CS61第五次课程记录

傅海平

Institute Of Computing Technology,

Chinese Academy Of Sciences

haipingf@gmail.com

November 18, 2011

Contents

1 Topics

动态内存管理

2 Progress

晚上7点开始，开始的两个半小时(7:00 - 9:30)学习 Lec10-Dynamic_memory_1.pdf 和 Lec11-Dynamic_memory_2.pdf 两张课程讲义，然后9:30开始讨论学习过程中遇到的问题 。

3 Learning Details

3.1 Course Sketch

- 动态内存管理
- 性能需求
- 内存碎片
- 空闲块链表管理
  - 隐式链表
  - 显式链表
  - 隔离链表
  - 权衡...
- 进程虚拟内存结构图,$.text, .rodata, .data, .bss, Heap, Stack$
  - 堆 heap

  - 程序通过malloc/realloc/free等库函数申请和释放内存，在Linux系统中，通过 brk() 进行系统调用申请内存，另外，sbrk() 不是系统调用，是C库函数。在Linux系统上，程序被载入内存时，内核为用户进程地址空间建立了代码段、数据段和堆栈段，在数据段与堆栈段之间的空闲区域用于动态内存分配。
  - 内核数据结构 mm_struct 中的成员变量 start_code 和 end_code 是进程代码段 的起始和终止地址，start_data 和 end_data是进程数据段的起始和终止地址，start_stack 是进程堆栈段起始地址，start_brk是进程动态内存分配起始地址（堆的起始地址），还有一个 brk（堆的当前最后地址），就是动态内存分配当前 的终止地址。C语言的动态内存分配基本函数是malloc（在Linux上的基本实现是通过内核的 brk系统调用。brk是一个非常简单的系统调用，只是简单地改变 mm_struct 结构 的成员变量brk的值。）

- 内存分配方式
  - 显式分配: malloc/free
  - 隐式分配: Java, Python中的垃圾回收机制
- 性能需求
  - 快速响应应用程序的内存申请
  - 高效，不浪费
  - 吞吐率：单位时间完成的内存申请/释放次数
- 内存利用率
  - 内存分配器本身需要元数据保存空闲链表信息
  - 内存利用率：$\frac{totalamountofmemoryallocatedtotheapplication}{totalheapsize} < 100\%$
  - 权衡：内存利用率，分配速度和效率

- 内存碎片
  - 内部碎片
  - 外部碎片
- 空闲链表管理
  - 隐式空闲链表：思想，无显式结构记录空闲/被分配的内存块。如 何寻找空闲块（遍历整个堆：首次适应，下次适应，最佳适应/策略）
  - 边界标志法：逆向遍历块链表，常数时间合并相邻空闲块，四种情 况
  - 小结：实现简单，内存分配线性时间复杂度，内存释放常数时间复 杂度，malloc/free并不采用隐式链表管理空闲块。
  - 显式空闲链表：思想，采用显式的数据结构记录空闲块
    - 双向链表
    - 已分配的块并不需要指针记录其位置
    - 依然使用了边界标志
    - 分配和释放内存方式
    - 如何将最近被释放的插入到空闲链表中?策略? LILO，按地址 序插入
    - 小结：链接信息可以存放在payload区，因为只有空闲链表才 使用双向链表管理，而被分配的块则不需要链接信息。
  - 隔离链表
    - 思想：相同的块串接起来。小块内存块（2， 3， 4， 5， 6）各自组成空闲表，而大的内存块链表中的每个块大小值通常为2的指 数。
    - 分配策略：1，决定合适的空闲块链表，若找到，则分配，2，若没有找到，则使用sbrk向内核申请分配内存。

    * 小结：高吞吐率，内存利用率

- 内存对齐：malloc 通常返回 8 字节对齐地址。
- dlmalloc，ptmalloc,ptmalloc2,ptmalloc3
- hoard，jemalloc，google-perftools，dmalloc，nedmalloc
- Doug Lea allocator
  - 使用了边界标志，8字节对齐地址的“chunk“，空闲 chunk 有头部和尾部， 被分配的chunk 仅有头部，至少16字节（$4 + 4 + 8$）
  - Bins：小块的chunk（循环双向链表），大块的chunk（> 256），特殊的二叉树，
  - 分配算法：1，找到合适的bin；2，最佳适应（循环使 用，LRU）；
  - 释放策略
- 常见的内存问题
  - Dereferencing bad pointers
  - 读未初始化内存
  - 内存覆盖
  - 重复释放
  - ...
- 如何处理内存错误
  - GDB
  - Valgrind
  - DrMemory
  - ...

