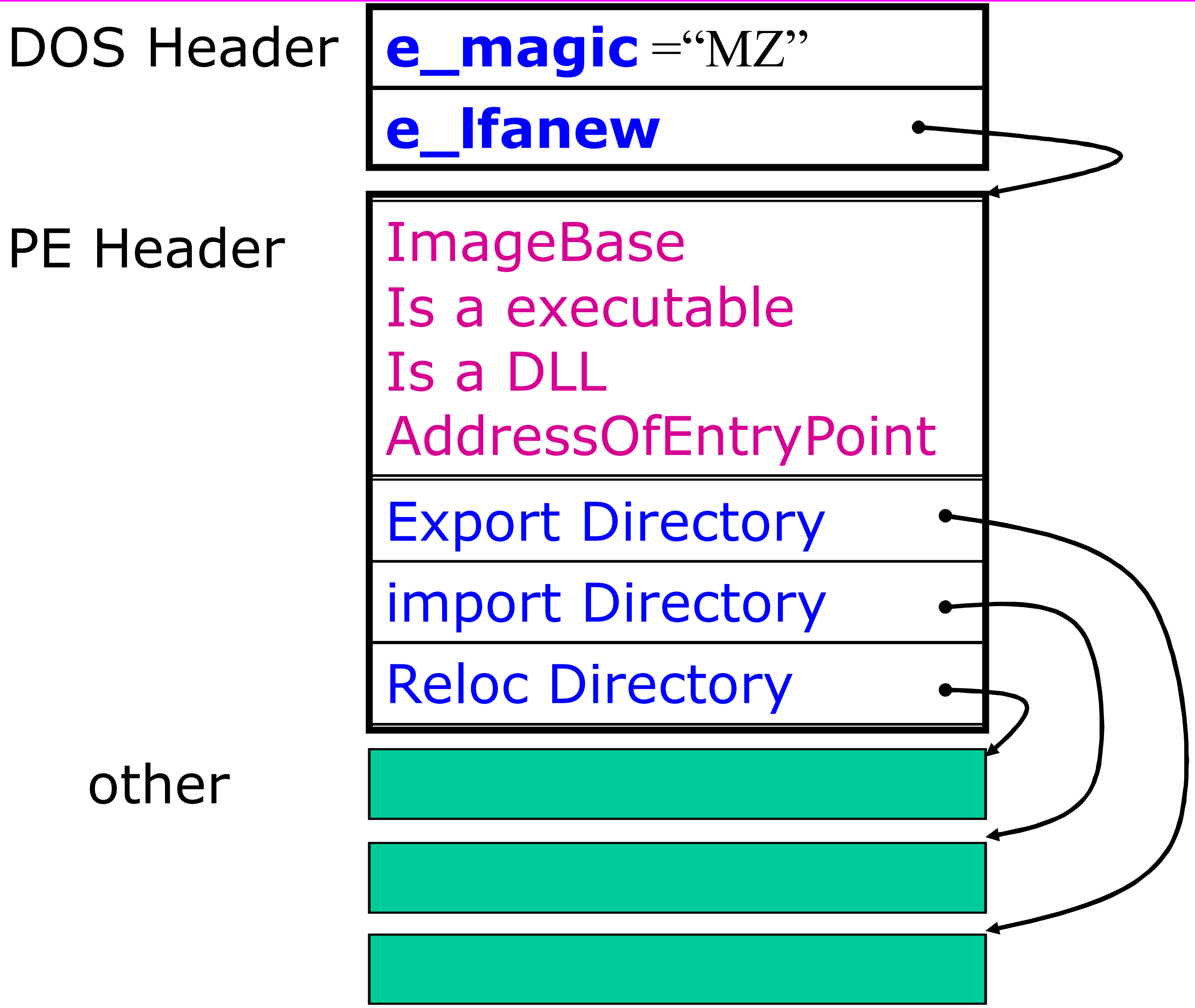
Stack frame

ECX and EDX are not used

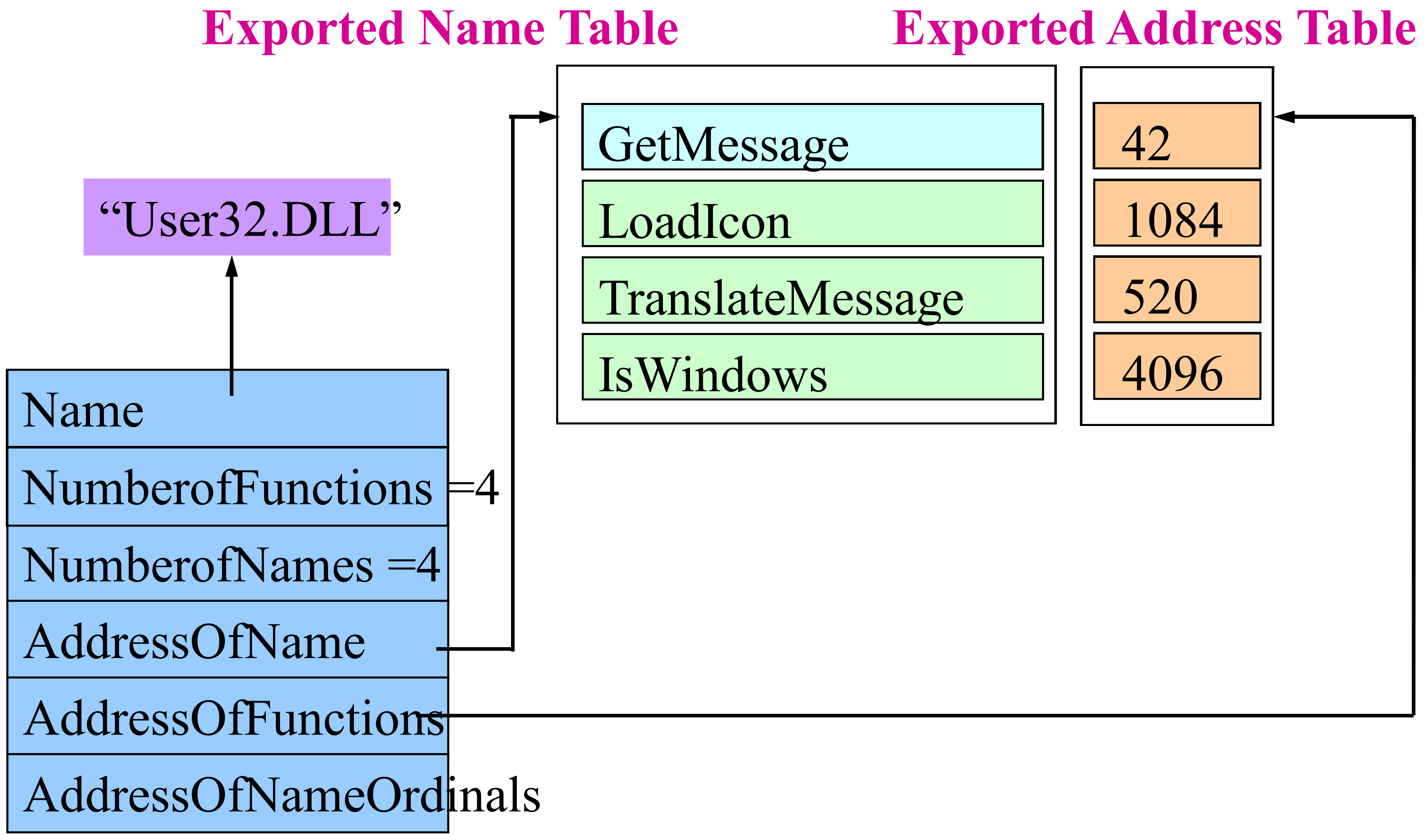
thiscall

- It is used by C++ member functions that do not use variable arguments.
- The callee cleans up the stack;
- **this** is the first argument, stored in **ECX**;
- Function name is decorated by the C++ compiler in an extraordinarily complicated mechanism that encodes the types of each of the parameters, among other things. This is necessary because C++ permits function overloading, ensuring the various overloads have distinct names.

