第七章　运行时刻环境

南京大学
戴新宇
2021-6

主要内容

- 概要
- 存储组织
- 栈管理
- 堆管理
- 垃圾回收
- 总结

概要

- 在编译器分析源程序语言中包含的各种信息并产生目标代码后，将于操作系统协作，在目标机器上执行代码。
- 为此，编译器创建并管理*运行时刻环境*：目标程序的运行环境
  - 代码和数据的存储分配
  - 函数间的衔接
  - 参数传递
  - 与操作系统、输入输出设备的接口等

编译系统存储组织

- 编译系统给程序里的对象分配空间，包括代码，静态和动态数据，全局和局部数据等
- 本章讨论存储位置的分配以及对代码和数据的访问，重点讨论存储管理
  - 栈分配
  - 堆管理
  - 垃圾回收

存储管理

- 编译程序对目标程序运行时的数据空间（逻辑地址空间）的组织和管理。
- 常见的编译器对于目标程序运行时刻的空间划分
  - 以连续字节块存储
  - 字节是内存最小的编址单位
  - 一个字节是8个二进制位
  - .....

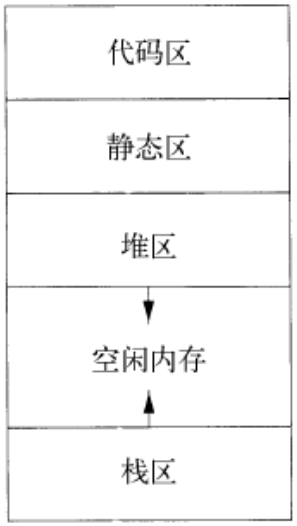
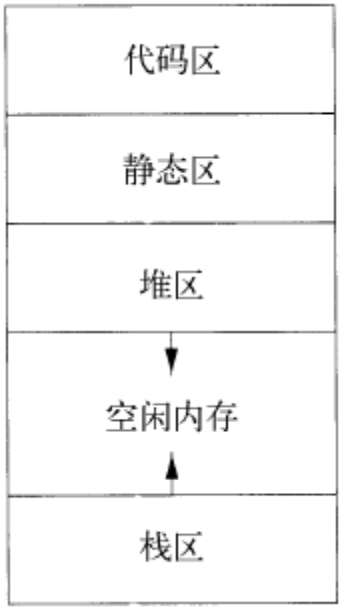


图 7-1　运行时刻内存被划分成代码区和数据区的典型方式

存储管理（续）



- 代码区：放置可执行目标代码的数据区域，在编译时刻该区域的大小就可以确定。这个区通常在存储的低地址端
- 静态区：程序的某些数据对象，如全局常量和编译器产生的某些数据
- 动态区域
  - 堆区和栈区被放在剩余地址空间的两端。它们的大小随着程序运行会发生改变。注意栈区和堆区的增长方向

静态和动态存储分配

- 数据在运行时刻环境中的内存位置的布局及分配是存储管理的关键问题
- 静态和动态
  - 观察源程序，在编译时刻就能够确定某些数据的存储分配，这样的存储分配称为静态
  - 在运行时刻才能决定的存储分配，称为动态
- 动态存储分配
  - 栈式存储
    - 栈式存储与函数调用相关
  - 堆式存储

空间的栈式分配

- 函数是常用语言中对某些特定动作单元的抽象。
- 与函数调用相关的数据在栈中进行存储和管理
  - 当一个函数被调用时，用于存放该函数的局部变量的空间被压入栈
  - 当这个函数结束时，该空间从栈中弹出
  - 栈区空间可以被多个活动阶段不重叠的函数共享

函数及调用

- 函数包括函数名和函数体
- 函数可能带有参数：形参、实参
- 函数调用：函数名出现在可执行的语句中，执行被调用函数的函数体。
- 函数体的一次执行称为该函数的一个活动（Activation）。
- 活动生存期：从函数体执行的第一步到最后一步的步序列。
- 函数活动的嵌套特性：即一个函数p的一个活动调用了函数q，那么q的该次活动必定在p的活动结束之前结束。

函数调用示例

- 使用递归的快速排序算法

```
enter main()
    enter readArray()
    leave readArray()
    enter quicksort(1,9)
        enter partition(1,9)
        leave partition(1,9)
        enter quicksort(1,3)
            ...
        leave quicksort(1,3)
        enter quicksort(5,9)
            ...
        leave quicksort(5,9)
    leave quicksort(1,9)
leave main()
```

```
int a[11];
void readArray() { /* 将 9 个整数读入到 a[1], ..., a[9]中。*/
    int i;
    ...
}
int partition(int m, int n) {
    /* 选择一个分割值 v , 划分 a[m..n],
       使得 a[m..p-1] 小于 v, a[p] = v ,
       并且 a[p+1..n] 大于等于 v 。返回 p. */
    ...
}
void quicksort(int m, int n) {
    int i;
    if (n > m) {
        i = partition(m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
}
main() {
    readArray();
    a[0] = -9999;
    a[10] = 9999;
    quicksort(1,9);
}
```