- 垃圾回收
  - 标记-清除
  - 分代垃圾回收
  - 引用计数

3.2 Problems

3.3 Solutions

3.3.1 malloc/free

Glibc中malloc/free算法分析：

对于小于 128k 的块在 heap 中分配。

1. 堆是通过 brk 的方式来增长或压缩的，如果在现有的堆中不能找到合适的 chunk ， 会 通过增长堆的方式来满足分配，如果堆顶的空闲块超过一定的阀值会收缩堆，所以只要堆顶 的空间没释放，堆是一直不会收缩的。

2. 堆中的分配信息是通过两个方式来记录。

第一．是通过 chunk 的头， chunk 中的头一个字是记录前一个 chunk 的大小，第二个字是记录当前 chunk 的大小和一些标志位，从第三个字开始是要使用的内存。所以通过内存地址可以找到 chunk ，通过 chunk 也可以找到内存地址。还可以找到相邻的下一个 chunk ，和相邻的前一个 chunk 。一个堆完全是由 n 个 chunk 组成 。

第二．是由 3 种队列记录 ，只用空闲 chunk 才会出现在队列中，使用的 chunk 不会出现在队列中。如果内存块是空闲的它会挂到其中一个队列中，它是通过内存复用的方式，使用空闲 chunk 的第 3 个字和第 4 个字当作它的前链和后链（变长块是第 5 个字和第 6 个字），省的再分配空间给它。

第一种队列是 bins ， bins 有 128 个队列，前 64 个队列是定长的，每隔 8 个字节大小的块分配在一个队列，后面的 64 个队列是不定长的，就是在一个范围长度的都分配在一个队列中。所有长度小于 512 字节（大约）的都分配在定长的队列中 。后面的 64 个队列是变长的队列，每个队列中的 chunk 都是从小到大排列的。

第二种队列是 unsort 队列（只有一个队列），（是一个缓冲）所有 free 下来的如果要进入 bins 队列中都要经过 unsort 队列。

第三种队列是 fastbins ，大约有 10 个定长队列，（是一个高速缓冲）所有 free 下来的并且长度是小于 80 的 chunk 就会进入这种队列中。进入此队列的 chunk 在 free 的时候并不修改使用位，目的是为了避免被相邻的块合并掉。

3.malloc 的步骤

1)先在 fastbins 中找，如果能找到，从队列中取下后（不需要再置使用位为 1 了）立刻返回。

2)判断需求的块是否在小箱子 （ bins 的前 64 个 bin ）范围，如果在小箱子的范围，并且刚好有需求的块，则直接返回内存地址；如果范围在大箱子（ bins 的后 64 个 bin ）里，则触发 consolidate 。（因为在大箱子找一般都要切割，所以要优先合并，避免过多碎片）

3)然后在 unsort 中取出一个 chunk ，如果能找到刚好和想要的 chunk 相同大小的 chunk ，立刻返回，如果不是想要 chunk 大小的 chunk ，就把他插入到 bins 对应的队列中去。转 3 ，直到清空，或者一次循环了 10000 次。

4)然后才在 bins 中找，找到一个最小的能符合需求的 chunk 从队列中取下，如果剩下的大小还能建一个 chunk ，就把 chunk 分成两个部分，把剩下的 chunk 插入到 unsort 队列中去，把 chunk 的内存地址返回。

5)在 topchunk （是堆顶的一个 chunk ，不会放到任何一个队列里的）找，如果能切出符合要求的，把剩下的一部分当作 topchunk ，然后返回内存地址。

6)如果 fastbins 不为空，触发 consolidate 即把所有的 fanbins 清空 （是把 fanbins 的使用位置 0 ，把相邻的块合并起来，然后挂到 unsort 队列中去），然后继续第 3 步。