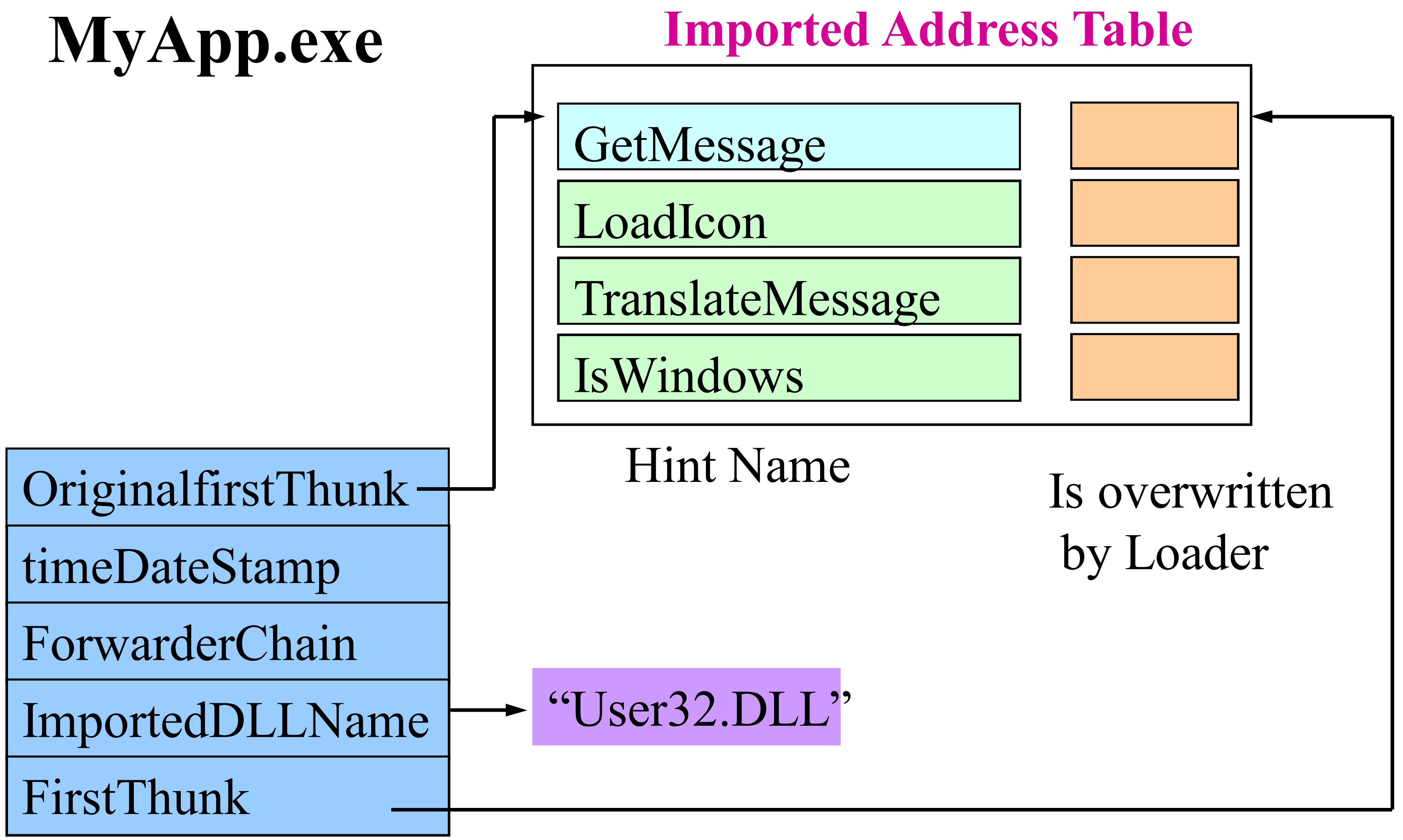
executable file format



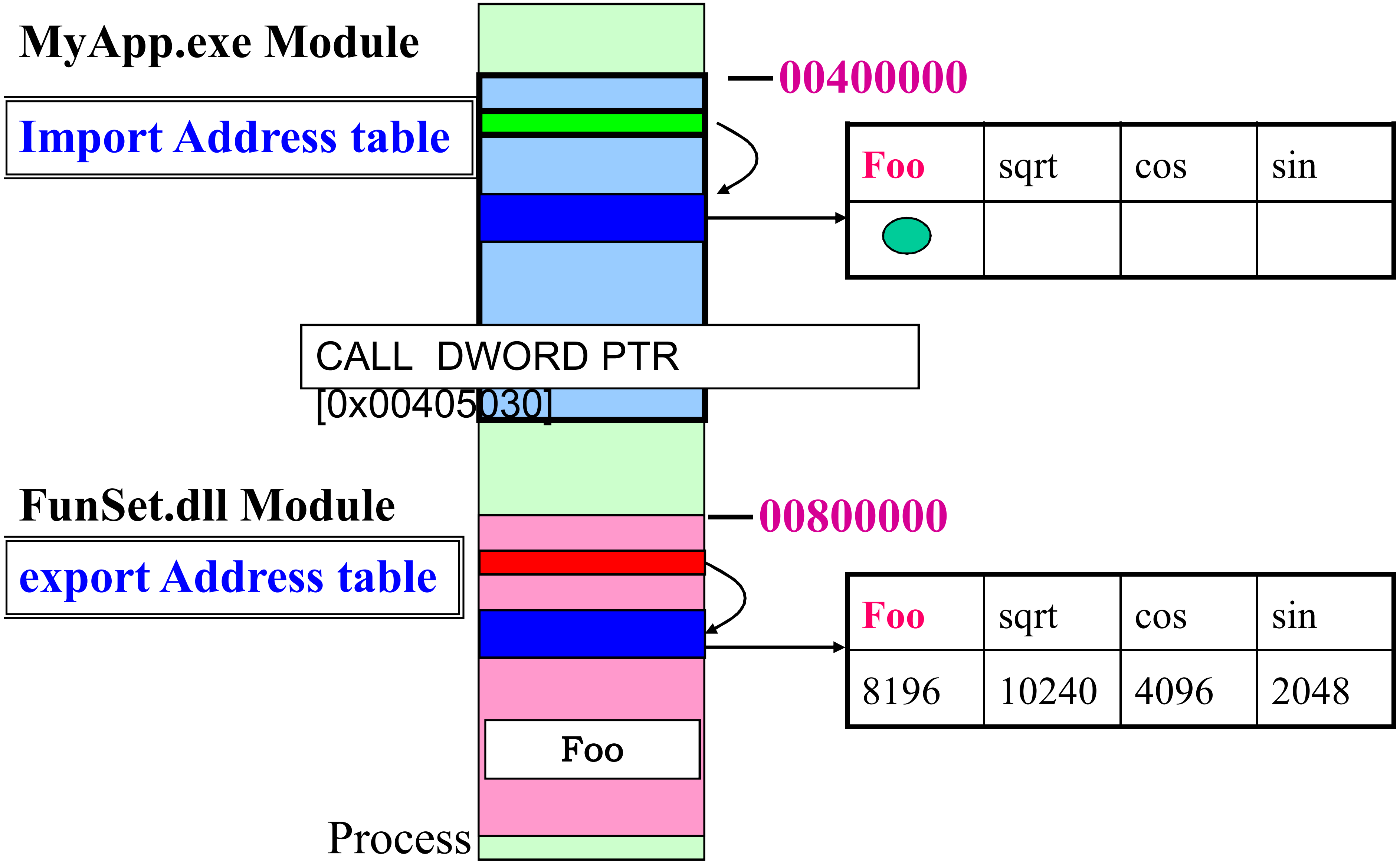
Export Address table



import Address table



Loader- execute a program



Foo	sqrt	cos	sin
●			

Foo	sqrt	cos	sin
8196	10240	4096	2048

超越微软

```
401000:   PUSH    EBP
401001:   MOV     EBP,  ESP
401003:   SUB     ESP, 00000088
401009:   PUSH    EDI
40100A:   MOV     DWORD PTR [EBP- 00000084],  00406030
401014:   MOV     DWORD PTR [EBP- 00000088],  00000002
401036:   MOV     DWORD PTR [004088E8], 12345678
4010E1:   MOV     EAX, DWORD PTR [EBP - 00000088]; PUSH EAX
4010F9:   CALL    DWORD PTR [0x00405030]
4010FE:   ADD     ESP, 4
```

超越的意义何在?

```
401000:   PUSH    EBP
401001:   MOV     EBP,  ESP
401003:   SUB     ESP, 00000088
401009:   PUSH    EDI
40100A:   MOV     DWORD PTR [EBP- 00000084],  EDX +6030
401014:   MOV     DWORD PTR [EBP- 00000088],  00000002
401036:   MOV     DWORD PTR [EDX + 78E8], 12345678
          PUSH    EDX
          MOV     ECX, EDX
4010E1:   MOV     EAX, DWORD PTR [EBP - 00000088]; PUSH EAX
          MOV     EDX, DWORD PTR [ECX + 4096]
4010F9:   CALL    DWORD PTR [ECX + 4030]
4010FE:   ADD     ESP, 4
          POP     EDX
```

同一进程内模块之间的互操作

两个二进制模块在一个进程中来进行互操作：
互操作形式：**函数调用；** 互操作契约(共识): **函数调用规范；**
契约载体：**.h 文件；**

互操作中的2个问题：1）**参数传递；2）代码和数据定位；**

1) 参数传输： 媒介，长度，顺序；
2) 返回值：媒介，和长度;
3) 谁负责清除栈中传递的参数;
4) 函数命名规则;
5) 寄存器的保存与恢复;

参数传递媒介：stack, register

通过名字定位;
策略：导出地址表(callee);
导入地址表(caller);
实现：Compiling;
Linking;
Loading;

互操作契约

Vendor:

在服务方的头文件(support.h)中：

```
int  __declspec(dllexport) __cdecl MyFunc( char *p, double f );
```

6要素

Client:

提供给客户的头文件(support.h)中：

```
int  __declspec(dllimport) __cdecl MyFunc( char *p, double f );
```

Static linking

In the static linking, the imported code and data will be copied into the executable as part of the overall client application from the DLLs. So after linking, the executable is no longer dependent on the DLLs.