活动树

- 我们可以用一颗树来说明在整个程序运行期间的所有函数的活动，这棵树称为活动树。
- 树中每个节点对应于一个活动，根节点是main函数的活动。在表示函数p的某个活动的节点上，其子节点对应于被p的这次活动调用的各个函数的活动。我们按照这些活动被调用的顺序，自左向右地表示它们。
- 注意：一个子节点必须在其右兄弟节点的活动开始之前结束。（不交迭）

```
    enter quicksort(1,9)
        enter partition(1,9)
        leave partition(1,9)
        enter quicksort(1,3)
            ...
        leave quicksort(1,3)
        enter quicksort(5,9)
            ...
        leave quicksort(5,9)
    leave quicksort(1,9)
leave main()
```

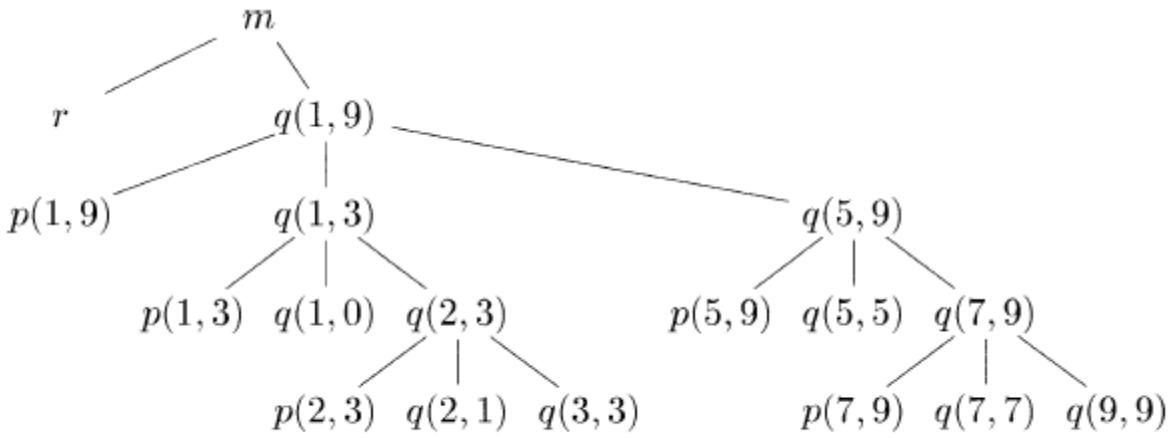


图 7-4　表示 *quicksort* 的某次运行中的调用的活动树

- 活动树与程序行为间的对应关系
  - 函数调用序列和活动树的先序遍历相对应
  - 函数返回序列和活动树的后序遍历相对应
  - 假定控制流在某个函数的活动中，该活动对应于树上节点N。N及其祖先节点是当前活跃的活动。这些函数调用的顺序是它们从根节点到N的路径上出现的顺序。这些活动按照该顺序的反向顺序返回。
- 先进后出.....适合栈存储

活动记录

- 函数的调用和返回的相关数据和信息由控制栈(运行时刻栈)进行管理。
- 每个活跃的活动有一个位于栈中的活动记录。
- 活动树的根位于栈底，栈中全部活动记录的序列对应于在活动树中从根节点到达当前活跃的活动节点的路径。当前程序控制所在活动的活动记录位于栈顶
- （存储所有的尚未返回的函数调用信息）

活动记录存储示例

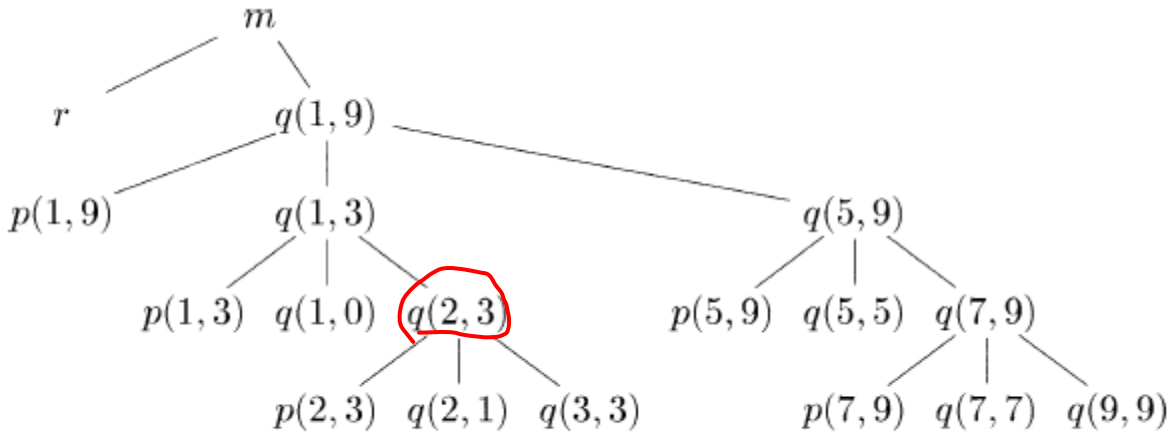


图 7-4　表示 *quicksort* 的某次运行中的调用的活动树

- 假定栈底在上

活动记录的内容

- 临时变量：某些临时中间结果
- 局部数据：保存该函数执行中使用的局部数据
- 机器状态信息：在对此函数的此次调用之前的机器状态信息，函数返回时需要恢复这些信息
- 访问链：在其它活动记录中存放的数据，访问时需要访问链进行定位。
- 控制链：指向调用者的活动记录
- 返回值：存放被调用函数返回给调用函数的值
- 实在参数：存放调用函数提供的实在参数

