Cliff Notes for My Presentation

Slide 3: Dynamic Storage Allocation

- dynamic storage allocation is allocation of memory at runtime
- important because we do not always know ahead of time how much space we'll need
- in languages such as C and C++, when we dynamically allocate memory, the programmer must also explicitly deallocate the memory
- not deallocating memory properly is a common source of programming error
- two most common programming errors are dangling references and memory leaks

Slide 4: Problems with Explicit Memory Deallocation

- dangling references occur when the programmer deallocates memory too soon
- in the example, **`pi`** is deleted before all references to it have been removed
- when we try to access the value pointed to by **`q`** (which consequently was the value pointed to by **`pi`**), it no longer exists

Slide 5: Problems with Explicit Memory Deallocation

- a memory leak occurs when the programmer forgets to delete dynamically allocated memory
- in this case, the memory can no longer be returned to the heap
- in the example, you can see that the programmer requested memory at the beginning of the function but never returned it to the heap
- both dangling references and memory leaks can cause a lot of havoc in a program

Slide 6: Solving the Problem

- to avoid dealing with the problems of explicit memory deallocation, languages such as lisp and java use a "garbage collector"
- the garbage collector reclaims objects that are no longer reachable and copies the survivors to another area of memory
- these dead objects are referred to as garbage

Slide 7: Beltway

- this summer my project dealt with optimizing the garbage collector (gc) that Steve, Kathryn, and others created

- this gc is called Beltway and follows the following 5 key ideas of copying gc
  1. "most objects die young"
     - statistics show that between 80-95% of objects die before the next megabyte of heap has been allocated
  2. "give objects time to die"
     - we do not want to collect the youngest objects because they are likely to still be alive
  3. "Avoid collecting old objects"
     - not as likely to be dead
  4. "incrementality improves responsiveness"
     - breaking down amount that needs to be collected at one time increases response time
  5. "copying gc can improve locality"
     - reduces amount of fragmentation
     - when survivors are copied, they are copied contiguously in memory

Slide 8: Organizational Terms

- before looking at an example of Beltway, there are two terms you must know: belts and increments
- the beltway collector is made up of what we call "belts"
- and the belts are made up of "increments" (which are independently collectible regions of memory)
- both belts and increments can be collected independently

Slide 9: A Simple Example
Slide 10: A More Interesting Example

Slide 11: So What's the Problem?

- you might be asking yourself this very question – so what's the problem?
- the beltway examples I showed you follow the 5 principles of copying gc and the beltway gc seems to be a great new collector
- unfortunately, Steve and Kathryn found they could not get FIFO in the nursery without a significant performance hit
- this is the problem we tried to solve this summer
- to understand the reasons behind this problem, we must first look at how pointers are handled in beltway

Slide 12: Pointers and Write Barriers

- pointers whose *source* and *target* are in the same increment are called "intra-increment" pointers and are ignored
- pointers whose *source* and *target* are in different increments need only be remembered if the *target* could be collected before the *source*

- these are old-young pointers (pointers going from an object in an older increment to an object in a younger increment)
- pointers are tested by a write barrier
- the write barrier also has the function of remembering old-young pointers
- to understand how a write barrier works, we need to look at a diagram of a heap

Slide 13: What are Write Barriers?

- if we implement two write barriers (the gray lines), the heap is divided into 3 increments
- if we say that the leftmost increment is the youngest increment (meaning we collect from left to right), we now need to remember only those pointers that cross a barrier from right to left
- this eliminates the need to remember all pointers in the heap
- pointers "to remember" in grey (left) are the pointers the write barrier would have to remember if we were collecting increment 1
- pointers "to remember" in red (middle) are the pointers the write barriers would have to remember if we were collecting increment 2
- pointers "to remember" in gray (right) are the pointers the write barrier would have to remember if we were collecting increment 3
- write barrier benefits us by: giving us better performance, targeted collection, and shorter pause times (less time spent in gc)
- the write barrier sounds like a good idea but it does not come without its cost