7)还找不到话就调用 sysalloc ，其实就是增长堆了。然后返回内存地址。

4.free 的步骤

1)如果和 topchunk 相邻，直接和 topchunk 合并，不会放到其他的空闲队列中去。

2)如果释放的大小小于 80 字节，就把它挂到 fastbins 中去，使用位仍然为 1 ，当然更不会去合并相邻块。

3)如果释放块大小介于 80-128k ，把 chunk 的使用位置成 0 ，然后试图合并相邻块，挂到 unsort 队列中去，如果合并后的大小大于 64k ，也会触发 consolidate ，（可能是周围比较多小块了吧），然后才试图去收缩堆。（收缩堆的条件是当前 free 的块大小加上前后能合并 chunk 的大小大于 64k ，并且要堆顶的大小要达到阀值，才有可能收缩堆）

4)对于大于 128k 的块，直接 munmap

二．大于 128k 的块通过 mmap 的方式来分配。

3.3.2 GC

引用计数（ Reference Counting ）算法

1960 年以前，人们为胚胎中的 Lisp 语言设计垃圾收集机制时，第一个想到的算法是引用计数算法。拿餐巾纸的例子来说，这种算法的原理大致可以描述为：

午餐时，为了把脑子里突然跳出来的设计灵感记下来，我从餐巾纸袋中抽出一张餐巾纸，打算在上面画出系统架构的蓝图。按照“餐巾纸使用规约之引用计数版”的要求，画图之前，我必须先在餐巾纸的一角写上计数值 1 ，以表示我在使用这张餐巾纸。这时，如果你也想看看我画的蓝图，那你

就要把餐巾纸上的计数值加 1 ，将它改为 2 ，这表明目前有 2 个人在同时使用这张餐巾纸（当然，我是不会允许你用这张餐巾纸来擦鼻涕的）。你看完后，必须把计数值减 1 ，表明你对该餐巾纸的使用已经结束。同样，当我将餐巾纸上的内容全部誊写到笔记本上之后，我也会自觉地把餐巾纸上的计数值减 1 。此时，不出意外的话，这张餐巾纸上的计数值应当是 0 ，它会被垃圾收集器——假设那是一个专门负责打扫卫生的机器人——捡起来扔到垃圾箱里，因为垃圾收集器的惟一使命就是找到所有计数值为 0 的餐巾纸并清理它们。

引用计数算法的优点和缺陷同样明显。这一算法在执行垃圾收集任务时速度较快，但算法对程序中每一次内存分配和指针操作提出了额外的要求（增加或减少内存块的引用计数）。更重要的是，引用计数算法无法正确释放循环引用的内存块，对此， D. Hillis 有一段风趣而精辟的论述：

一天，一个学生走到 Moon 面前说：“我知道如何设计一个更好的垃圾收集器了。我们必须记录指向每个结点的指针数目。” Moon 耐心地给这位学生讲了下面这个故事：“一天，一个学生走到 Moon 面前说：‘我知道如何设计一个更好的垃圾收集器了……’”

D. Hillis 的故事和我们小时候常说的“从前有座山，山上有个庙，庙里有个老和尚”的故事有异曲同工之妙。这说明，单是使用引用计数算法还不足以解决垃圾收集中的所有问题。正因为如此，引用计数算法也常常被研究者们排除在狭义的垃圾收集算法之外。当然，作为一种最简单、最直观的解决方案，引用计数算法本身具有其不可替代的优越性。 1980 年代前后， D. P. Friedman ， D. S. Wise ， H. G. Baker 等人对引用计数算法进行了数次改进，这些改进使得引用计数算法及其变种（如延迟计数算法等）在简单的环境下，或是在一些综合了多种算法的现代垃圾收集系统中仍然可以一展身手。

标记－清除（ Mark-Sweep ）算法

第一种实用和完善的垃圾收集算法是 J. McCarthy 等人在 1960 年提出并成功地应用于 Lisp 语言的标记－清除算法。仍以餐巾纸为例，标记－清

除算法的执行过程是这样的:

午餐过程中，餐厅里的所有人都根据自己的需要取用餐巾纸。当垃圾收集机器人想收集废旧餐巾纸的时候，它会让所有用餐的人先停下来，然后，依次询问餐厅里的每一个人："你正在用餐巾纸吗？你用的是哪一张餐巾纸？"机器人根据每个人的回答将人们正在使用的餐巾纸画上记号。询问过程结束后，机器人在餐厅里寻找所有散落在餐桌上且没有记号的餐巾纸（这些显然都是用过的废旧餐巾纸），把它们统统扔到垃圾箱里。