Pitfalls:

1) the executable has much larger file size.

2) once the library updates, client application must be compiled and linked. the modularity of components is lost.

同一机器内进程之间的函数调用

1）要有存根（Stub 和 proxy);

2) 线程间的同步机制;

3) 要有侦听线程;

4) 函数调用不仅要传递参数，而且还要加上函数标识信息(DLL名，函数名）；

5) 要有进程间数据传递管线（pipeline)（IPC），及传递数据的序列化（Serialization）；

6) 有关上述2）、4）、5）项的契约（Contract）；

客户方要有代理（proxy）

```
int _MyFunc_proxy( char c, int i, double f )   {
    Pipeline = ExecutableEnvirnment. GetPipeline(“interoperation”);
    Pipeline.AddDllName(“MyModule”);
    Pipeline.AddFunctionName(“MyFunc_stub”);
    Pipeline.AddCharParameter(c );
    Pipeline.AddIntParameter(i );
    Pipeline.AddDoubleParameter(f );
    TriggerEvent(hCallEvent);
   WaitforSingleObject(hReturnEvent);
   int ret = Pipeline.FetchIntValue( );
   Pipeline.Close( );
   Return ret;
}
```

作用：
1) 代理；
2) 参数的序列化；

进程间的同步

Pipeline由运行环境提供

服务方要有存根（stub）

```
int myFunc_stub ( unsign char *pParameters )   {
    char c;
     int i;
    double f;
    c = AbstractChar[pParameter,0,0);
    i = AbstractInt(pParameter,1,4) ;
    f = AbstractDouble(pParameter,5, 13) ;
    int ret = MyFunc(c, i, f) ;
    Return ret;
}
```

作用：
1) 代理；
2) 参数的解析；

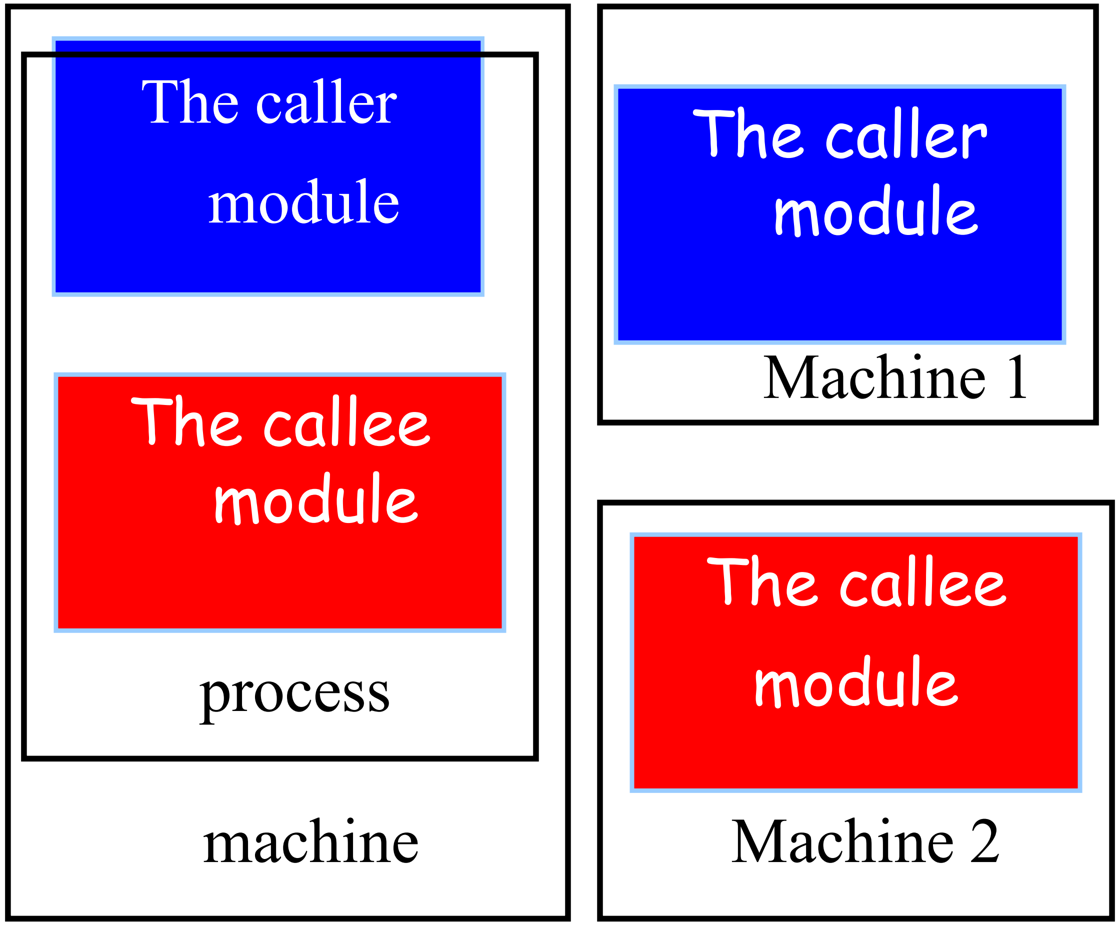
服务方守护进程(Damon)的实现

```
void _ServiceThread(    )   {
   unsigned char szDllName[128], szFunName[128], pBuffer[4096]
   Bool   (*pfn)( unsigned char *) = 0;
   Pipeline = CreatePipeline("interoperation");
   while (1)   {
          WaitforSingleObject(hCallEvent);
          Pipeline.FetchDllName(szDllName);
          Pipeline. FetchFunctionName(szFunName);
          Pipeline. FetchParameters(pBuffer);
           HINSTANCE h = LoadLibrary(szDllName);
           if (h)  *(FARPROC* )&pfn = GetProcAddress(h, szFunName);
           If (pfn)  int result = pfn(pBuffer);
           Pipeline.AddIntValue(result );
           TriggerEvent(hReturnEvent);
   }   //while
}
```

Function Virtualization

4 situations: same module, same process, same machine; different machine;

```
int main( int argc, char *argv[ ] )
  {
   char szBuf [128];
   char *psz = "Hello";
   unsigned long   p = 2;
    g_myGlobalVariable = 0x12;
//Procedure invocation :
  MyFunction( p );
}

int MyFunction (int anArg) {
    int aVariable;
    aVariable = anArg;
    return aVariable;
}
```



客户方的的自动适配

当服务方在本进程上时，直接调用函数，否则调用代理函数；

How to implement?

Please you answer.

不同机器之间的互操作

和进程之间的互操作一样；只不过Pipeline的实现不是通过共享内存，而是通过网络socket;

其中的核心问题：

1) Identifier：IP ＋Port ＋ DLL＋ Procedure;

2) Data encoding (Data Representation); NDR

3)Network Transportation Protocol；

DCE RPC

Compontents:

- ✓ MIDL compiler;
- ✓ Run-time libraries and header files;
- ✓ Name service provider (or the Locator);
- ✓ Endpoint mapper (or the port mapper);

•**Transportation protocol:** TCP/IP ,UDP/IP , named pipes transport, LPC , HTTP (IIS);

•**Data encoding**: NDR.

