7.7.5 The Train Algorithm

While the generational approach is very efficient for the handling of immature objects, it is less efficient for the mature objects, since mature objects are moved every time there is a collection involving them, and they are quite unlikely to be garbage. A different approach to incremental collection, called the *train algorithm*, was developed to improve the handling of mature objects. It can be used for collecting all garbage, but it is probably better to use the generational approach for immature objects and, only after they have survived a few rounds of collection, "promote" them to another heap, managed by the train algorithm. Another advantage to the train algorithm is that we never have to do a complete garbage collection, as we do occasionally for generational garbage collection.

To motivate the train algorithm, let us look at a simple example of why it is necessary, in the generational approach, to have occasional all-inclusive rounds of garbage collection. Figure 7.29 shows two mutually linked objects in two partitions i and j, where $j > i$. Since both objects have pointers from outside their partition, a collection of only partition i or only partition j could never collect either of these objects. Yet they may in fact be part of a cyclic garbage structure with no links from the outside. In general, the "links" between the objects shown may involve many objects and long chains of references.

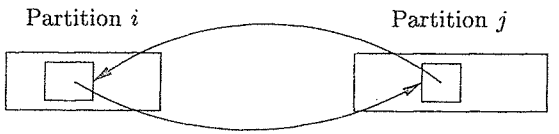

Figure 7.29: A cyclic structure across partitions that may be cyclic garbage

In generational garbage collection, we eventually collect partition j, and since $i < j$, we also collect i at that time. Then, the cyclic structure will be completely contained in the portion of the heap being collected, and we can tell if it truly is garbage. However, if we never have a round of collection that includes both i and j, we would have a problem with cyclic garbage, just as we did with reference counting for garbage collection.

The train algorithm uses fixed-length partitions, called *cars*; a car might be a single disk block, provided there are no objects larger than disk blocks, or the car size could be larger, but it is fixed once and for all. Cars are organized into *trains*. There is no limit to the number of cars in a train, and no limit to the number of trains. There is a lexicographic order to cars: first order by train number, and within a train, order by car number, as in Fig. 7.30.

There are two ways that garbage is collected by the train algorithm:

- The first car in lexicographic order (that is, the first remaining car of the first remaining train) is collected in one incremental garbage-collection step. This step is similar to collection of the first partition in the generational algorithm, since we maintain a "remembered" list of all pointers

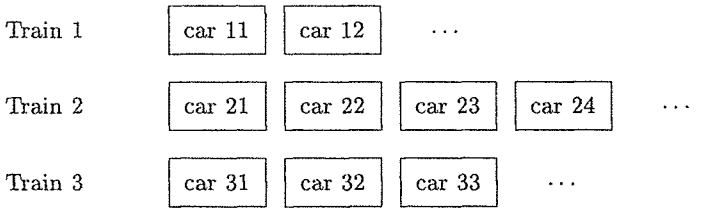

Figure 7.30: Organization of the heap for the train algorithm

from outside the car. Here, we identify objects with no references at all, as well as garbage cycles that are contained completely within this car. Reachable objects in the car are always moved to some other car, so each garbage-collected car becomes empty and can be removed from the train.

- Sometimes, the first train has no external references. That is, there are no pointers from the root set to any car of the train, and the remembered sets for the cars contain only references from other cars in the train, not from other trains. In this situation, the train is a huge collection of cyclic garbage, and we delete the entire train.

Remembered Sets

We now give the details of the train algorithm. Each car has a remembered set consisting of all references to objects in the car from

a) Objects in higher-numbered cars of the same train, and

b) Objects in higher-numbered trains.

In addition, each train has a remembered set consisting of all references from higher-numbered trains. That is, the remembered set for a train is the union of the remembered sets for its cars, except for those references that are internal to the train. It is thus possible to represent both kinds of remembered sets by dividing the remembered sets for the cars into "internal" (same train) and "external" (other trains) portions.

Note that references to objects can come from anywhere, not just from lexicographically higher cars. However, the two garbage-collection processes deal with the first car of the first train, and the entire first train, respectively. Thus, when it is time to use the remembered sets in a garbage collection, there is nothing earlier from which references could come, and therefore there is no point in remembering references to higher cars at any time. We must be careful, of course, to manage the remembered sets properly, changing them whenever the mutator modifies references in any object.

Managing Trains

Our objective is to draw out of the first train all objects that are not cyclic garbage. Then, the first train either becomes nothing but cyclic garbage and is therefore collected at the next round of garbage collection, or if the garbage is not cyclic, then its cars may be collected one at a time.