运行时刻栈示例

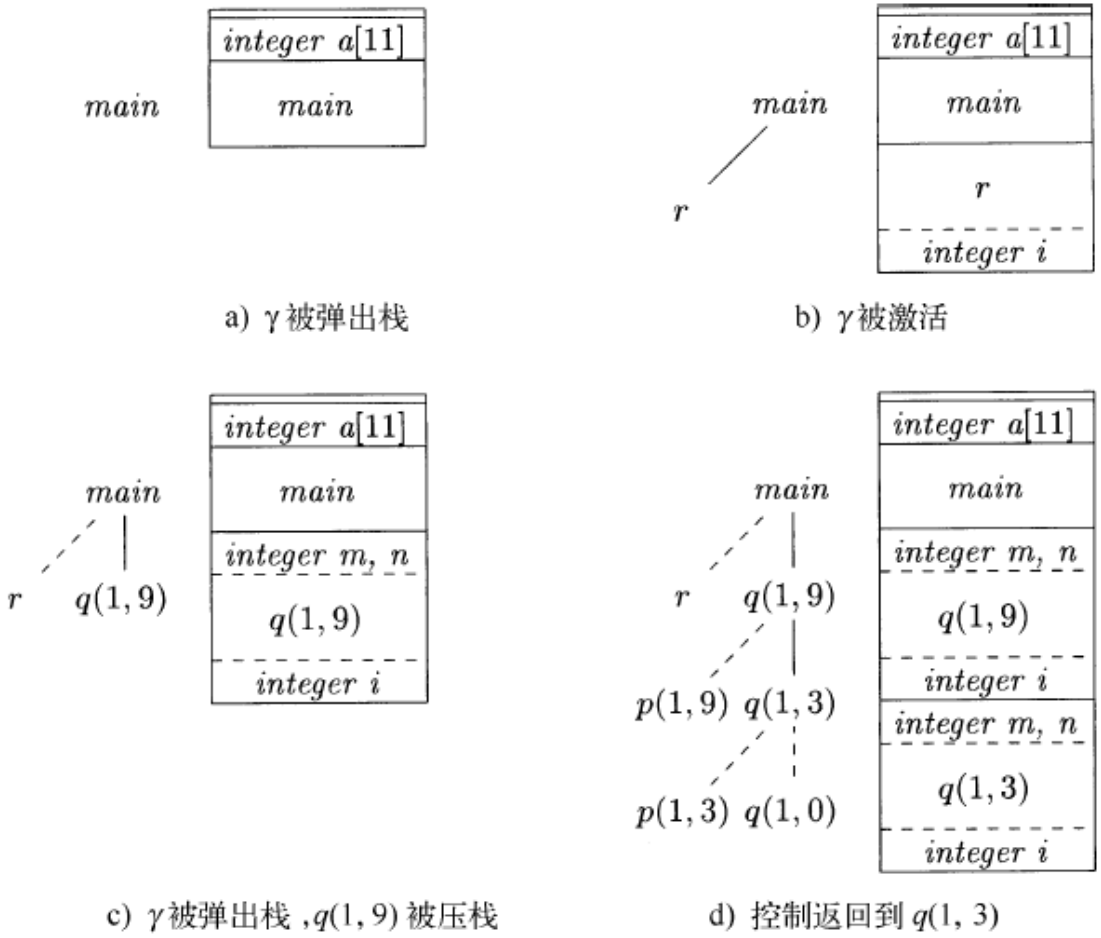


图 7-6　向下增长的活动记录栈

调用代码序列、返回代码序列

- 实现函数调用的代码称为调用代码序列
  - 为一个活动记录在栈中分配空间，并在此记录的字段中填写信息
- 返回代码序列是恢复机器状态，使得调用函数能够在调用结束后继续执行的代码。
- 调用代码序列分为调用者和被调用者的代码
- 调用代码设计原则，要与活动记录布局的设计相呼应
  - 调用者和被调用者之间传递的值（包括参数和返回值），一般被放在被调用者活动记录的开始位置
  - 固定长度的项（包括控制链、访问链和机器状态字段）放在记录的中间位置
  - 开始不知道大小的项（例如动态数组）被放置在活动记录的尾部
  - 确定“栈顶”指针所指的位置

调用者和被调用者合作管理栈示例

- 调用代码
  - 调用者计算实参的值
  - 调用者将返回地址和top_sp的值放在被调用者的活动记录中，然后增加top_sp的值
  - 被调用者保存寄存器值和其它状态信息
  - 被调用者初始化其局部数据并开始执行
- 返回代码
  - 被调用者将返回值放在与参数相邻的位置
  - 被调用者恢复top_sp和其它寄存器，然后跳转到由调用者放在机器状态字段中的返回地址
  - 尽管top_sp已经被减小，但调用者仍然可以根据当前top_sp的位置找到返回值。