章节小结－互操作的要点

- 客户和服务方之间的统一共识通过.h(头）文件来达成；
- 在同一进程中的互操作，通过编译器，链接器，加载器来完成；
- 在不同进程之间的互操作，通过开发工具产生**存根**和**支持环境**来完成；
- 互操作并没有给应用程序员带来额外负担，额外的工作由**开发工具**自动完成；

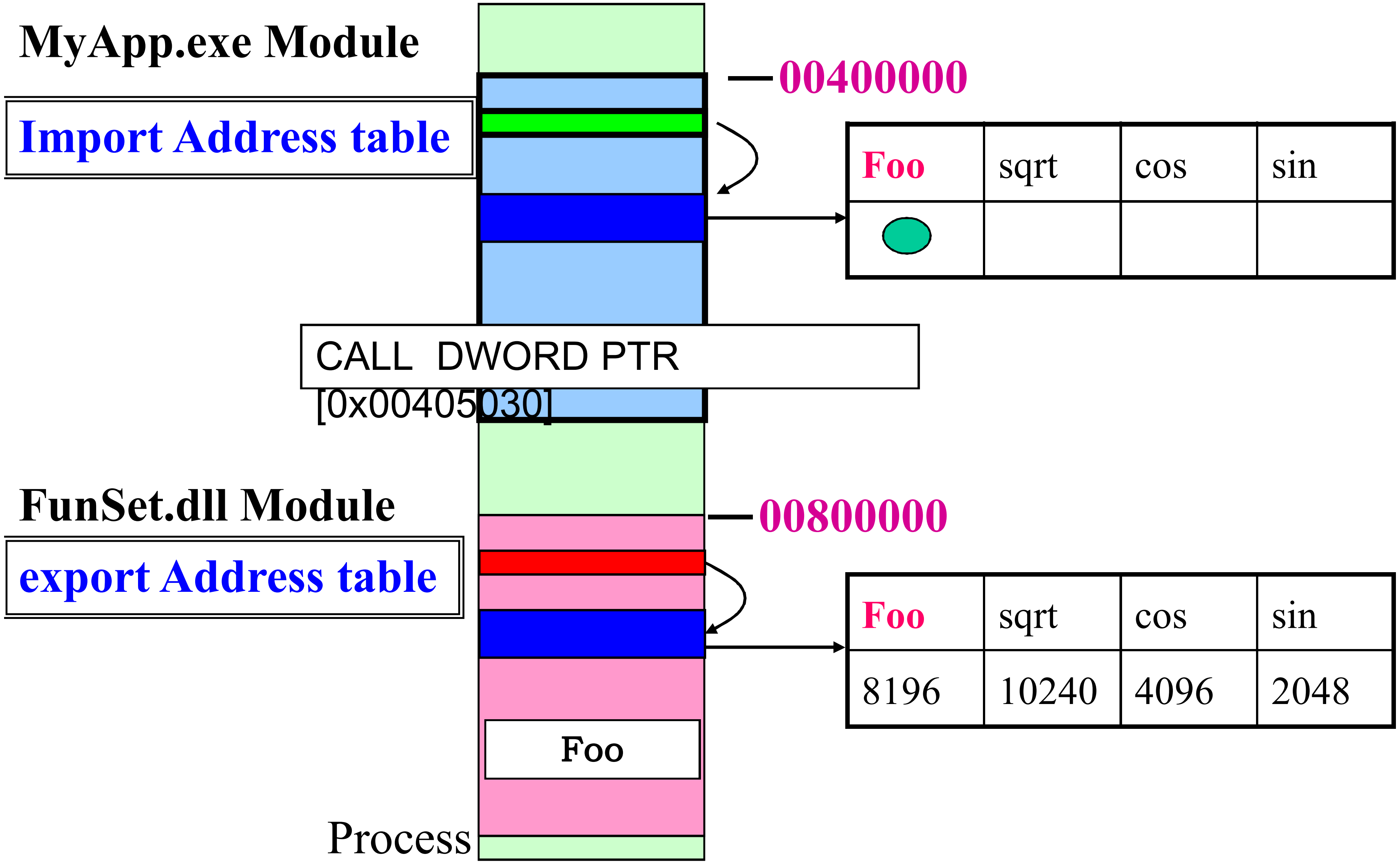
作业

- 网上查阅DCE RPC资料，然后用visual C++设计和编程实现不同机器间的互操作,主要工作有:
- 写出一个运行环境支持库，一个damon看护进程（exe文件，一个DLL）；
- 针对一个函数:

```
Int MyFunc(int p1, char *p2);
```

- 写出它的**proxy and stub**代码;
- 写一个客户和服务程序，就此函数实现互操作。

问题



MyApp.exe Module
Import Address table
00400000
Foo
sqrt
cos
sin
CALL DWORD PTR [0x00405030]
FunSet.dll Module
export Address table
00800000
Foo
sqrt
cos
sin
8196
10240
4096
2048
Foo
Process

问题1

```
•int main( int argc, char *argv[] )     {
•401000:   PUSH      EBP
•401001:   MOV       EBP,  ESP
•401003:   SUB       ESP, 00000088
•401009:   PUSH      EDI
•40100A:   MOV    DWORD PTR [EBP- 00000084],  00406030
•401014:   MOV    DWORD PTR [EBP- 00000088],  00000002
•401036:   MOV    DWORD PTR [004088E8], 12345678
•4010E1:   MOV    EAX, DWORD PTR [EBP - 00000088]
•4010F9:   CALL   DWORD PTR [0x00405030]
•4010FE:   ADD    ESP, 4    }
```

➢为什么要导入表？既然事先假定了一个模块在进程中的加载地址，为什么不事先填上导入表？还要在加载时来填入？

➢既然存在加载地址冲突问题，为什么还要建立导入表？为什么不把CALL DWORD PTR [0x00405030] 直接改成 CALL　808196?

答：存在一个分散与集中的问题，当加载地址冲突时， CALL　808196方案要修改多个页，而DWORD PTR [0x00405030]方案只需要修改一个页；

问题2

```
•int main( int argc, char *argv[] )     {
•401000:    PUSH       EBP
•401001:    MOV        EBP,  ESP
•401003:    SUB        ESP, 00000088
•401009:    PUSH       EDI
•40100A:   MOV     DWORD PTR [EBP- 00000084],  00406030
•401014:    MOV    DWORD PTR [EBP- 00000088],  00000002
•401036:    MOV    DWORD PTR [004088E8], 12345678
•4010E1:    MOV    EAX, DWORD PTR [EBP - 00000088]
•4010F9:    CALL    DWORD PTR [0x00405030]
•4010FE:    ADD    ESP, 4    }
```

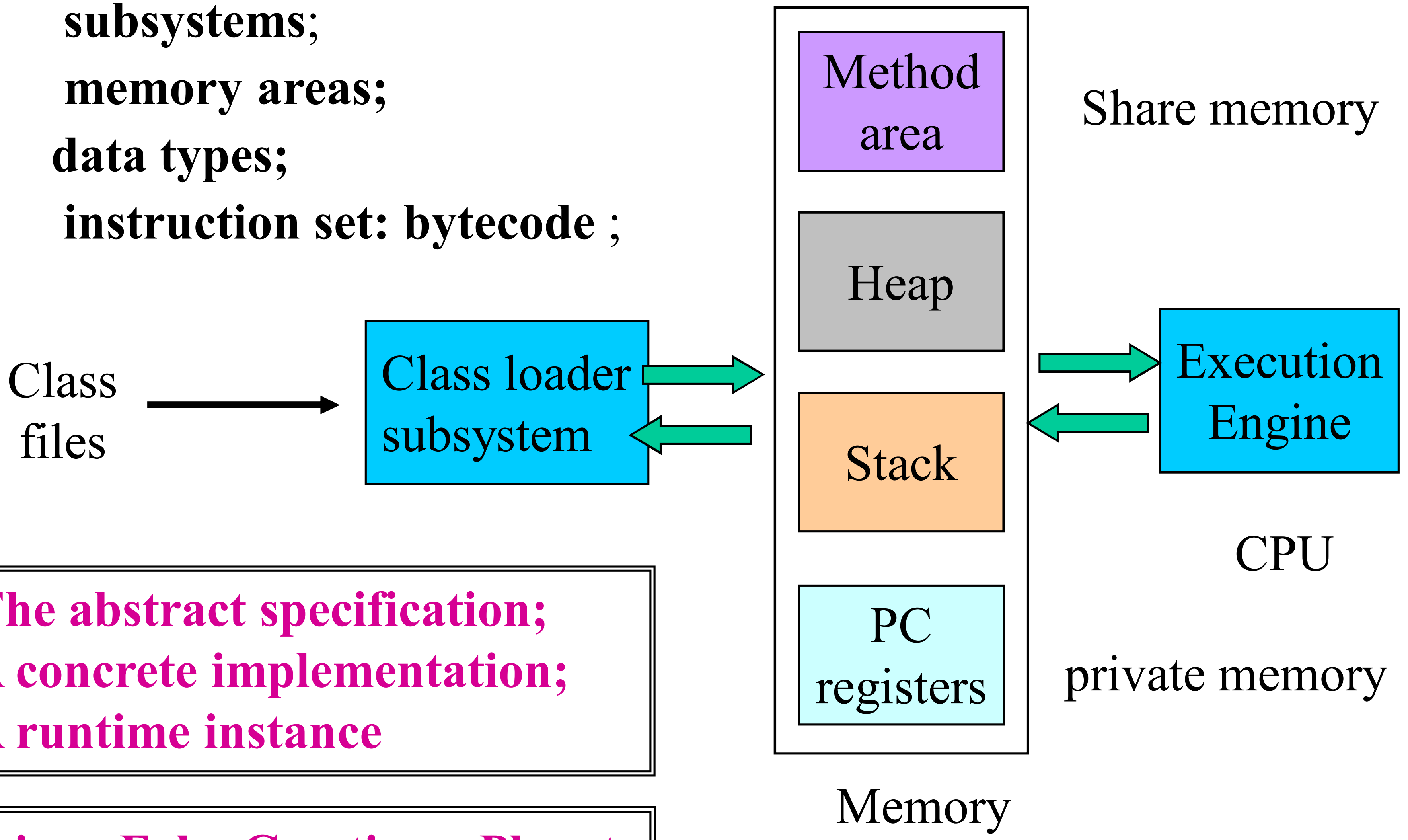
➢实现代码的位置无关性（code independent on position), 需要引入对象；对象的位置信息，使用者要知道。即我们提出的方案，也是提出Java的关键出发点；

