计算机程序设计语言

子程序的概念

张 天

软件工程组

计算机科学与技术系

南京大学

本章要点

- 本章主要讨论编程语言中的子程序的概念，涉及如下一些具体内容：
  - 程序和子程序的运行模型
    - 了解程序和子程序的运行模型，类似于从计算机体系结构的角度理解软件
  - 子程序的参数、局部变量
  - 系统子程序

子程序

- 子程序是**过程抽象**的表现形式
  - 在现代编程语言中，对过程的**抽象**和**复用**主要使用子程序来表示
  - 这样，可以在程序中使用一个调用子程序的语句，来代替一组计算的具体描述
  - 子程序是编程语言设计领域中最重要的概念之一

参数传递

- 子程序与其他部分之间的数据交换通过两种方式
  - 直接访问非本地变化（都可见的变量）
  - 参数传递
- 形参与实参
  - 形参：在子程序定义的时候，出现在子程序头部的是形式参数，称为形参
  - 形参都是静态的，只有在实际发生子程序调用的时候才会绑定到具体的存储空间上
  - 实参：子程序调用语句必须给出的是一些列与形参对应的实际参数，称为实参

形式参数的对应

- 两种主要的实参到形参的绑定方式
  - 由位置决定具体的对应（**位置参数**）
  - 由参数名来决定具体的对应（**关键字参数**）
- 差别
  - 当参数的**个数较少**时，通过位置参数的方式既安全也可靠，但当个数较多时就比较容易出错
  - 关键字参数要求程序员必须记住关键字，同时程序也会增加更多的代码量

示例

```
sumer(length = my_length,
      list = my_array,
      sum = my_sum)
```

Python关键字参数的调用方式

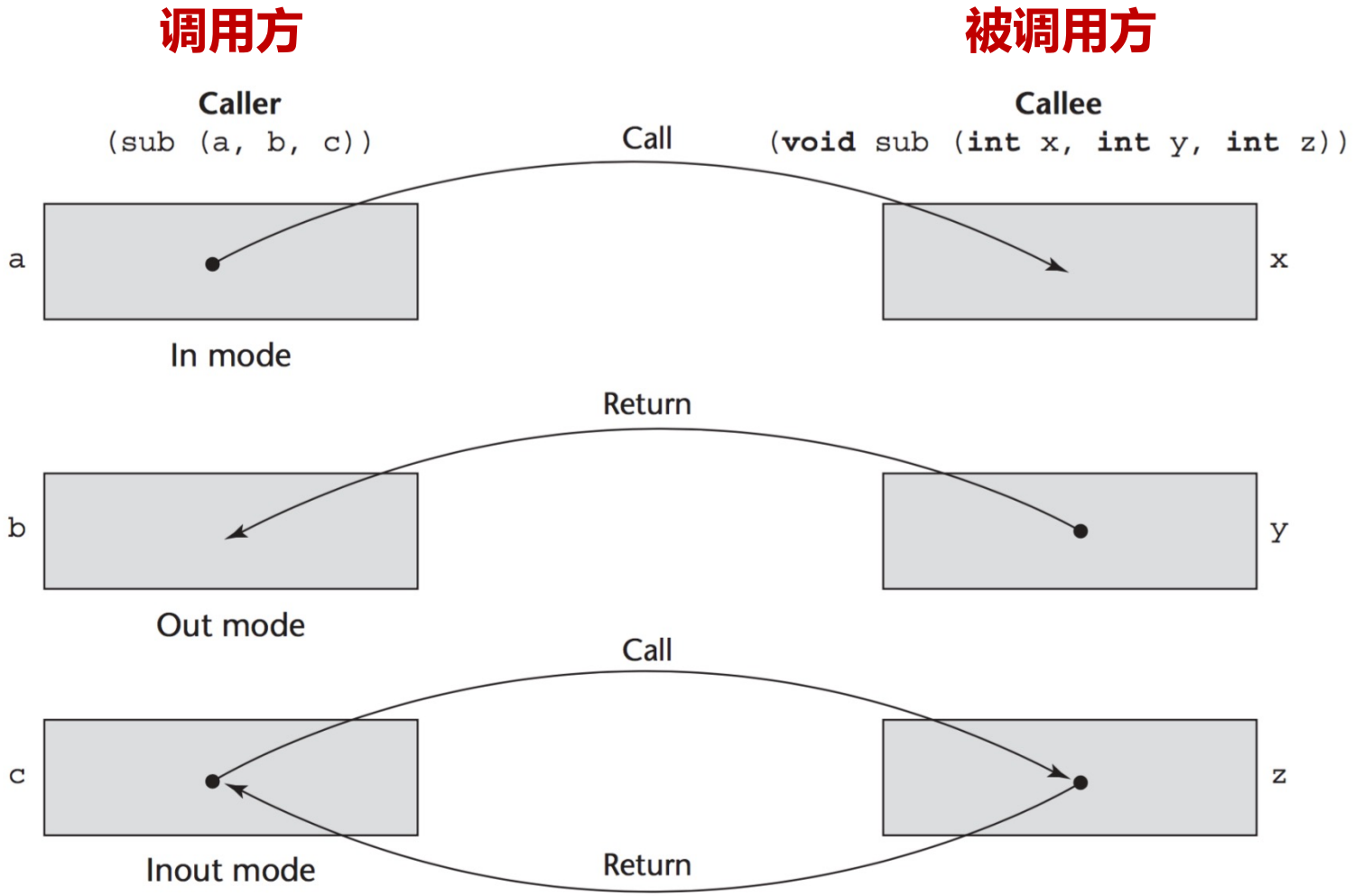
```
sumer(my_length,
      sum = my_sum,
      list = my_array)
```