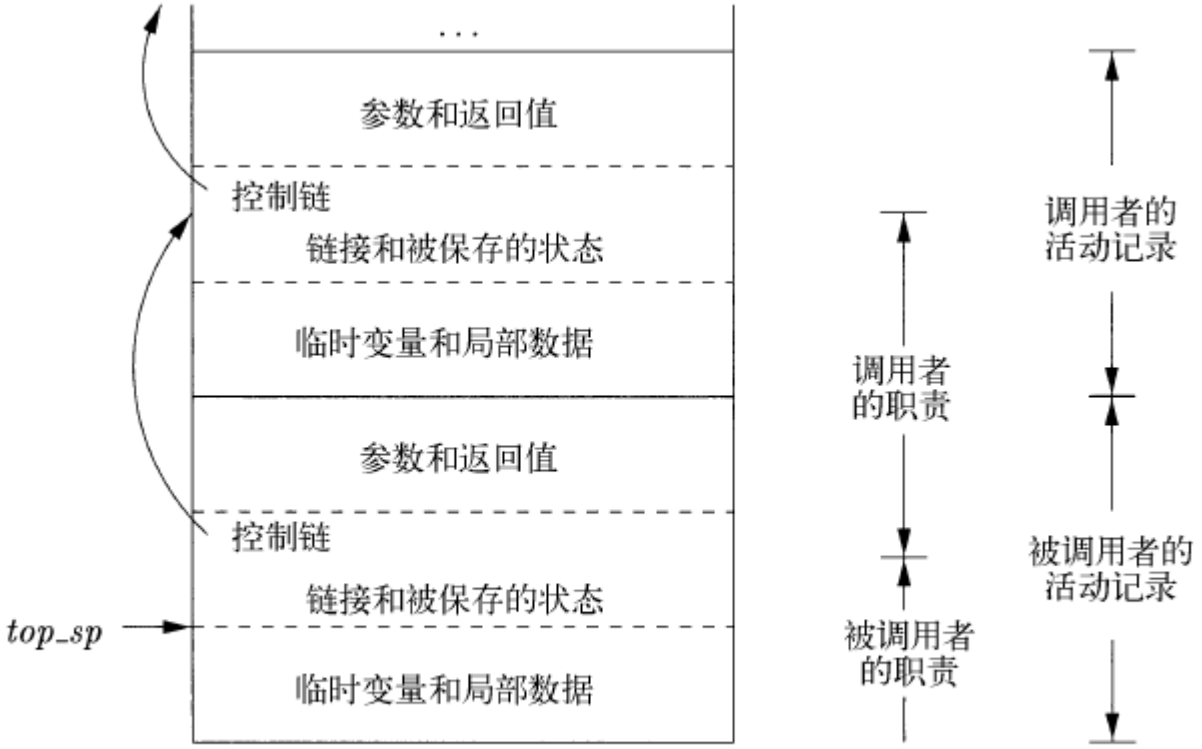


图 7-7　调用者和被调用者之间的任务划分

堆管理

- 堆用来存储那些生命周期不确定，由程序显式删除来结束生存期的数据对象。它们的生存期与函数无关。
- 如C++，java中的new出来的对象。
- 存储管理器:用来分配和回收堆区空间
- 手工回收，free delete
- 自动回收，垃圾回收器。

存储管理器

- 两个基本功能
  - 分配空间。当程序为一个变量或对象请求内存空间时，存储管理器产生一段连续的满足被请求大小的堆空间。如果有足够的空间，则分配，如果没有足够的空间，则获得虚拟内存以增加堆区空间，再没有空间，则给出相应信息。
  - 回收空间。存储管理器把回收的空间返还到空闲空间中，从而可以复用该空间以满足其它的分配请求。
- 最好具有下列特性
  - 空间效率。尽量减少存储碎片。
  - 程序效率。能够充分利用存储子系统。
  - 低开销。减少分配和回收的时间。