补充学习资料

- **Introduction to Java and .Net technology;**

Java virtual machine

- **subsystems;**
- **memory areas;**
- **data types;**
- **instruction set: bytecode ;**



- **The abstract specification;**
- **A concrete implementation;**
- **A runtime instance**

C > java Echo Greetings, Planet

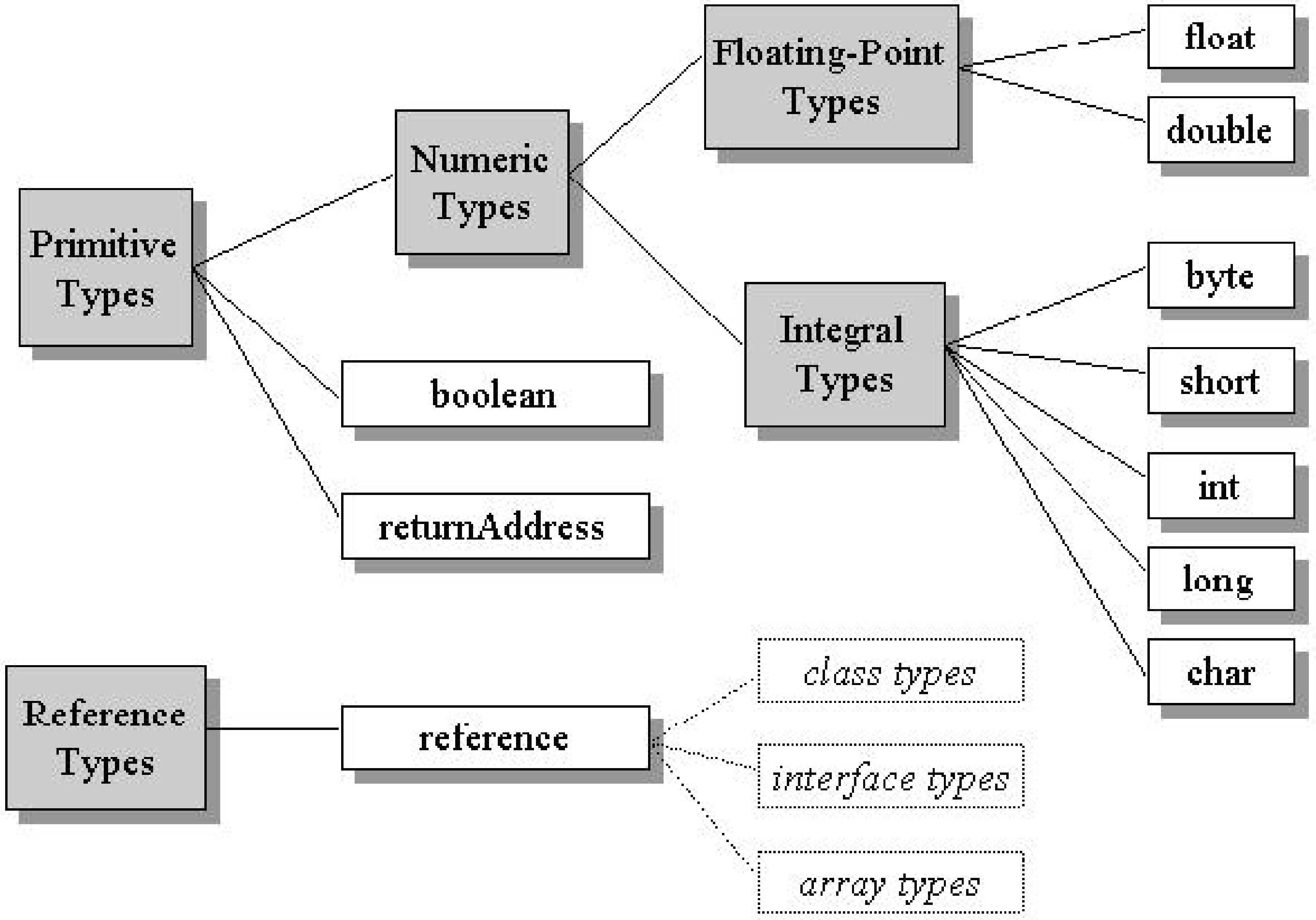
Where is the class files?

The starting point for a Java application

```
class Echo {
    public static void  main(String[] args) {
        int len = args.length;
        for (int i = 0; i < len; ++i) {
            System.out.print(args[i] + " ");
        }
        System.out.println();
    }
}
```

```
C > java Echo Greetings, Planet
```

Java Data Types



Java class file format

Protocol:

Java binary files**".class"**:

conform to the Java class file format.

Class information

- Name;
- It’s direct superclass
- Type: a class or an interface
- The type's modifiers : such as public, abstract, final;
- An ordered list of the direct superinterfaces;

- Field information;
- Method information;
- All class (static) declared variables ;
- A reference to class ClassLoader;

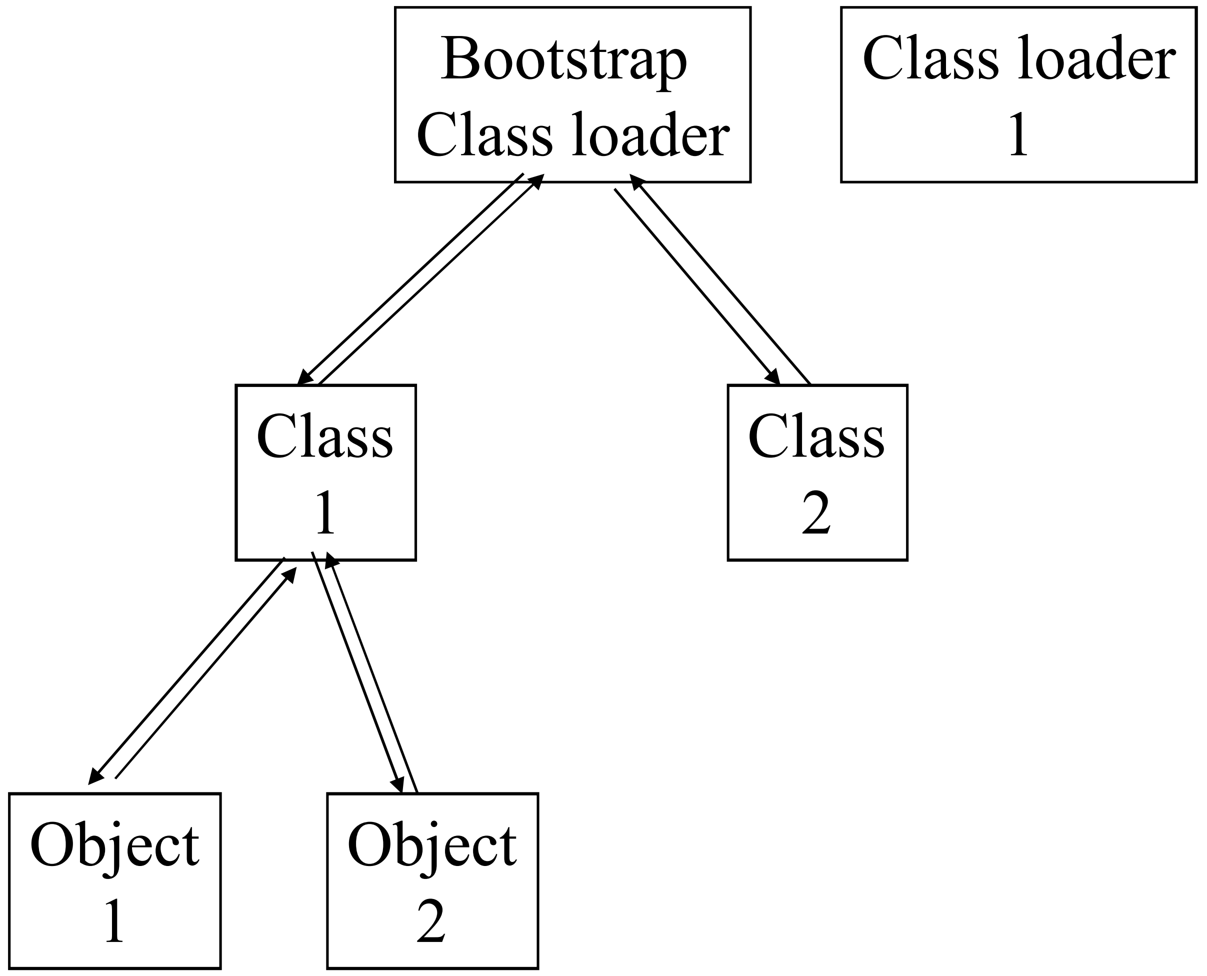
Field Information

- name;
- type;
- modifiers , such as public, private, protected, static, final, volatile, transient;