Python关键字和位置参数在子程序调用中混用

参数传递的语义模型

- 形参的三个不同语义模型
  - 形参**只接收**来自对应实参的数据（输入型）
  - 形参可以将自己的值传回实参（输出型）
  - 以上两者皆可（输入输出型）
- 关于怎样传输，有两种概念模型
  - 复制实际的值
  - 通过访问途径（即地址）

传参的三种语义模型


调用方
被调用方
Caller
(sub (a, b, c))
Call
Callee
(void sub (int x, int y, int z))
a
x
In mode
Return
b
y
Out mode
Call
c
z
Inout mode
Return

参数传递的实现模型

- 在实现各种参数传递时，有如下一些常用的模式
  - 按值传递
  - 按结果传递
  - 按值-结果传递
  - 按引用传递

程序的基本结构

- 程序的运行模型直接决定了子程序的运行模型

- 在介绍子程序的运行模型之前，有必要先简单介绍一下程序的静态和动态基本结构

进程

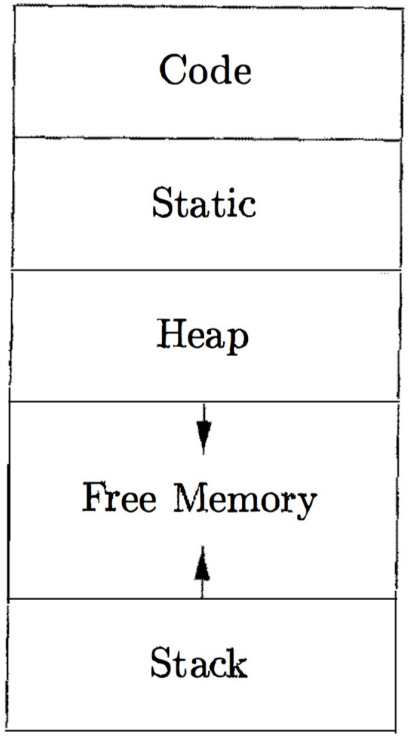
- 进程（process）的经典定义
  - 进程就是一个执行中的程序的实例
- 进程上下文
  - 系统中的每个程序都运行在某个进程的上下文（**context**）中
  - 上下文是由程序正确运行所需的**状态**组成的，包括存储器中的程序的**代码**和**数据**、**程序栈**、**通用目的寄存器的内容**、**程序计数器**、**环境变量**以及**打开文件描述符**的集合

进程的意义

- 进程提供给应用程序的假象：
  - 就好像我们的程序是系统当前运行着的**唯**一的程序
  - 我们程序好像是**独占**地使用处理器和存储器
  - 处理器就好像是**无间断**地一条接一条地执行程序中的指令
  - 程序中的代码和数据好像是系统存储器中**唯**一的对象

内存格局

- 这是一个非常典型的运行时刻内存格局



代码区

静态区

堆区

栈区

注意：事实上，这个内存的格局是倒过来的！

也就是说，实际上栈是向下生长的

代码区

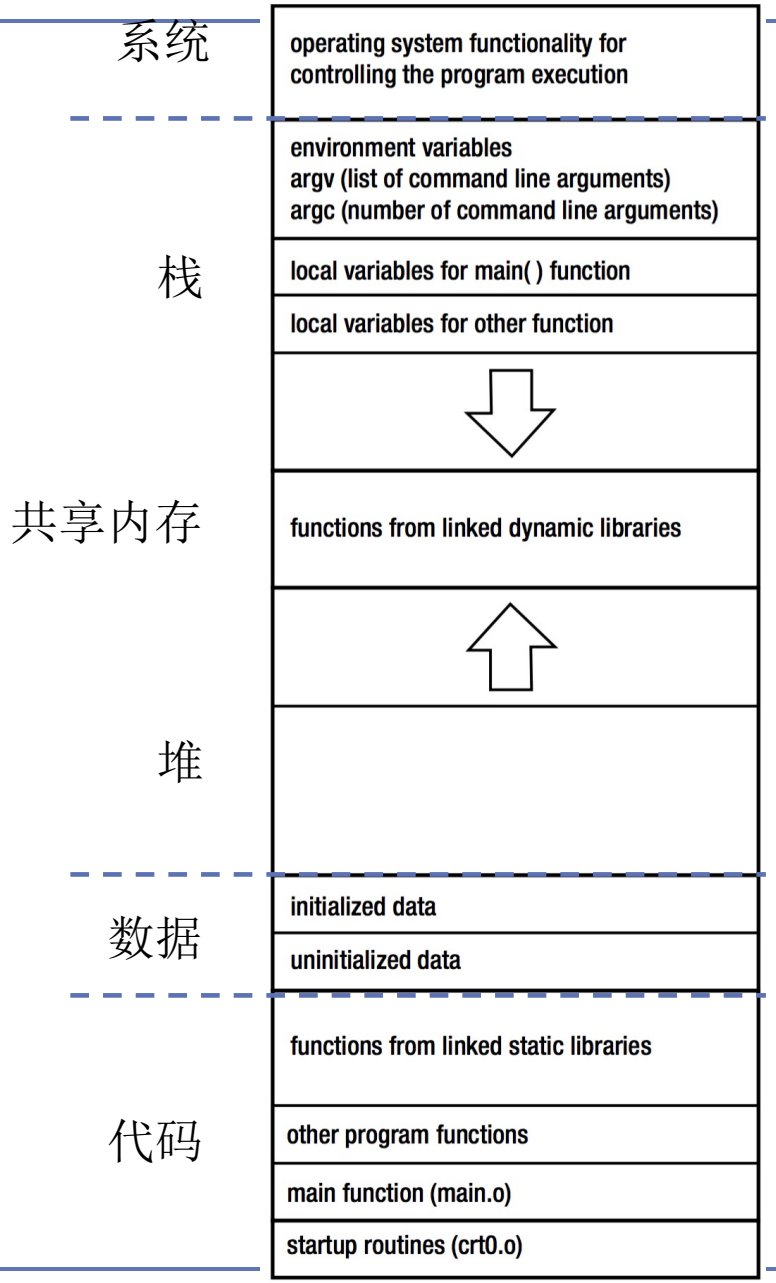
- 生成的目标代码的大小在编译时刻就已经固定下来了，因此编译器可以将可执行目标代码放在一个**静态确定**的区域，即代码区
- 代码区通常位于运行时刻内存的的低端位置