计算机的存储层次结构

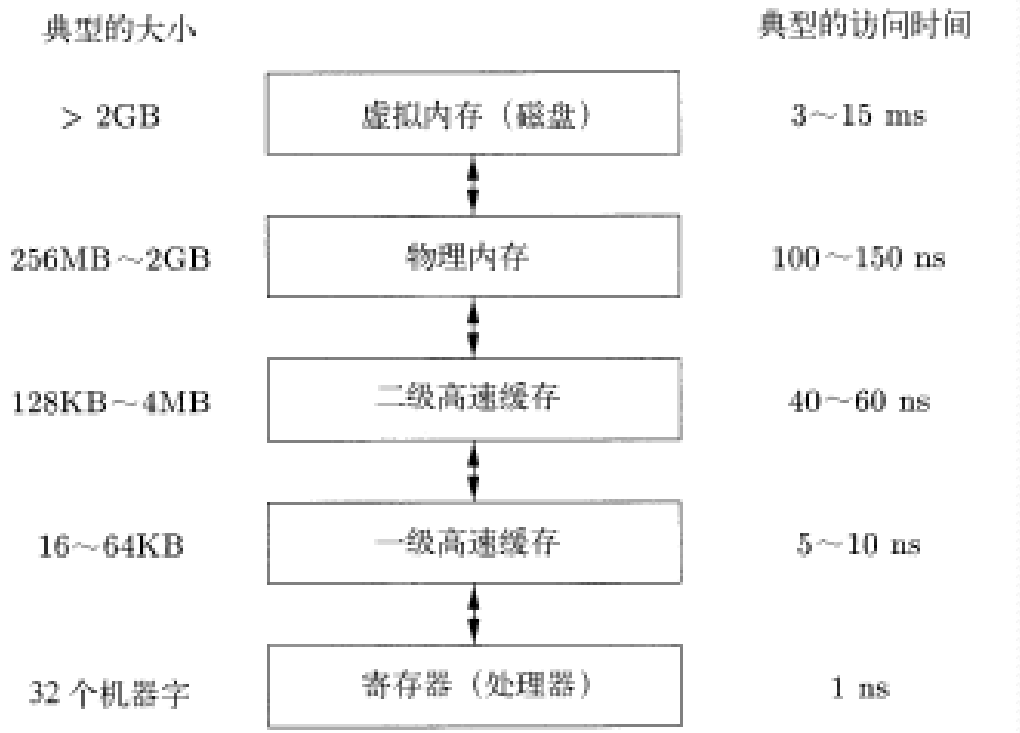


图 7-16　典型的内存层次结构的配置

程序中的局部性

- 程序具有高度的局部性(locality)
  - 时间局部性：一个程序访问的存储位置很可能将在一个很短的时间段内被再次访问
  - 空间局部性：被访问过的存储位置的临近位置很可能在一个很短的时间段内被访问。
- 90%的时间用来执行10%的代码
- 局部性这一特性恰好可能充分利用计算机的层次储存结构
- 需要动态调整最快存储中的存储的指令

碎片整理

- 堆区从连续的一个空闲单元，经过若干次分配和回收，空间被分割成若干空闲存储块和已用存储块。
- 空闲存储块被称为“窗口”（hole）
- 可能会有越来越多、越来越小的小“窗口”
- 尽可能减少碎片的空间分配策略
- 对空闲空间进行管理和结合
  - Best-fit:将请求的存储分配在满足请求的最小可用窗口中。将大的窗口保留下来满足后续的更大请求。
  - First-fit: 对象被放置在第一个（地址最低）能够容纳请求对象的窗口中。

碎片整理（续）

- 尽可能减少碎片的空间分配策略
  - Best-fit:将请求的存储分配在满足请求的最小可用窗口中。将大的窗口保留下来满足后续的更大请求。
  - First-fit: 对象被放置在第一个（地址最低）能够容纳请求对象的窗口中。
- 对空闲空间进行管理和结合：将连续的空闲空间合并，以满足后续更大请求的需要。
  - 边界标记：标记free /used
  - 一个双重链接的、嵌入式的空闲列表

一个堆的片段

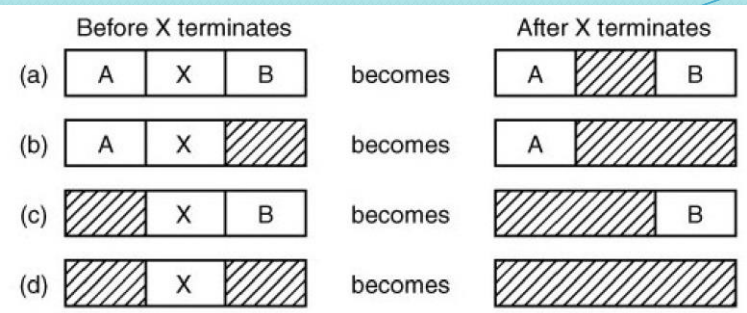


Fig:Four neighbor combinations for the terminating process, X.

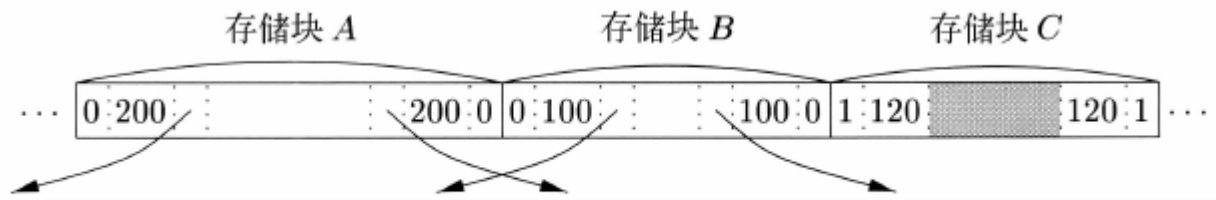


图 7-17　堆的片段和一个双重链接的空闲列表

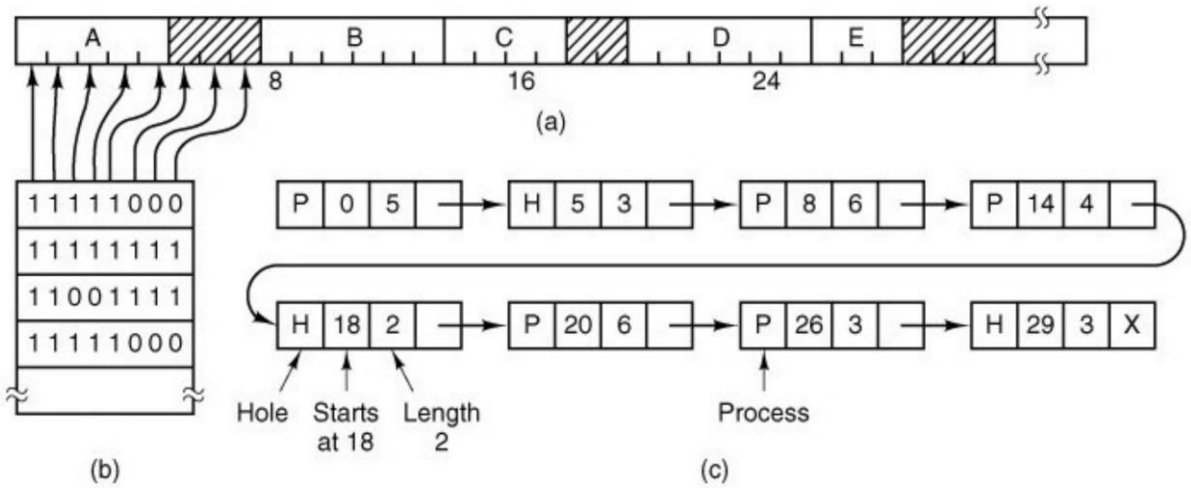


Fig:(a) A part of memory with five processes and three holes. The tick marks show the memory allocation units. The shaded regions (0 in the bitmap) are free. (b) The corresponding bitmap. (c) The same information as a list.