Slide 14: Overhead of the Write Barrier

- overhead of the write barrier comes in two forms:
- fast path: does the pointer cross a write barrier?
- slow path: if so, does the pointer need to be remembered?
- as stated, the fast path is cheap (we use a mask and XOR for this) and the slow path is expensive
- now getting back to the problem of the performance hit from a FIFO nursery

Slide 15: Motivation for Optimization

- Steve and Kathryn found that the expensive slow path was being evaluated a lot
- having a FIFO nursery requires many look-ups to determine whether a pointer needs to be remembered
- the quick solution Kathryn and Steve found was to remove the FIFO behavior from the nursery
- to remove the FIFO behavior, two rules needed to be implemented: there could only be a single nursery and it always had to be collected first

- however this made beltway more specific case and Kathryn wanted the gc to be more general
- also without a FIFO nursery, they could not reap the benefits of OF (the idea that we should “give objects time to die”)

Slide 16: Why was Slow Path Evaluated A lot?

- the final clue to the performance hit puzzle was to determine why the slow path was being evaluated so much
- Steve and Kathryn hypothesized that it was because of pointer to TIBs
- TIB stands for “type information block”
- Important things to know about TIBs:
    - created when a class is loaded
    - declared as an array of object references and allocated into the nursery (just like all other objects)
    - stores information about the class such as methods, virtual methods, etc
    - every object has a pointer to the TIB in its header

Slide 17: The TIB Write Barrier

- because pointers to the TIBs are in the object header, they are not seen by the normal write barrier
- Steve and Kathryn implemented a TIB write to remember pointers to TIBs if necessary
- however the TIB write barriers account for most of write barrier activity

  AND

  furthermore, it is never necessary to remember pointers to TIBs
- so if we removed the TIB write barrier we thought we could decrease the number of slow path evaluations, thus solving the problem of the performance hit in the nursery
- but we can only remove w/b if we move all TIBs to a “special” place in memory

Slide 18: Optimizations for Beltway

- Here’s what we did
- we implemented an immortal space for TIBs to live for the duration of the program
- the immortal space will never be collected
- turned off the TIB write barrier

Slide 19: A Simple Case First

- but before we could implement these optimizations, we wanted to look at a simpler case to get an idea of how they would affect performance

- so we implemented these changes in an Appel generational copying collector

Slide 20: Appel Generational Copying GC

- objects are grouped into 2 generations according to their “age”
- age is determined by the amount of the heap allocated since the object was created
- generation 0 is called the nursery and generation 1 is called the older generation
- objects are allocated into the nursery
- when the nursery fills, we collect it and promote survivors to the older generation
- we continue allocation into the nursery and the cycle continues
- when the older generation fills, we collect both the older generation and the nursery
- pointers are handled as in beltway (i.e. intra-generational pointers are ignored and old-young intergenerational pointers must be remembered)

Slide 21: Appel Facts to Remember

- nursery is always collected before the older generation
- collection is not FIFO
- nursery is collected most often following the hypothesis that “most objects die young”
- note that this hypothesis is one of the principles of copying gc

Slide 22: Optimizations for Appel

- just like our optimization for Beltway, in Appel we created an immortal space for TIBs and removed the tib write barrier
- the only difference with the tib write barrier is that in Appel the tib write barrier only has a fast path
- this fast path does both tests (does a pointer cross a boundary and if so does it need to be remembered)

Slide 23/24: Results for Appel

- Notice that removing the TIB write barrier (green line and aqua line) causes a significant decrease in runtime (as much as 10% in some places) as compared to Appel without the optimizations
- putting the TIBs in an immortal space does not have much affect on the run-time
- javac: Sun JDK Java compiler compiling jess (another benchmark which is a system shell)
- pseudojbb: emulates a 3-tier transaction processing system

Slide 25: Results for Beltway

- we do not have any results for beltway yet (they are running on the computer as we speak)
- however based on the results from Appel, we expect that removing the TIB write barrier will lower the run-time (probably more so than in Appel)
- we also do not believe that putting the TIBs into the immortal space will have much of an effect on run-time (as in Appel)