We therefore need to start new trains occasionally, even though there is no limit on the number of cars in one train, and we could in principle simply add new cars to a single train, every time we needed more space. For example, we could start a new train after every k object creations, for some k. That is, in general, a new object is placed in the last car of the last train, if there is room, or in a new car that is added to the end of the last train, if there is no room. However, periodically, we instead start a new train with one car, and place the new object there.

Garbage Collecting a Car

The heart of the train algorithm is how we process the first car of the first train during a round of garbage collection. Initially, the reachable set is taken to be the objects of that car with references from the root set and those with references in the remembered set for that car. We then scan these objects as in a mark-and-sweep collector, but we do not scan any reached objects outside the one car being collected. After this tracing, some objects in the car may be identified as garbage. There is no need to reclaim their space, because the entire car is going to disappear anyway.

However, there are likely to be some reachable objects in the car, and these must be moved somewhere else. The rules for moving an object are:

- If there is a reference in the remembered set from any other train (which will be higher-numbered than the train of the car being collected), then move the object to one of those trains. If there is room, the object can go in some existing car of the train from which a reference emanates, or it can go in a new, last car if there is no room.

- If there is no reference from other trains, but there are references from the root set or from the first train, then move the object to any other car of the same train, creating a new, last car if there is no room. If possible, pick a car from which there is a reference, to help bring cyclic structures to a single car.

After moving all the reachable objects from the first car, we delete that car.

Panic Mode

There is one problem with the rules above. In order to be sure that all garbage will eventually be collected, we need to be sure that every train eventually becomes the first train, and if this train is not cyclic garbage, then eventually

all cars of that train are removed and the train disappears one car at a tim However, by rule (2) above, collecting the first car of the first train can produc a new last car. It cannot produce two or more new cars, since surely all th objects of the first car can fit in the new, last car. However, could we be in situation where each collection step for a train results in a new car being adde and we never get finished with this train and move on to the other trains?

The answer is, unfortunately, that such a situation is possible. The proble arises if we have a large, cyclic, nongarbage structure, and the mutator manag to change references in such a way that we never see, at the time we colle a car, any references from higher trains in the remembered set. If even or object is removed from the train during the collection of a car, then we are OI since no new objects are added to the first train, and therefore the first trai will surely run out of objects eventually. However, there may be no garbag at all that we can collect at a stage, and we run the risk of a loop where w perpetually garbage collect only the current first train.

To avoid this problem, we need to behave differently whenever we encount a *futile* garbage collection, that is, a car from which not even one object can b deleted as garbage or moved to another train. In this "panic mode," we mak two changes:

1. When a reference to an object in the first train is rewritten, we maintai the reference as a new member of the root set.

2. When garbage collecting, if an object in the first car has a reference fro the root set, including dummy references set up by point (1), then w move that object to another train, even if it has no references from oth trains. It is not important which train we move it to, as long as it is nc the first train.

In this way, if there are any references from outside the first train to objec in the first train, these references are considered as we collect every car, an eventually some object will be removed from that train. We can then leave pan mode and proceed normally, sure that the current first train is now smaller tha it was.

7.7.6 Exercises for Section 7.7

Exercise 7.7.1: Suppose that the network of objects from Fig. 7.20 is manage by an incremental algorithm that uses the four lists *Unreached*, *Unscanne Scanned*, and *Free*, as in Baker's algorithm. To be specific, the *Unscanned* li is managed as a queue, and when more than one object is to be placed on this li due to the scanning of one object, we do so in alphabetical order. Suppose als that we use write barriers to assure that no reachable object is made garbag Starting with A and B on the *Unscanned* list, suppose the following even occur:

i. A is scanned.

ii. The pointer $A \rightarrow D$ is rewritten to be $A \rightarrow H$.

iii. B is scanned.

iv. D is scanned.

v. The pointer $B \rightarrow C$ is rewritten to be $B \rightarrow I$.

Simulate the entire incremental garbage collection, assuming no more pointers are rewritten. Which objects are garbage? Which objects are placed on the *Free* list?

Exercise 7.7.2: Repeat Exercise 7.7.1 on the assumption that

a) Events (*ii*) and (*v*) are interchanged in order.

b) Events (*ii*) and (*v*) occur before (*i*), (*iii*), and (*iv*).

Exercise 7.7.3: Suppose the heap consists of exactly the nine cars on three trains shown in Fig. 7.30 (i.e., ignore the ellipses). Object o in car 11 has references from cars 12, 23, and 32. When we garbage collect car 11, where might o wind up?

Exercise 7.7.4: Repeat Exercise 7.7.3 for the cases that o has

a) Only references from cars 22 and 31.

b) No references other than from car 11.