Method Information

- name;
- Signature:
  - return type (or void);
  - The number, types , order of the method's parameters ;
- The method's modifiers , such as public, private, protected, static, final, synchronized, native, abstract;
- The method's bytecodes;
- The sizes of the operand stack and local variables sections of the method's stack frame;
- An exception table;

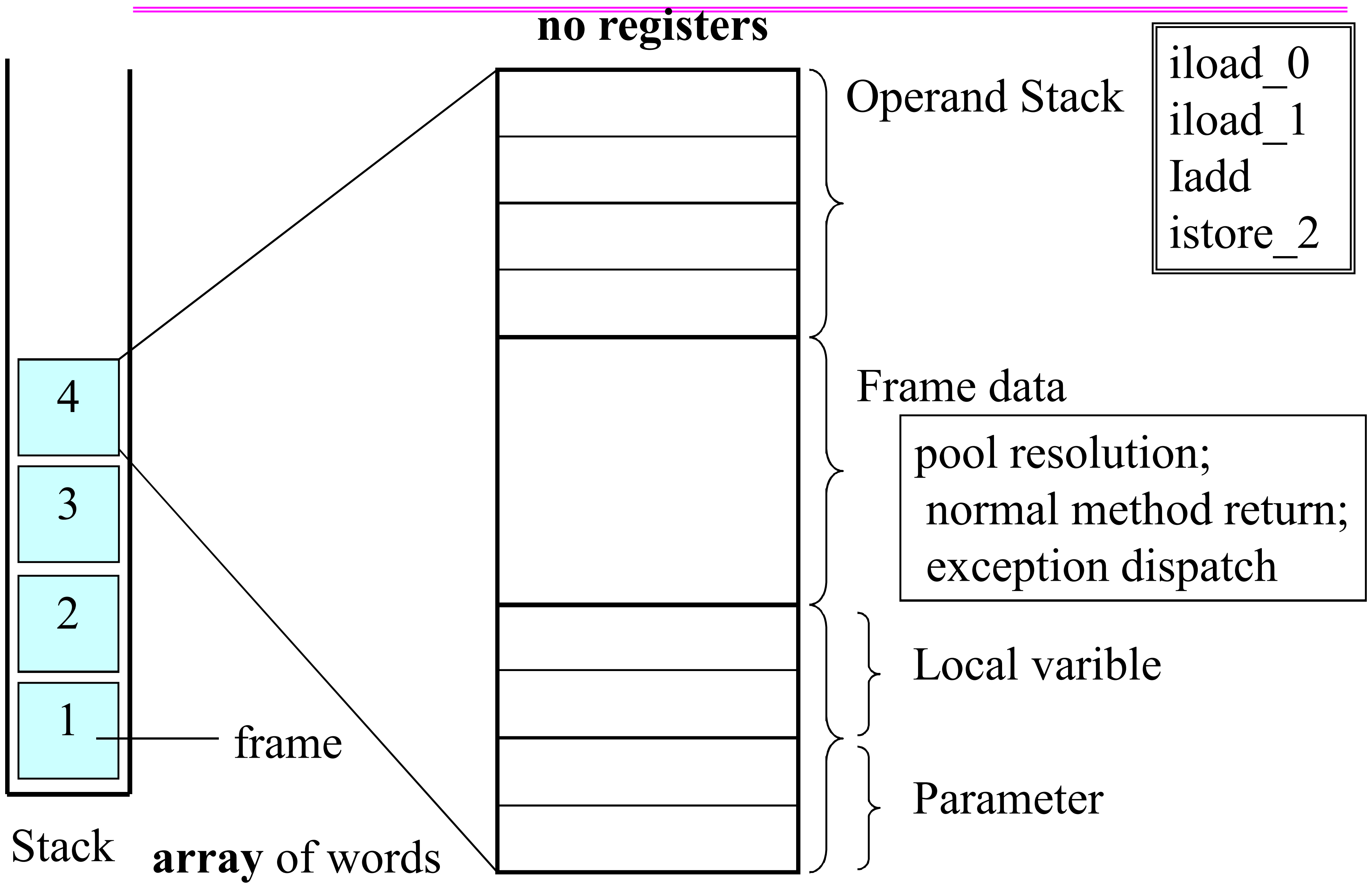
Hierarchical organization structure



Loader

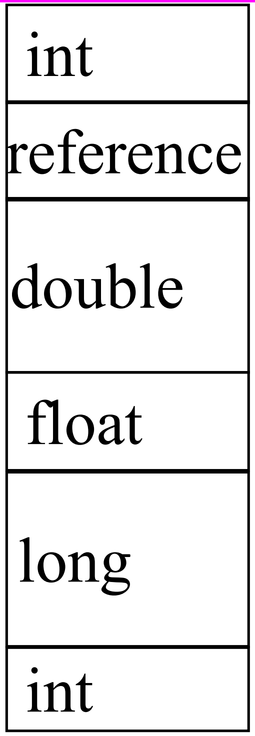
- **Loading**: finding and importing the binary data for a type
- **Linking**:
  - **Verification**: ensuring the correctness of the imported type
  - **Preparation**: allocating memory for class variables and initializing the memory to default values
  - **Resolution**: **transforming symbolic references from the type into direct references**.
- **Initialization**: invoking Java code that initializes class variables to their proper starting values.

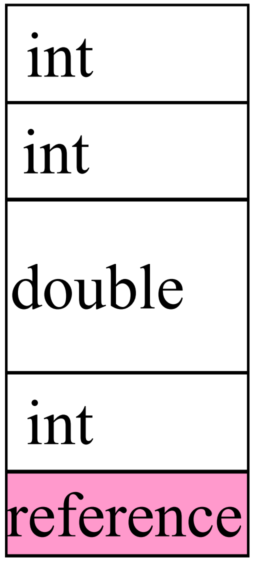
Java stack frame



函数调用协议

```
class Example3a    {
   int  cp;
   public static int runClassMethod(int i, long l,
   float f, double d, Object o, byte b) {
         return 0;
   }
   public int runInstanceMethod(char c, double d,
   short s, boolean b) {
         return 0;
   }
}
```





从左到右的顺序压栈

a sequence of instructions

- *opcode* **followed by zero or more** *operands*

```
class Act {
  public static void doMathForever() {
      int i = 0;
      for (;;) {
          i += 1;
          i *= 2;
      }
  }
}
```

```
0    iconst_0        // 03
1    istore_0        // 3b
2    iinc 0, 1       // 84 00 01
5    iload_0         // 1a
6    iconst_2        // 05
7    imul            // 68
8    istore_0        // 3b
9    goto 2          // a7 ff f9
```