人工回收请求

- 理想情况，删除所有不会再被访问的存储
- 人工回收带来的问题
  - 内存泄漏：一直未能删除不会被引用的数据
  - 悬空指针引用：引用已经被删除的数据

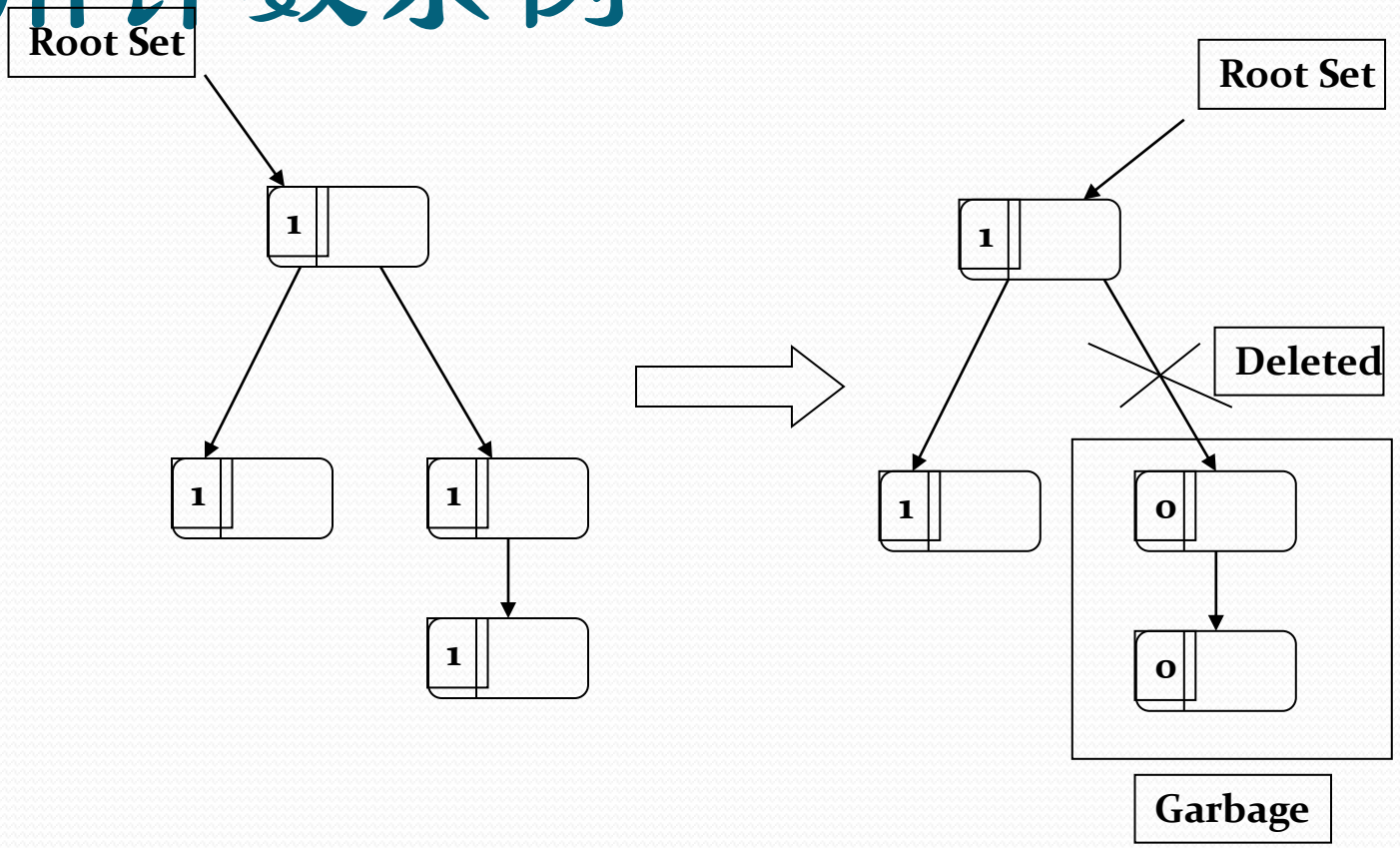
垃圾回收概述

- 垃圾：不能被引用的数据
- 基本思想：可达性分析。分析一个存储是否可以被引用或到达。
- 根集：不需要任何指针操作就可以被程序直接访问的数据。如在java中，所有静态区数据和栈中所有变量。
- 可达对象的集合随着程序的执行而变化
  - 对象分配。返回一个指向新创建的存储区域的引用。这个操作向可达对象集中添加成员。
  - 参数传递和返回值。对象引用从实参传递给形参，从返回结果传回给调用者。这些引用指向的对象可达。
  - 引用赋值。对于引用u和v， u=v的赋值。
  - 函数返回。
- 总体说来，新的对象通过对象分配可达。参数传递和赋值可以传递可达性。赋值和函数返回可能结束对象的可达性。当一个对象变得不可达时，可能会导致更多的对象变得不可达。