数据区

- 与代码类似，程序的某些数据对象的大小可以在编译时刻知道，它们可以被放置在另一个称为数据区的**静态区域**中
- 放置在这个区域的数据包括全局常量、初始化的全局和静态变量以及他们的值
- 静态区的数据其**地址**可以被**编译到代码中**，这可以提高程序的运行效率

堆和栈

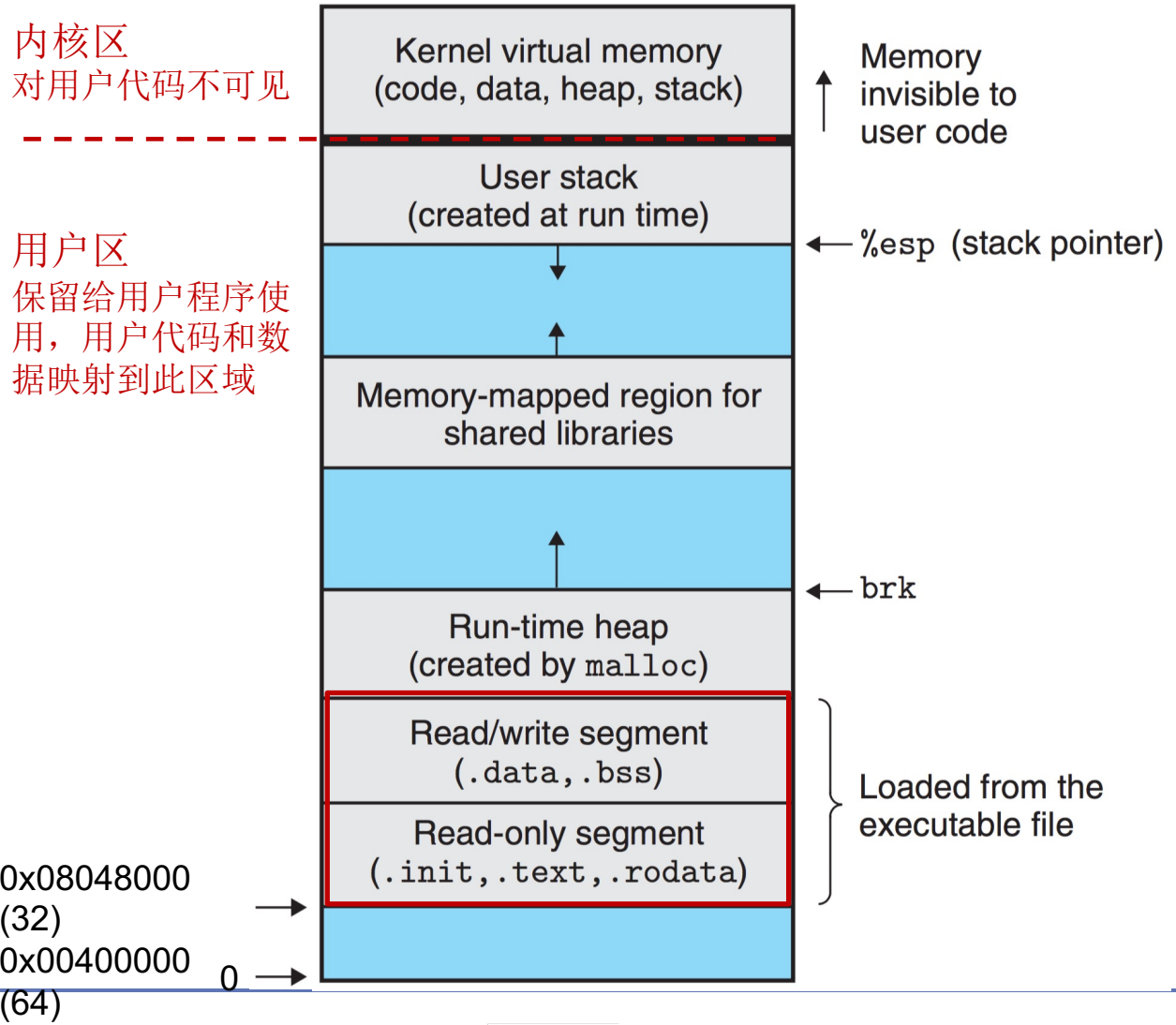
- 动态区域
  - 为了将运行时刻的空间利用率最大化，专门设计了动态区域堆和栈，用于存放会在程序运行时动态创建和回收的数据
- 栈（stack）：
  - 栈区用来存放称为**活动记录**（ activation record 或帧 frame）的数据结构，这些活动记录主要用于**子程序的调用**
- 堆（heap）：
  - 堆区主要用于在运行时存放动态产生的较大的数据，这部分数据没有名字，主要是通过指针（或引用）来获取的