Exercise 7.7.5: Suppose the heap consists of exactly the nine cars on three trains shown in Fig. 7.30 (i.e., ignore the ellipses). We are currently in panic mode. Object o_1 in car 11 has only one reference, from object o_2 in car 12. That reference is rewritten. When we garbage collect car 11, what could happen to o_1?

7.8 Advanced Topics in Garbage Collection

We close our investigation of garbage collection with brief treatments of four additional topics:

1. Garbage collection in parallel environments.
2. Partial relocations of objects.
3. Garbage collection for languages that are not type-safe.
4. The interaction between programmer-controlled and automatic garbage collection.

7.8.1 Parallel and Concurrent Garbage Collection

Garbage collection becomes even more challenging when applied to applications running in parallel on a multiprocessor machine. It is not uncommon for server applications to have thousands of threads running at the same time; each of these threads is a mutator. Typically, the heap will consist of gigabytes of memory.

Scalable garbage-collection algorithms must take advantage of the presence of multiple processors. We say a garbage collector is *parallel* if it uses multiple threads; it is *concurrent* if it runs simultaneously with the mutator.

We shall describe a parallel, and mostly concurrent, collector that uses a concurrent and parallel phase that does most of the tracing work, and then a stop-the-world phase that guarantees all the reachable objects are found and reclaims the storage. This algorithm introduces no new basic concepts in garbage collection per se; it shows how we can combine the ideas described so far to create a full solution to the parallel-and-concurrent collection problem. However, there are some new implementation issues that arise due to the nature of parallel execution. We shall discuss how this algorithm coordinates multiple threads in a parallel computation using a rather common work-queue model.

To understand the design of the algorithm we must keep in mind the scale of the problem. Even the root set of a parallel application is much larger, consisting of every thread's stack, register set and globally accessible variables. The amount of heap storage can be very large, and so is the amount of reachable data. The rate at which mutations take place is also much greater.

To reduce the pause time, we can adapt the basic ideas developed for incremental analysis to overlap garbage collection with mutation. Recall that an incremental analysis, as discussed in Section 7.7, performs the following three steps:

1. Find the root set. This step is normally performed atomically, that is, with the mutator(s) stopped.
2. Interleave the tracing of the reachable objects with the execution of the mutator(s). In this period, every time a mutator writes a reference that points from a *Scanned* object to an *Unreached* object, we remember that reference. As discussed in Section 7.7.2, we have options regarding the granularity with which these references are remembered. In this section, we shall assume the card-based scheme, where we divide the heap into sections called "cards" and maintain a bit map indicating which cards are *dirty* (have had one or more references within them rewritten).
3. Stop the mutator(s) again to rescan all the cards that may hold references to unreached objects.

For a large multithreaded application, the set of objects reached by the root set can be very large. It is infeasible to take the time and space to visit all such objects while all mutations cease. Also, due to the large heap and the large

number of mutation threads, many cards may need to be rescanned after all objects have been scanned once. It is thus advisable to scan some of these cards in parallel, while the mutators are allowed to continue to execute concurrently.

To implement the tracing of step (2) above, in parallel, we shall use multiple garbage-collecting threads concurrently with the mutator threads to trace *most* of the reachable objects. Then, to implement step (3), we stop the mutators and use parallel threads to ensure that all reachable objects are found.

The tracing of step (2) is carried out by having each mutator thread perform part of the garbage collection, along with its own work. In addition, we use threads that are dedicated purely to collecting garbage. Once garbage collection has been initiated, whenever a mutator thread performs some memory-allocation operation, it also performs some tracing computation. The pure garbage-collecting threads are put to use only when a machine has idle cycles. As in incremental analysis, whenever a mutator writes a reference that points from a *Scanned* object to an *Unreached* object, the card that holds this reference is marked dirty and needs to be rescanned.

Here is an outline of the parallel, concurrent garbage-collection algorithm.

1. Scan the root set for each mutator thread, and put all objects directly reachable from that thread into the *Unscanned* state. The simplest incremental approach to this step is to wait until a mutator thread calls the memory manager, and have it scan its own root set if that has not already been done. If some mutator thread has not called a memory allocation function, but all the rest of tracing is done, then this thread must be interrupted to have its root set scanned.

2. Scan objects that are in the *Unscanned* state. To support parallel computation, we use a work queue of fixed-size *work packets*, each of which holds a number of *Unscanned* objects. *Unscanned* objects are placed in work packets as they are discovered. Threads looking for work will dequeue these work packets and trace the *Unscanned* objects therein. This strategy allows the work to be spread evenly among workers in the tracing process. If the system runs out of space, and we cannot find the space to create these work packets, we simply mark the cards holding the objects to force them to be scanned. The latter is always possible because the bit array holding the marks for the cards has already been allocated.