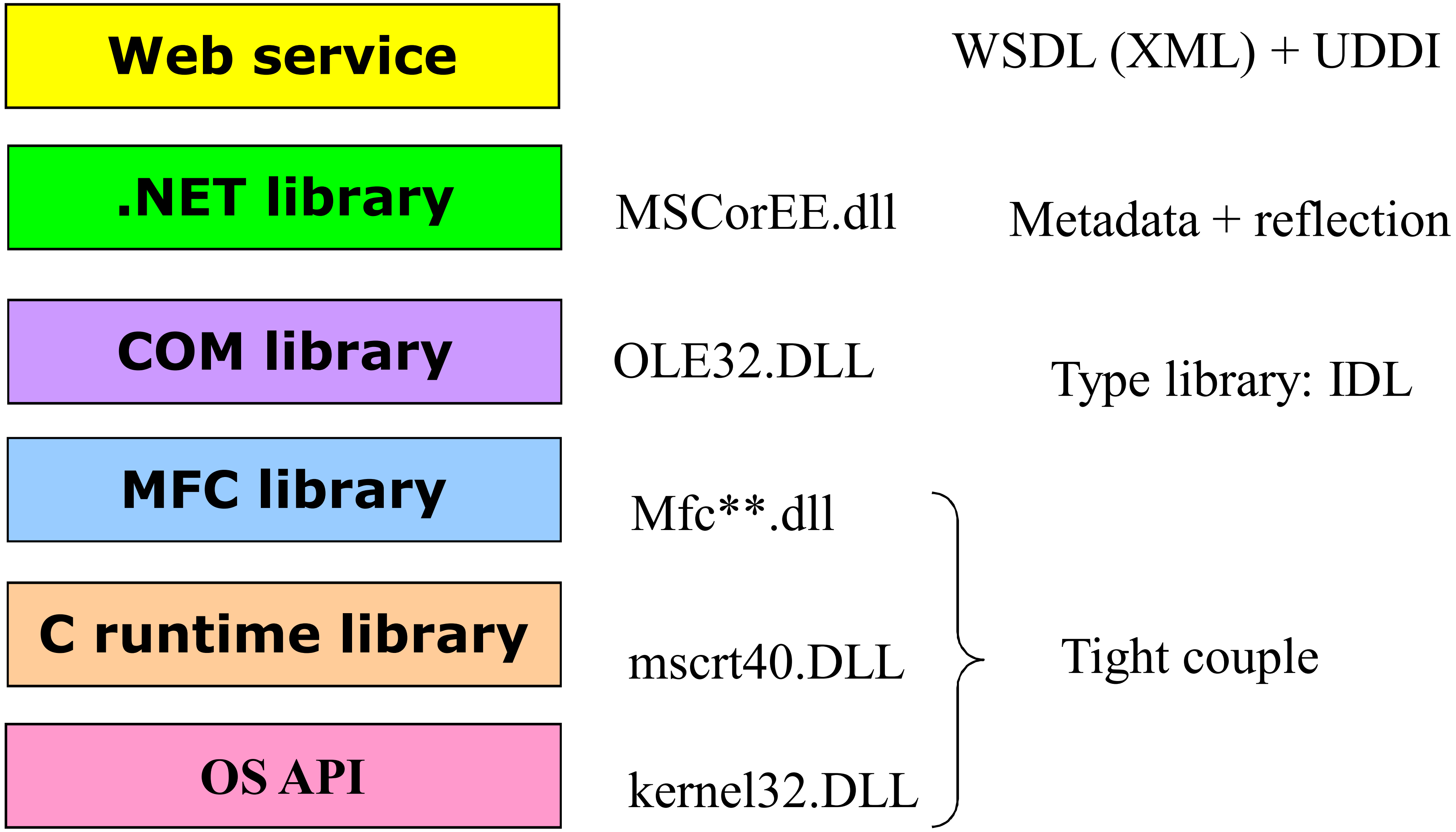
Benefits from JVM and .NET

- 实时自适应性，使用元数据，使得不同函数调用规程的代码可相互调用；
- 优化有两点：
  - 1）减少代码；
  - 2）利用高速存储器；
- 变间接为直接：减少了代码：提高了效率；例如函数inline，省去了参数传递的代码，动态函数内联必须要实时编译；例如间接的函数调用变成了直接的函数调用；
- 存储器的访问速度不相同，register，memory；数据被访问的频率不相同；高访问频率的数据应该放到高速度的存储器中。提高了效率；
- 机器不相同，例如寄存器的数量不相同，原有的编译器只能保守地使用寄存器；VM能够发现机器的特征，尽量地利用机器特征；
- 代码被执行的频率不相同，对于高频率执行的代码应该多做一些上述优化；

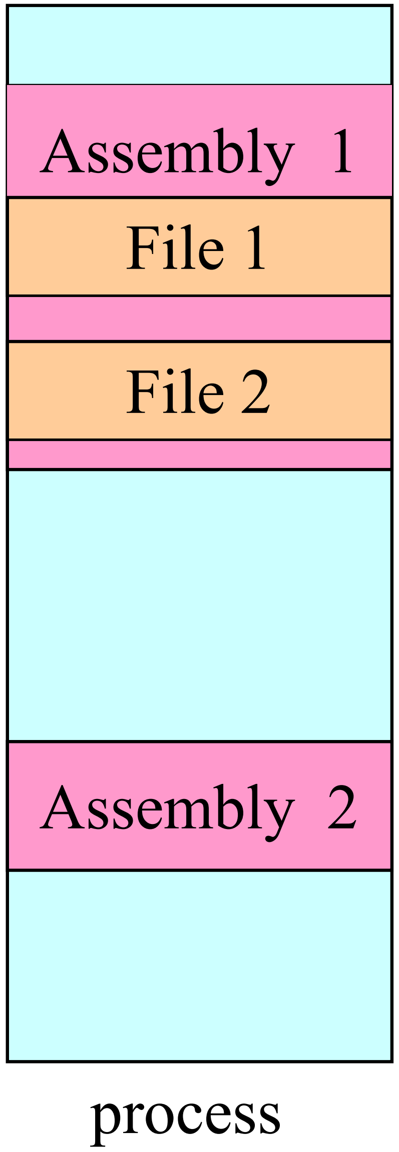
DLL Hell

- Since installing components for a new application can overwrite components of an old application, the old app can exhibit strange behavior or stop functioning altogether.

Microsoft trace



.NET executable



Process;

- **MSCorEE.dll;**
  - **Assembly** :security, version and **reference** management unit;
    - **file;**

the context of an assembly

Assembly:

- incremental and on-demand download;
- a unit of versioning and deployment;
- **the global assembly cache;**
- **the transient assembly cache;**

Execution mode of IL code

- 程序特性：
- 生产compiler的厂家屈指可数，它不会要调用和访问很多很多别人生产的组件；它是可以和机器、操作系统紧密结合的；
- 应用程序的生产厂家却成千上万，它的功能五花八门，必须用组合技术来集成别人的组件，另外，它最终运行环境也无法事先确定；
- Compile each MSIL instruction with its native code counterpart;
- interpreted code;
- Performance is a problem in such a pattern;

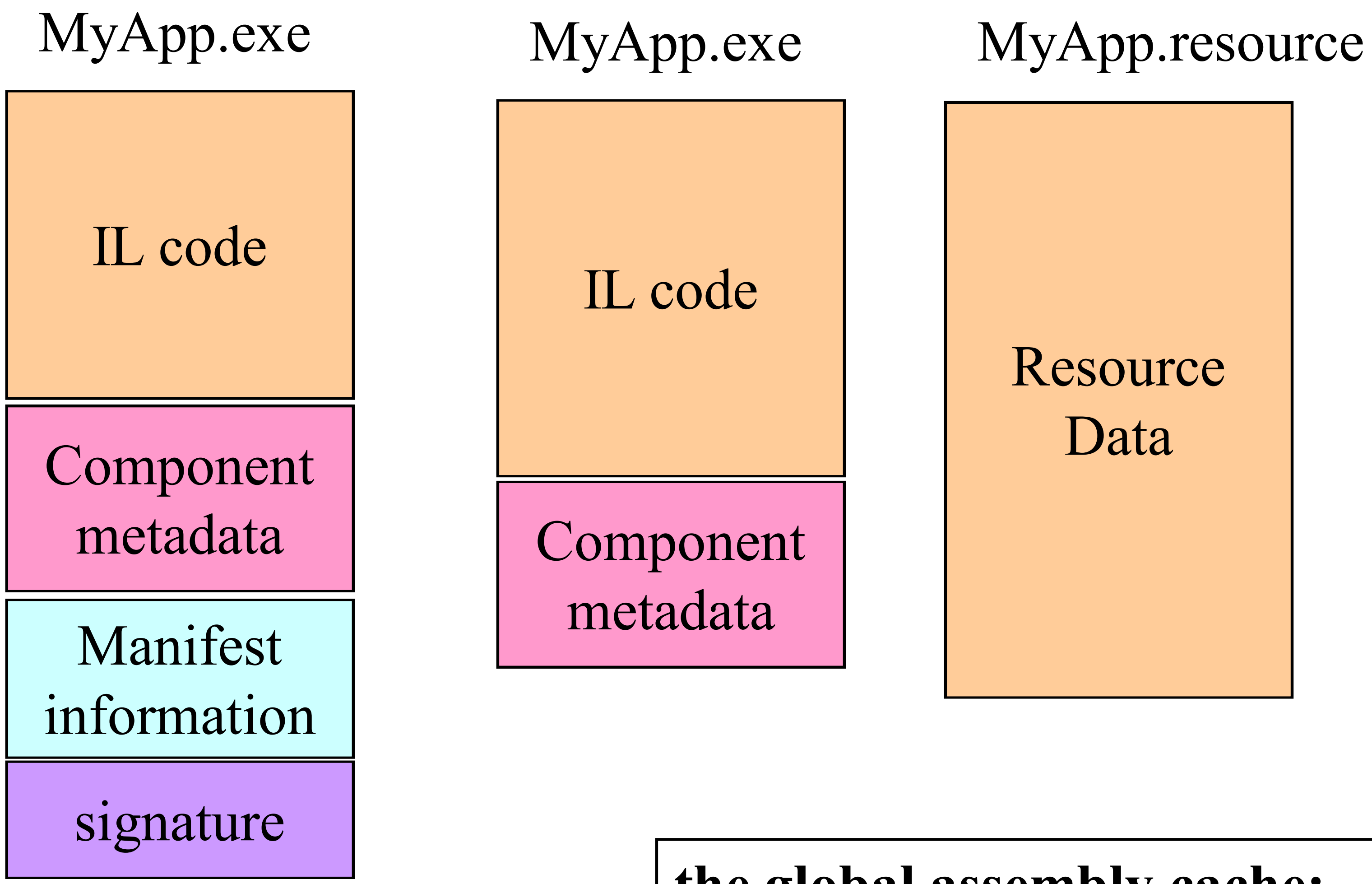
.NET executable

The primary purpose of a .NET executable is to get the .NET-specific information such as **metadata** and **intermediate language** (IL) into memory.