进程的内存映射布局

- 在现代操作系统设计中，进程是程序的一个运行单元
- 进程独享一个由操作系统管理的虚拟内存（编程模型假定可访问除系统内核区之外的整个地址空间）
- 左图是一个常见的Linux进程的内存映射布局

私有地址空间



- 进程为每个程序提供一种假象，好像它**独占**使用系统地址空间（从这个意义来看，是**私有**的）
- 重点关注**数据段**、**代码段**、**堆**和**栈**
- 注意地址和指针

内核区与用户区

- 地址空间的**顶部**是保留给内核的
  - 地址空间的这个部分包含内核在代码进程执行指令时（比如，当应用程序执行一个**系统调用**时）使用的代码、数据和栈
- 地址空间**底部**是保留给用户程序的
  - 最主要的部分包括代码、数据、堆和栈段
  - 代码段起始位置一般0x08048000(32位)或0x00400000(64位)
  - 最下面一部分是保留区（用于空指针处理等）

用户模式和内核模式

- 处理器必须能限制一个程序可以执行的指令以及可访问的地址空间范围
  - 处理器通常用**控制寄存器**的一个位模式位（mode bit）来提供这种控制功能，该位描述了进程当前享有的特权
- 内核模式（超级用户模式）
  - 该模式下的进程可以执行指令集中的任何指令，并且而已访问系统中任何存储器的位置
- 用户模式（默认的模式）
  - 进程不允许执行特权指令，如**停止处理器、改变模式位、发起一个I/O操作**，也不允许直接引用地址空间中内核区内的代码和数据
  - 用户程序必须通过**系统调用接口**间接的访问内核代码和数据

UNIX的可执行格式

- UNIX系统下，编译时默认的可执行文件的名字是a.out
  - 这是assembler output（汇编程序输出）的缩写，同时也是早期UNIX系统上的可执行文件格式（现在已不太常用）
  - 最初在**PDP-7**上，确是将源程序连接后进行汇编，并放在a.out文件中，在PDP-11上已经不这样了，但最后输出文件的名字仍然沿用这个习惯
- ELF格式（目前主要的UNIX家族可执行文件格式）
  - UNIX体系下的可执行文件，大部分采用了ELF（原意为Extensible Linker Format，**可扩展链接器格式**，现在代表Executable and Linking Format，**可执行和链接格式**）
  - UNIX上可以用**file**命令查看文件格式