垃圾回收概述（引用计数法）

- 用引用计数方法定位所有可达对象
- 引用计数
  - 对象分配。新对象的引用计数被设置为1
  - 参数传递。被传递给一个函数的每个对象的引用计数加1
  - 引用赋值。若u和v都是引用，对于语句u=v, v指向的对象的引用计数加1，u原来指向的原对象引用计数减1
  - 函数返回。该函数活动记录的局部变量中所指向的对象的引用减1.
  - 可达性传递丢失。当一个对象的引用计数变成0时，我们必须将该对象中的各个引用所指向的每个对象的引用计数减1

引用计数示例



引用计数不能回收不可达的循环数据结构

- 三个对象相互引用，没有来自外部的指针，又不是根集成员，都是垃圾，但是引用计数都大于o.

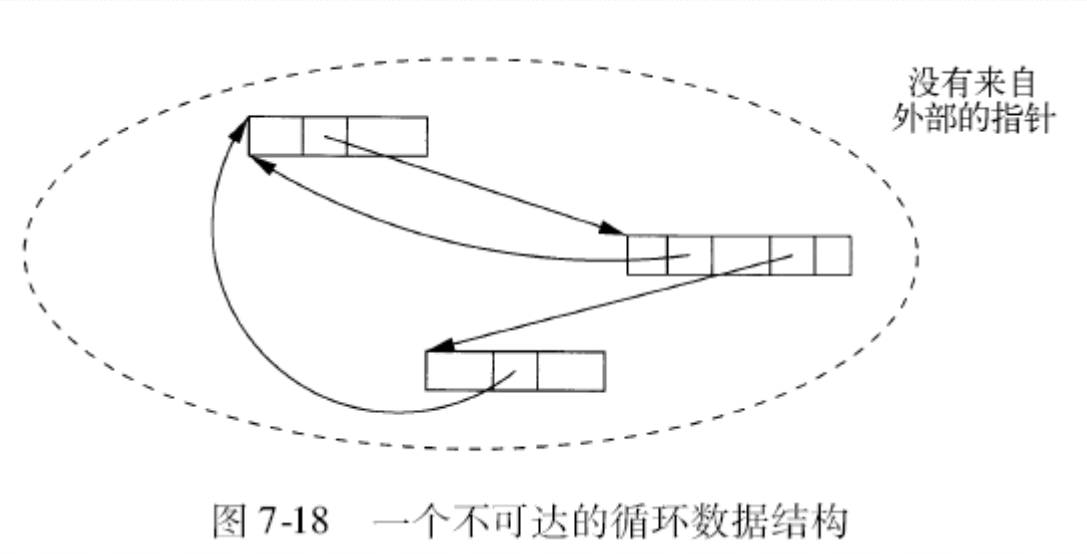


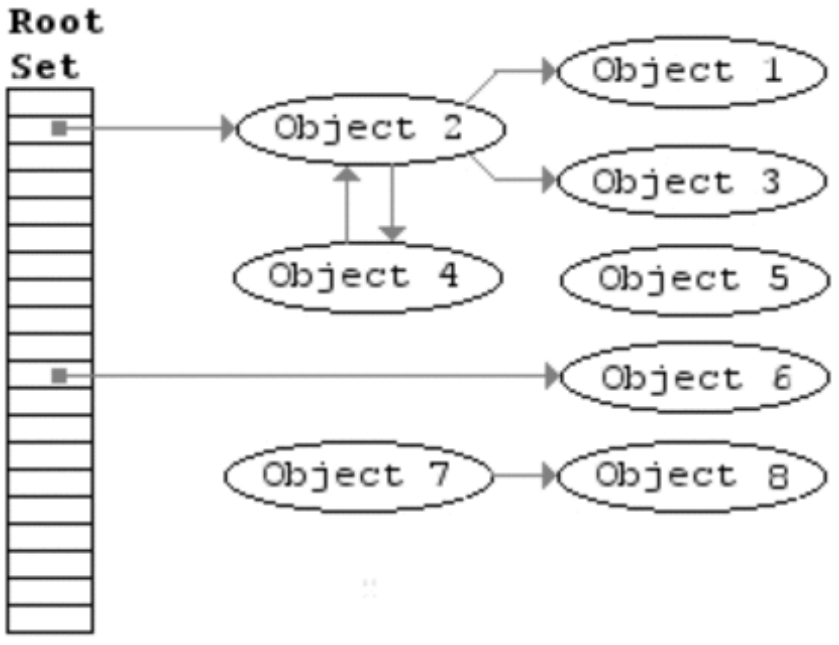
图 7-18　一个不可达的循环数据结构

基于跟踪的垃圾回收

- 垃圾回收器周期性的运行，通常在程序请求分配，但是是在没有充足空间的时候触发回收。
- 垃圾回收器被触发后运行，程序运行中止。
- 有三种方法
  - 基本的标记清扫式回收器
  - 标记压缩垃圾回收器
  - 拷贝回收器