正如其名称所暗示的那样，标记－清除算法的执行过程分为"标记"和"清除"两大阶段。这种分步执行的思路奠定了现代垃圾收集算法的思想基础。与引用计数算法不同的是，标记－清除算法不需要运行环境监测每一次内存分配和指针操作，而只要在"标记"阶段中跟踪每一个指针变量的指向——用类似思路实现的垃圾收集器也常被后人统称为跟踪收集器（Tracing Collector）

伴随着 Lisp 语言的成功，标记－清除算法也在大多数早期的 Lisp 运行环境中大放异彩。尽管最初版本的标记－清除算法在今天看来还存在效率不高（标记和清除是两个相当耗时的过程）等诸多缺陷，但在后面的讨论中，我们可以看到，几乎所有现代垃圾收集算法都是标记－清除思想的延续，仅此一点， J. McCarthy 等人在垃圾收集技术方面的贡献就丝毫不亚于他们在 Lisp 语言上的成就了。

复制（Copying）算法

为了解决标记－清除算法在垃圾收集效率方面的缺陷， M. L. Minsky 于 1963 年发表了著名的论文"一种使用双存储区的 Lisp 语言垃圾收集器（A LISP Garbage Collector Algorithm Using Serial Secondary Storage）"。M. L. Minsky 在该论文中描述的算法被人们称为复制算法，它也被 M. L. Minsky 本人成功地引入到了 Lisp 语言的一个实现版本中。

复制算法别出心裁地将堆空间一分为二，并使用简单的复制操作来完成垃圾收集工作，这个思路相当有趣。借用餐巾纸的比喻，我们可以这样理解 M. L. Minsky 的复制算法:

餐厅被垃圾收集机器人分成南区和北区两个大小完全相同的部分。午餐时，所有人都先在南区用餐（因为空间有限，用餐人数自然也将减少一半），用餐时可以随意使用餐巾纸。当垃圾收集机器人认为有必要回收废旧餐巾纸时，它会要求所有用餐者以最快的速度从南区转移到北区，同时随身携带自己正在使用的餐巾纸。等所有人都转移到北区之后，垃圾收集机器人只要简单地把南区中所有散落的餐巾纸扔进垃圾箱就算完成任务了。下一次垃圾收集的工作过程也大致类似，惟一的不同只是人们的转移方向变成了从北区到南区。如此循环往复，每次垃圾收集都只需简单地转移（也就是复制）一次，垃圾收集速度无与伦比——当然，对于用餐者往返奔波于南北两区之间的辛劳，垃圾收集机器人是决不会流露出丝毫怜悯的。

M. L. Minsky 的发明绝对算得上一种奇思妙想。分区、复制的思路不仅大幅提高了垃圾收集的效率，而且也将原本繁纷复杂的内存分配算法变得前所未有地简明和扼要（既然每次内存回收都是对整个半区的回收，内存分配时也就不用考虑内存碎片等复杂情况，只要移动堆顶指针，按顺序分配内存就可以了），这简直是个奇迹！不过，任何奇迹的出现都有一定的代价，在垃圾收集技术中，复制算法提高效率的代价是人为地将可用内存缩小了一半。实话实说，这个代价未免也太高了一些。

标记－整理（ Mark-Compact ）算法

标记－整理算法是标记－清除算法和复制算法的有机结合。把标记－清除算法在内存占用上的优点和复制算法在执行效率上的特长综合起来，这是所有人都希望看到的结果。不过，两种垃圾收集算法的整合并不像 1 加 1 等于 2 那样简单，我们必须引入一些全新的思路。 1970 年前后， G. L. Steele ， C. J. Cheney 和 D. S. Wise 等研究者陆续找到了正确的方向，标记－整理算法的轮廓也逐渐清晰了起来：

在我们熟悉的餐厅里，这一次，垃圾收集机器人不再把餐厅分成两个南北区域了。需要执行垃圾收集任务时，机器人先执行标记－清除算法的第一个步骤，为所有使用中的餐巾纸画好标记，然后，机器人命令所有就餐

者带上有标记的餐巾纸向餐厅的南面集中，同时把没有标记的废旧餐巾纸扔向餐厅北面。这样一来，机器人只消站在餐厅北面，怀抱垃圾箱，迎接扑面而来的废旧餐巾纸就行了。

实验表明，标记－整理算法的总体执行效率高于标记－清除算法，又不像复制算法那样需要牺牲一半的存储空间，这显然是一种非常理想的结果。在许多现代的垃圾收集器中，人们都使用了标记－整理算法或其改进版本。