段或节

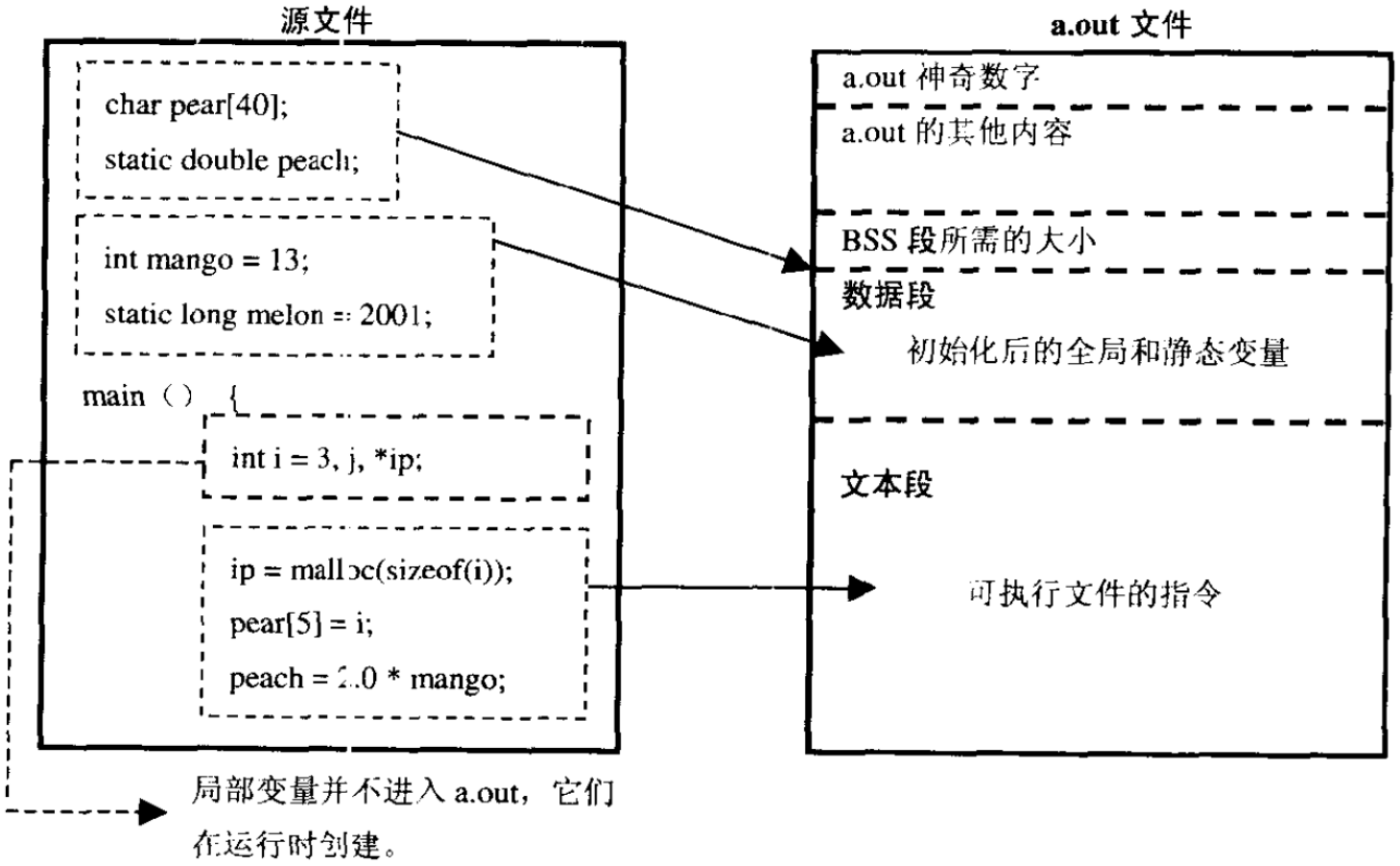
- 不同的可执行文件虽然格式会有差异，但主要的构造格局大致相似，都是使用**段（segment）**或称**节（section）**来组织内容
- 使用段的意义
  - 虽然不同的可执行文件的格式不同的，但都会都会使用类似的段的形式来组织文件内容
  - **链接**和**装载**都是基于段的方式来处理的，这样更容易操作系统对程序进行管理，例如设置不同段的可读写性

重要的节

- 常见的可执行文件的格式有很多种，如ELF、COFF等，但总会包含这样几个合成的节（section）或段（segment）
  - **.bss节：保存程序内存镜像中未初始化的数据**
  - **.data节：保存程序内存镜像中的初始化数据**
  - .text节：保存代码，也就是程序的可执行指令

- 注：根据惯例，段的名字以点(.)开头，一些重要段类型的名字是平台无关的 （在不同平台和二进制文件格式）。

C示例


源文件
char pear[40];
static double peach;
int mango = 13;
static long melon = 2001;
main ( ) {
int i = 3, j, *ip;
ip = malloc(sizeof(i));
pear[5] = i;
peach = 2.0 * mango;
局部变量并不进入 a.out，它们在运行时创建。
a.out 文件
a.out 神奇数字
a.out 的其他内容
BSS 段所需的大小
数据段
初始化后的全局和静态变量
文本段
可执行文件的指令

内存映射的创建

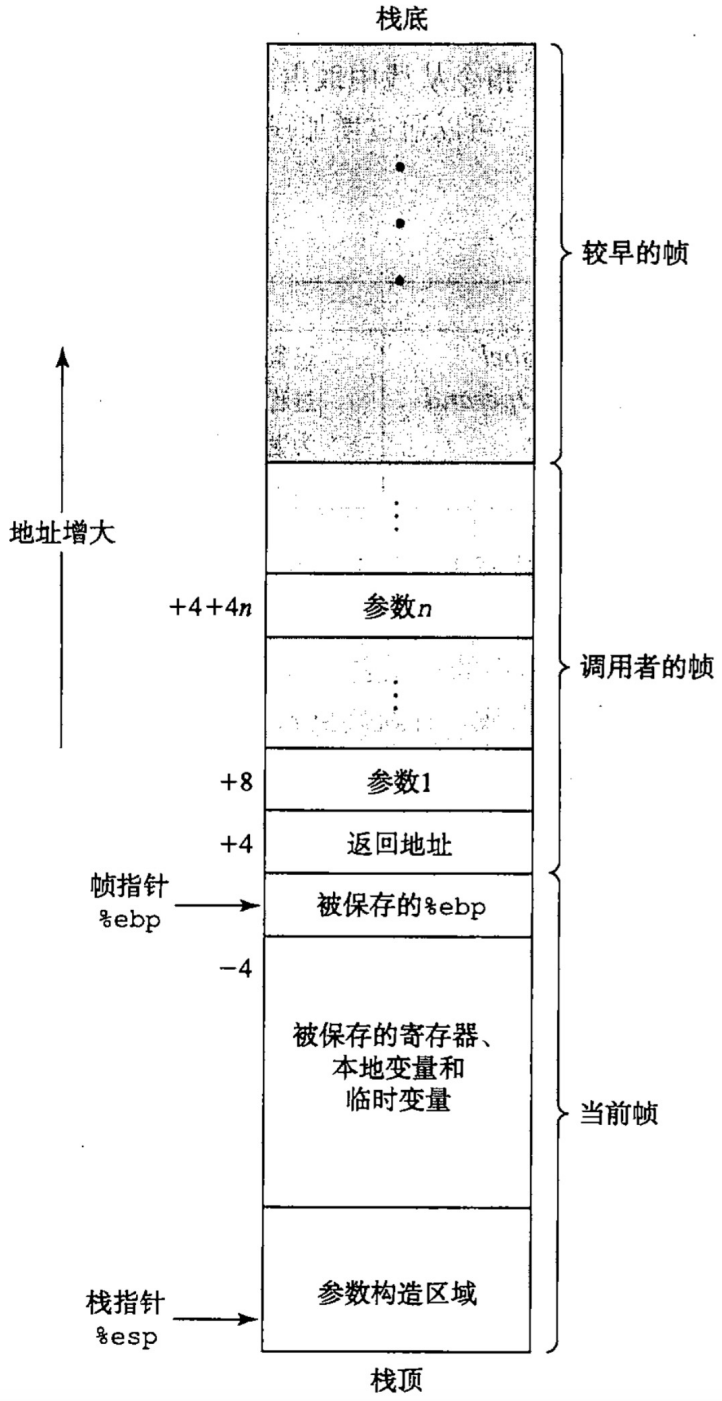
- 程序运行过程中的内存映射是如何创建的：
  - 程序的二进制文件中包含了程序运行过程中的内存映射布局的细节
  - 链接器将编译好的二进制模块合并，填充各个内存映射**段**（或**节**），以创建可执行二进制文件的整体框架
  - 进程内存映射的初始化建立工作是通过**程序装载器**这一系统工具来完成的