Its **entry point** is usually a tiny **stub** of code. That stub just jumps to an exported function in **MSCORLIB.DLL** (**_CorExeMain** or **_CorDllMain**). From there, MSCOREE takes charge, and starts using the **metadata** and **IL** from the executable file.

.NET information is defined in **CorHDR.H** and **WINNT.H**.

Sections of a assembly file



the global assembly cache;

the transient assembly cache;

Assembly identity

A textual simple name: the file name hold the assembly manifest;

A compatibility version number : incompat.compat.hotfix;

A cultural locale ;

A cryptographic public-key ;

Assembly manifest info

assembly's identity:

name:myapp; version:1.0.0; locale:English;

Public-key:xxx;

Files that make up the assembly:

Myapp.dll hash value;

My app.resource hash value;

Assembly exports:

……….;

Assembly import (the assemblies that the assembly depends on)

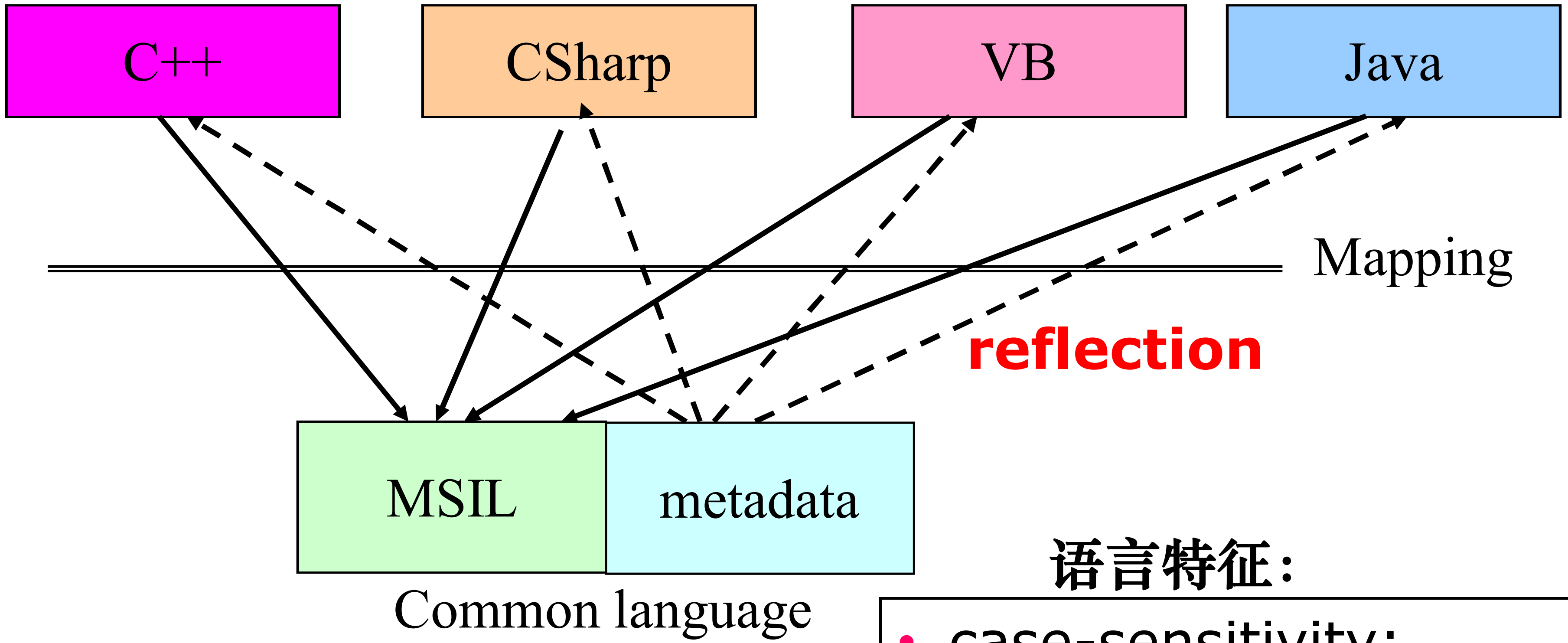
name: mscorlib; version:1.0.0; locale:English;

Public-key:xxx;

name: system.net; version:1.0.3; locale:English;

Public-key:xxx;

.NET



语言特征：

- case-sensitivity;
- unsigned integers;
- operator overloading;
- unions;
- 函数调用中参数个数可变;
- Multiple inheritance;

➢**standard set of types;**
➢**self-describing type information;**
➢**common execution environment;**

.NET virtual machine

The common language runtime use **metadata** to:

- ✓locate and load class types in the file;
- ✓lay out object instances in memory;
- ✓resolve method invocations and field references;
- ✓translate MSIL to native code;

- STACK;
- OPERATION;
- NEWOBJ;
- CALL;
- Hardware Abstraction layer: 线性设备,平面设备,多维设备;

Just-in-time compiling

- should file be as the basic compile unit?
  - convert all of the file's MSIL code to CPU instructions at load time?
  - very rarely does a user cause an application to execute all of its code.
- functions are the basic compile unit?
- Non-branch IL code block are the basic compile unit?
-
- PreJit.exe ,when you turn it on, it compile an entire assembly to native code and save the result on disk.

Just-in-time compiling (cont.)

- When a class type is loaded, it connects stub code to each method. When a method is called, the stub code directs program execution to the component of the common language runtime engine that is responsible for compiling the method's MSIL into native code.

- Once the JIT compiler has compiled the MSIL, the method's stub is replaced with the address of the compiled code. Whenever this method is called in the future, the native code will just execute and the JIT compiler will not have to be involved in the process.

The base framework assembly

Hundreds of classes in the base library, the library is divided into namespaces that group related classes together.

System namespace:

exception handling;

garbage collection;

console I/O

Types conversion;

data formatting;

random number generation;

various math functions;

Namespace
System
System.Collections
System.Data
System.Globalization
System.IO
System.Net
System.Reflection
System.Runtime.Remoting
System.Security
System.Web.UI.WebControls
System.WinForms

The base framework assembly

System.Object type:

- allows two objects to be compared for equality,
- uniquely identify an object via a hash code,
- query the object's true class type;
- perform a shallow (bitwise) copy of the object;
- obtain a string representation of the object's current state;

Backward integration

- managed code calling unmanaged DLL functions;
- managed code instantiating and calling interface methods of COM servers;
- unmanaged code instantiating and calling methods on .NET servers;

.NET calling functions

the name of the function; (such as MessageBoxA)

the name of DLL file containing the function (such as User32.dll);

How to marshal the function's parameters;

.NET calling COM

Use TlbImp.exe utility parses a COM type library and produces a wrapper (a managed DLL) whose metadata describes the methods that wrap the COM server's interface methods;

This wrapper acts as a proxy between the managed and unmanaged code, handling all administrative tasks such as **marshaling**, **AddRef**, **Release**, and so on.

COM calling .NET

the common language runtime generates a COM-callable wrapper as **a proxy**, handling all administrative tasks such as marshaling, AddRef, Release, and so on.

The register assembly utility (RegAsm.exe), generates a GUID for the .NET server and **updates the registry**;

Use TlbExp.exe utility to generate **a type library**;

.NET Type Safety

Type-safe programs reference only memory that has been allocated for their use and access objects only through their exposed interfaces.

combining strong typing in the metadata (parameters, members and array elements, return values of methods, and static values) with strong typing in the MSIL (local variables and stack slots).

Evidence can include information such as which Internet zone and site the code originated from, its shared name, and its publisher's identity.

When a method demands access, the common language runtime walks up the call stack and checks to see if all assemblies in the call chain have the required permission.