增量收集（ Incremental Collecting ）算法

对实时垃圾收集算法的研究直接导致了增量收集算法的诞生。

最初，人们关于实时垃圾收集的想法是这样的：为了进行实时的垃圾收集，可以设计一个多进程的运行环境，比如用一个进程执行垃圾收集工作，另一个进程执行程序代码。这样一来，垃圾收集工作看上去就仿佛是在后台悄悄完成的，不会打断程序代码的运行。

在收集餐巾纸的例子中，这一思路可以被理解为：垃圾收集机器人在人们用餐的同时寻找废弃的餐巾纸并将它们扔到垃圾箱里。这个看似简单的思路会在设计和实现时碰上进程间冲突的难题。比如说，如果垃圾收集进程包括标记和清除两个工作阶段，那么，垃圾收集器在第一阶段中辛辛苦苦标记出的结果很可能被另一个进程中的内存操作代码修改得面目全非，以至于第二阶段的工作没有办法开展。

M. L. Minsky 和 D. E. Knuth 对实时垃圾收集过程中的技术难点进行了早期的研究， G. L. Steele 于 1975 年发表了题为“多进程整理的垃圾收集（ Multiprocessing compactifying garbage collection ）”的论文，描述了一种被后人称为“ Minsky-Knuth-Steele 算法”的实时垃圾收集算法。 E. W. Dijkstra ， L. Lamport ， R. R. Fenichel 和 J. C. Yochelson 等人也相继在此领域做出了各自的贡献。 1978 年， H. G. Baker 发表了“串行计算机上的实时表处理技术（ List Processing in Real Time on a Serial Computer ）”一文，系统阐述了多进程环境下用于垃圾收集的增量收集算法。

增量收集算法的基础仍是传统的标记－清除和复制算法。增量收集算法

通过对进程间冲突的妥善处理，允许垃圾收集进程以分阶段的方式完成标记、清理或复制工作。详细分析各种增量收集算法的内部机理是一件相当繁琐的事情，在这里，读者们需要了解的仅仅是： H. G. Baker 等人的努力已经将实时垃圾收集的梦想变成了现实，我们再也不用为垃圾收集打断程序的运行而烦恼了

分代收集（ Generational Collecting ）算法

和大多数软件开发技术一样，统计学原理总能在技术发展的过程中起到强力催化剂的作用。 1980 年前后，善于在研究中使用统计分析知识的技术人员发现，大多数内存块的生存周期都比较短，垃圾收集器应当把更多的精力放在检查和清理新分配的内存块上。这个发现对于垃圾收集技术的价值可以用餐巾纸的例子概括如下:

如果垃圾收集机器人足够聪明，事先摸清了餐厅里每个人在用餐时使用餐巾纸的习惯——比如有些人喜欢在用餐前后各用掉一张餐巾纸，有的人喜欢自始至终攥着一张餐巾纸不放，有的人则每打一个喷嚏就用去一张餐巾纸——机器人就可以制定出更完善的餐巾纸回收计划，并总是在人们刚扔掉餐巾纸没多久就把垃圾捡走。这种基于统计学原理的做法当然可以让餐厅的整洁度成倍提高。

D. E. Knuth ， T. Knight ， G. Sussman 和 R. Stallman 等人对内存垃圾的分类处理做了最早的研究。 1983 年， H. Lieberman 和 C. Hewitt 发表了题为“基于对象寿命的一种实时垃圾收集器（ A real-time garbage collector based on the lifetimes of objects ）”的论文。这篇著名的论文标志着分代收集算法的正式诞生。此后，在 H. G. Baker ， R. L. Hudson ， J. E. B. Moss 等人的共同努力下，分代收集算法逐渐成为了垃圾收集领域里的主流技术。

分代收集算法通常将堆中的内存块按寿命分为两类，年老的和年轻的。垃圾收集器使用不同的收集算法或收集策略，分别处理这两类内存块，并特别地把主要工作时间花在处理年轻的内存块上。分代收集算法使垃圾收集器在有限的资源条件下，可以更为有效地工作——这种效率上的提高在

今天的 Java 虚拟机中得到了最好的证明。

4 Conclusion