子程序的运行模型

- 子程序的**基本运行模型**主要涉及到如下一些概念
  - 运行时的内存格局
  - 控制栈（control stack）
  - 活动记录（activation record）或帧（frame）
  - 堆动态数据
  - 子程序的调用过程
    - 重点讲解栈动态局部变量的语言（如C语言）的子程序调用过程

子程序的执行

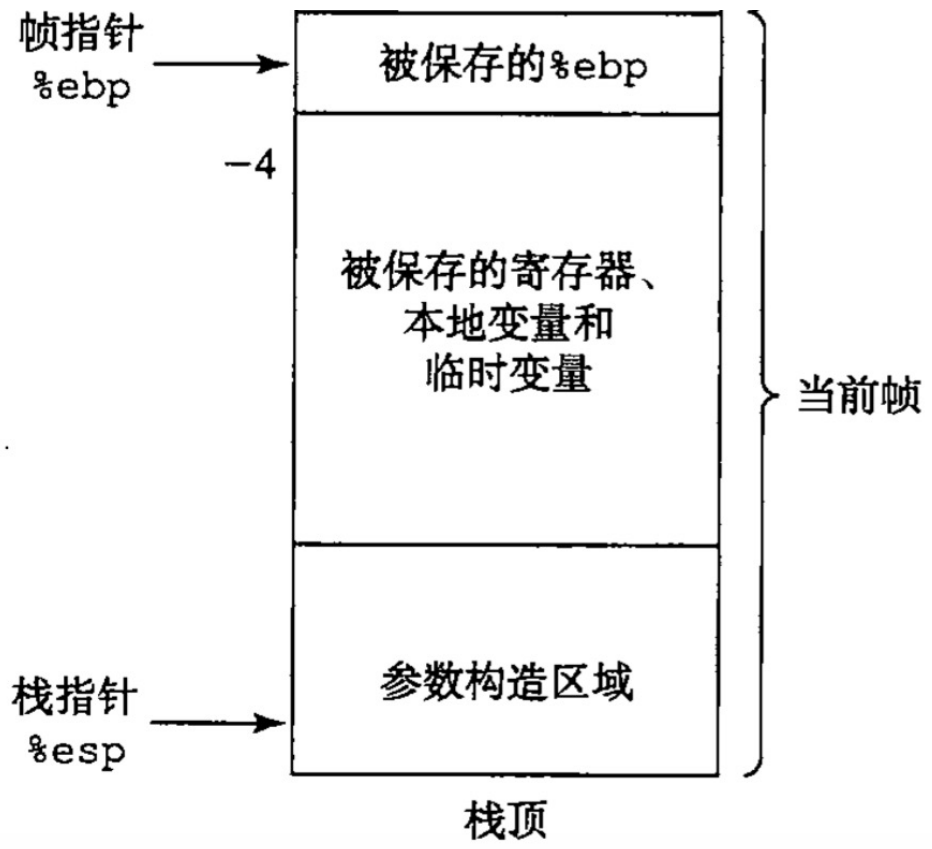
- 一般而言，子程序的执行包括
  - 将**数据**（以子程序参数和返回值的形式）和**控制**从程序代码的一部分传递到另一部分
  - 在进入子程序时为**局部变量**分配空间，并在退出时释放这些空间
- 注意：
  - 大多数机器只提供转移控制到过程和从过程中转移出控制这种简单的**指令**
  - 数据传递、局部变量的分配和释放通过操作**程序栈**来实现