基本的标记清扫式回收器

- 两个阶段
  - 标记：找出所有可达对象
  - 清扫：回收垃圾空间

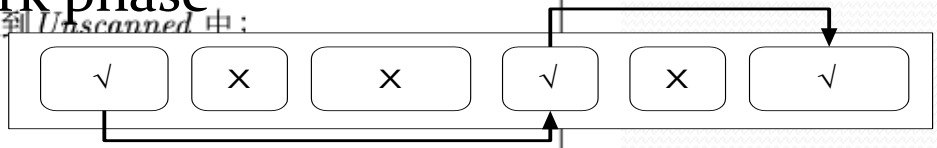


基本的标记清扫式回收器（续）

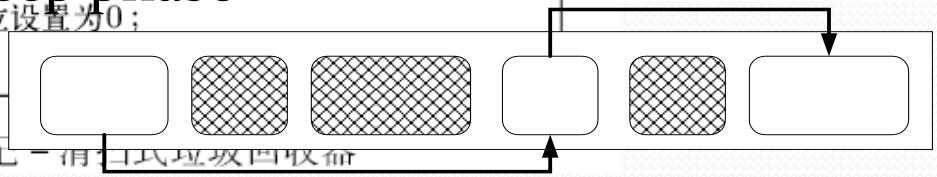
```
    /* 标记阶段 */
1)  把被根[illegible]的每个对象的 reached 位设置为 1，并把它加入
        到 Unscanned 列表中；
2)  while (Unscanned ≠ ∅) {
3)          从 Unscanned 列表中删除某个对象 o；
4)          for (在 o 中引用的每个对象 o′) {
5)                  if (o′ 尚未被访问到；即它的 reached 位为0) {
6)                      将o′的 reached 位设置为1；
7)                      将o′ 放到 Unscanned 中；
                    }
            }
    }
    /* 清扫阶段 */
8)  Free = ∅;
9)  for (堆区中的每个内存块o) {
10)         if (o未被访问到，即它的 reached 位为0)将o加入到 Free中；
11)         else 将o 的 reached 位设置为0；
    }
```

图 7-21　一个标记－清扫式垃圾回收器

- Mark phase



- Sweep phase



基本的标记清扫式回收器（续）

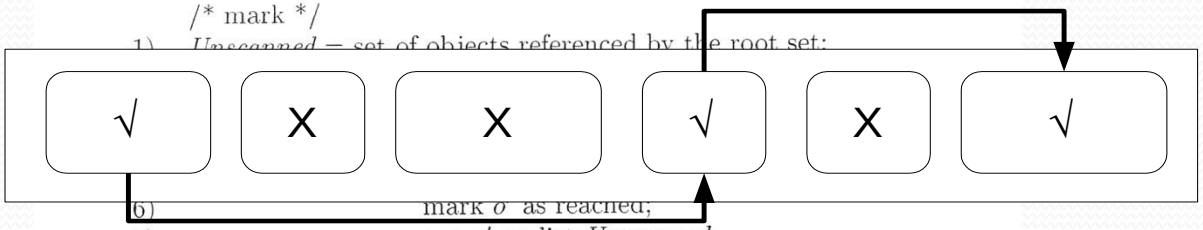
- 基本函数：计算可达对象的集合，然后取这个集合的补集。
  - 程序运行并发出分配请求
  - 垃圾回收器通过跟踪揭示可达性
  - 垃圾回收器收回不可达对象的存储空间

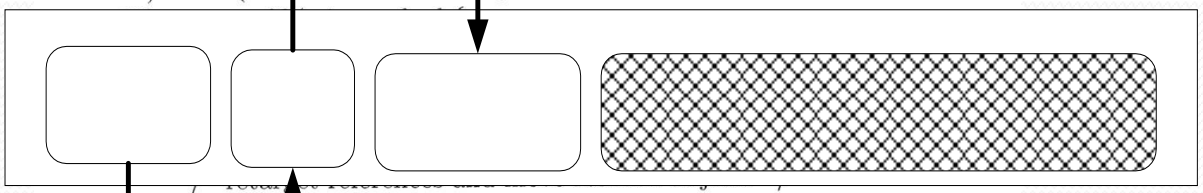
标记压缩回收器

- 在堆区内移动可达对象以消除存储碎片。
- 标记压缩垃圾回收器的三个阶段
  - 标记。和标记清扫算法相同
  - 扫描堆区中已分配内存段，并为每个可达对象计算新的地址。新地址从堆的最低端开始分配，因此在可达对象之间没有空闲存储窗口
  - 算法将对象拷贝到它们的新地址，更新对象中的引用，使之指向相应的新地址。

标记压缩回收器（续）

- 算法





```
    /* mark */
1)  Unscanned = set of objects referenced by the root set;
6)                  mark o' as reached;
7)                  put o' on list Unscanned;
                }
            }
        }
        /* compute new locations */
8)      free = starting location of heap storage;
9)      for (each chunk of memory o in the heap, from the low end) {
13)     for (each chunk of memory o in the heap, from the low end) {
14)         if (o is reached) {
15)             for (each reference o.r in o)
16)                 o.r = NewLocation(o.r);
17)             copy o to NewLocation(o);
            }
        }
18)     for (each reference r in the root set)
19)         r = NewLocation(r);
```

Figure 7.26: A Mark-and-Compact Collector

第7章总结

- 运行时刻组织
- 控制栈、栈分配、栈中非局部数据访问
- 堆管理、存储管理器、减少碎片、垃圾回收
- 可达性、引用计数、跟踪回收