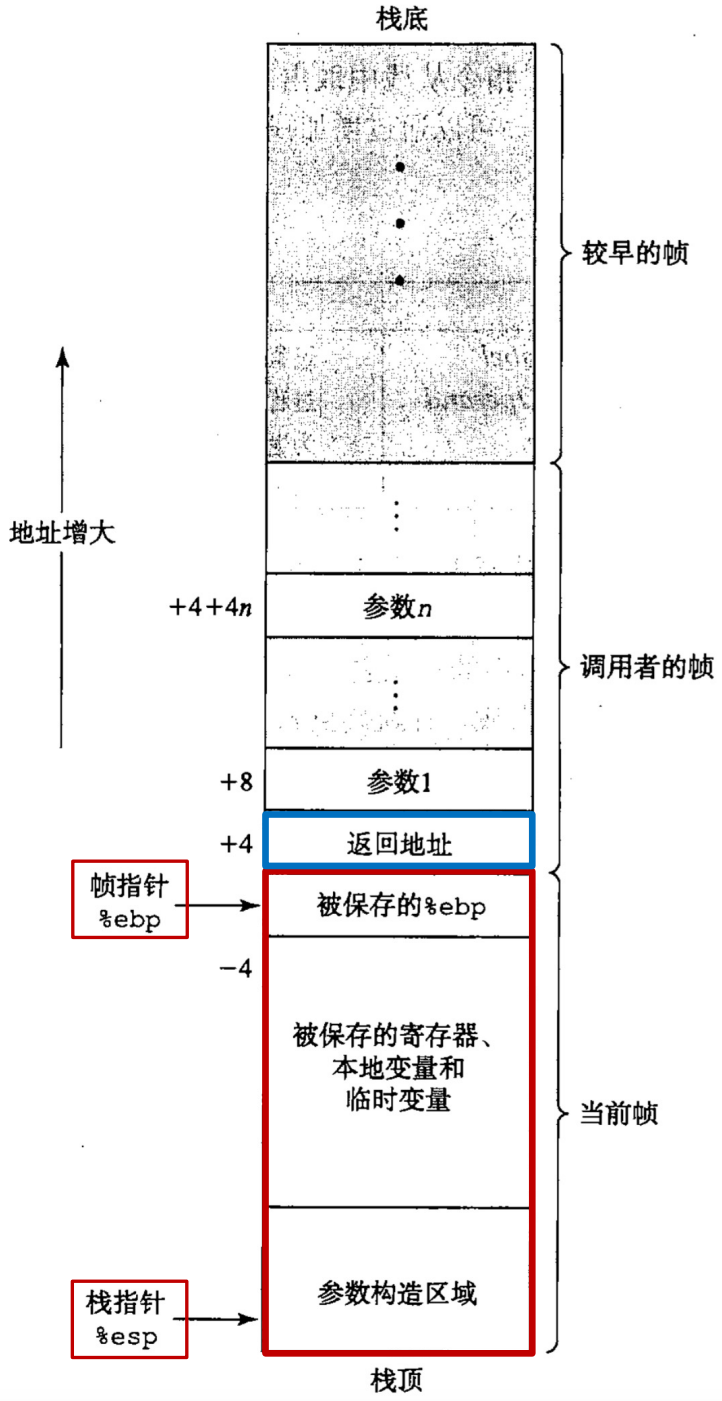
子程序的栈帧

- **参数传递**和**值的返回**方式（也称为**调用惯例**）是子程序调用最核心的问题之一
- 子程序的调用惯例主要依靠内存中的**程序栈**完成
- 左图是一**种**典型的栈帧结构

顶端的栈帧



- 为单个子程序分配的那部分栈称为栈帧（stack frame）
- 最顶端的栈帧以两个指针界定，寄存器%ebp为**帧指针**，而寄存器%esp为**栈指针**
- 子程序运行时，帧指针不移动，因此可以作为基地址来访问子程序的实际参数等



- 假设过程P（调用者）调用过程Q（被调用者），则Q的参数放在P的栈帧中
- 当调用Q时，P的返回地址被压入栈中，形成P栈帧的末尾
- Q的栈帧从保存帧指针的值（如寄存器%ebp的副本）开始，后面是保存其它寄存器的值

协同程序

- 协同程序（coroutine）是一种特殊的子程序
  - 传统的子程序调用时，调用程序与被调用者之间存在主从关系
  - 协同程序则采用了对称单元控制模型（symmetric unit control model）的机制
  - 程序执行的控制权在系统程序之间交替转移，而在交替执行的过程中，协同程序可以记住自己的**历史信息**

执行过程

- 协同程序常常在执行**一部分语句**之后，将控制权移交给另一个协同程序，而不是一次执行完毕（如到return结束）
- 在另一个协同程序中，也可以在执行中把控制权再交回来
- 重启协同程序时，它会从移交控制权的那条语句的下一条移交开始执行
- 以上过程可以反复交替，直至程序结束

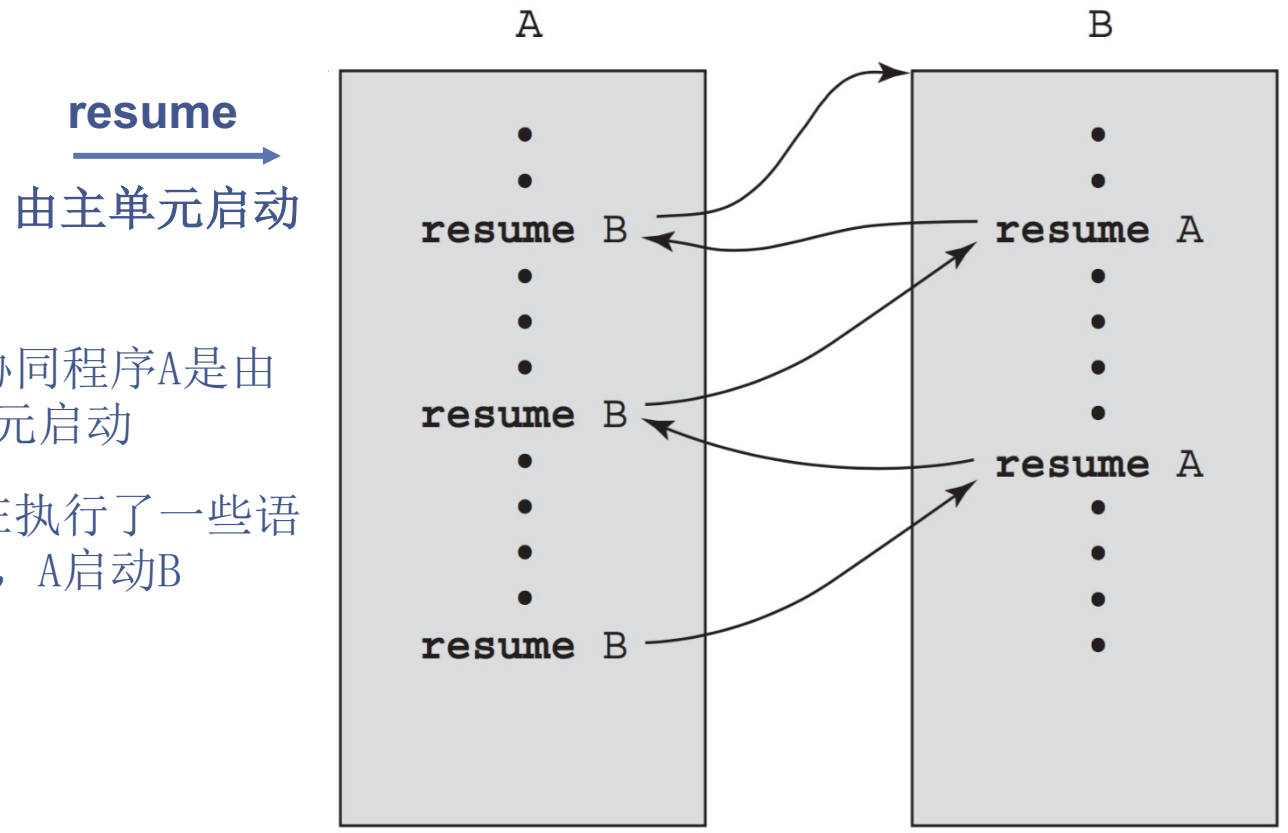
历史敏感

- 历史敏感性
  - 协同程序的最大特点是在交替的执行过程中可以维持自身的状态
  - 协同程序应用场景也正是这种带历史影响的子程序交互执行
  - 为了保持对历史的敏感性，系统程序会采用静态的局部变量
  - 注：协同程序的调用称为恢复

主单元

- 一般情况下，支持协同程序的应用程序使用名为“主单元”的程序单元创建（主单元不是协同程序）
- 协同程序创建时，会执行初始化代码，然后将控制权交还给主单元
- 主单元负责构造出整个系列（执行控制权会在他们之间相互传递）的协同程序
- 之后，主单元会继续执行其中一个协同程序，从而开始该系列协同程序之间的交替执行

应用场景



1、协同程序A是由主单元启动

2、在执行了一些语句后，A启动B

3、B第一次将控制权交给A时，A会从上一次执行结束的地方继续执行（A的局部变量保持和上一次执行结束时一致）

典型问题

- 利用协同程序的方式求解的一个典型问题是模拟扑克牌游戏
  - 主单元创建四个协同程序，每个协同程序都初始化自己的一手牌
  - 主单元启动第一个出牌的协同程序，该协同程序在出牌后恢复执行下一个协同程序
  - 如此循环，直至游戏结束

- 现代语言中，只有Lua完全支持协同程序

附录1：C的main参数

- main函数是个非常特殊的函数，其参数传递也很有意思
- C/C++语言中的main函数，经常带有参数argc，argv，如下：
  - int main(int argc, char** argv)
  - int main(int argc, char* argv[])
- 主要体会一下，char** argv 和 char* argv[] 可以达到同样的效果，但其实是不一样的解释
  - char** argv是双重指针：指向“指向char的指针”的指针
  - char* argv[]是指针的数组：“指向char的指针”的数组

附录2：C中指针、数组和字符串

- C中字符串就是char的数组
  - char str[] = “abc” 是语法糖，其实就是
  - char str[] = {‘a’, ‘b’, ‘c’, ’\0’}，共4个元素
- 实际编程中，往往这样用：
  - char * str = “abc”，这定义了一个“**指向字符的指针**”，指向的是该字符串首字符’a’的地址
  - 字符串常量一般保存在只读内存区域，所以
  - 实际应用中多用“**指向字符的指针**”来操作字符串

参考文献

1. 编程语言原理 10E，Robert Sebesta，清华大学出版社
2. C专家编程，Peter van der Linden，人民邮电出版社
3. 深入理解计算机系统（第2版），Randal Bryant等，机械工业出版社
4. 编译原理（龙书2），Alfred Aho等，机械工业出版社
5. 面向对象分析与设计（第3版），Grady Booch等，电子工业出版社