3. Scan the objects in dirty cards. When there are no more *Unscanned* objects left in the work queue, and all threads' root sets have been scanned, the cards are rescanned for reachable objects. As long as the mutators continue to execute, dirty cards continue to be produced. Thus, we need to stop the tracing process using some criterion, such as allowing cards to be rescanned only once or a fixed number of times, or when the number of outstanding cards is reduced to some threshold. As a result, this parallel and concurrent step normally terminates before completing the trace, which is finished by the final step, below.

4. The final step guarantees that all reachable objects are marked as reached. With all the mutators stopped, the root sets for all the threads can now be found quickly using all the processors in the system. Because the reachability of most objects has been traced, only a small number of objects are expected to be placed in the *Unscanned* state. All the threads then participate in tracing the rest of the reachable objects and rescanning all the cards.

It is important that we control the rate at which tracing takes place. The tracing phase is like a race. The mutators create new objects and new references that must be scanned, and the tracing tries to scan all the reachable objects and rescan the dirty cards generated in the meanwhile. It is not desirable to start the tracing too much before a garbage collection is needed, because that will increase the amount of floating garbage. On the other hand, we cannot wait until the memory is exhausted before the tracing starts, because then mutators will not be able to make forward progress and the situation degenerates to that of a stop-the-world collector. Thus, the algorithm must choose the time to commence the collection and the rate of tracing appropriately. An estimate of the mutation rate from previous cycles of collection can be used to help in the decision. The tracing rate is dynamically adjusted to account for the work performed by the pure garbage-collecting threads.

7.8.2 Partial Object Relocation

As discussed starting in Section 7.6.4, copying or compacting collectors are advantageous because they eliminate fragmentation. However, these collectors have nontrivial overheads. A compacting collector requires moving all objects and updating all the references at the end of garbage collection. A copying collector figures out where the reachable objects go as tracing proceeds; if tracing is performed incrementally, we need either to translate a mutator's every reference, or to move all the objects and update their references at the end. Both options are very expensive, especially for a large heap.

We can instead use a copying generational garbage collector. It is effective in collecting immature objects and reducing fragmentation, but can be expensive when collecting mature objects. We can use the train algorithm to limit the amount of mature data analyzed each time. However, the overhead of the train algorithm is sensitive to the size of the remembered set for each partition.

There is a hybrid collection scheme that uses concurrent tracing to reclaim all the unreachable objects and at the same time moves only a part of the objects. This method reduces fragmentation without incurring the full cost of relocation in each collection cycle.

1. Before tracing begins, choose a part of the heap that will be evacuated.

2. As the reachable objects are marked, also remember all the references pointing to objects in the designated area.

3. When tracing is complete, sweep the storage in parallel to reclaim the space occupied by unreachable objects.

4. Finally, evacuate the reachable objects occupying the designated area and fix up the references to the evacuated objects.

7.8.3 Conservative Collection for Unsafe Languages

As discussed in Section 7.5.1, it is impossible to build a garbage collector that is guaranteed to work for all C and C++ programs. Since we can always compute an address with arithmetic operations, no memory locations in C and C++ can ever be shown to be unreachable. However, many C or C++ programs never fabricate addresses in this way. It has been demonstrated that a conservative garbage collector — one that does not necessarily discard all garbage — can be built to work well in practice for this class of programs.

A conservative garbage collector assumes that we cannot fabricate an address, or derive the address of an allocated chunk of memory without an address pointing somewhere in the same chunk. We can find all the garbage in programs satisfying such an assumption by treating as a valid address any bit pattern found anywhere in reachable memory, as long as that bit pattern may be construed as a memory location. This scheme may classify some data erroneously as addresses. It is correct, however, since it only causes the collector to be conservative and keep more data than necessary.

Object relocation, requiring all references to the old locations be updated to point to the new locations, is incompatible with conservative garbage collection. Since a conservative garbage collector does not know if a particular bit pattern refers to an actual address, it cannot change these patterns to point to new addresses.

Here is how a conservative garbage collector works. First, the memory manager is modified to keep a *data map* of all the allocated chunks of memory. This map allows us to find easily the starting and ending boundary of the chunk of memory that spans a certain address. The tracing starts by scanning the program's root set to find any bit pattern that looks like a memory location, without worrying about its type. By looking up these potential addresses in the data map, we can find the starting addresses of those chunks of memory that might be reached, and place them in the *Unscanned* state. We then scan all the unscanned chunks, find more (presumably) reachable chunks of memory, and place them on the work list until the work list becomes empty. After tracing is done, we sweep through the heap storage using the data map to locate and free all the unreachable chunks of memory.

7.8.4 Weak References

Sometimes, programmers use a language with garbage collection, but also wish to manage memory, or parts of memory, themselves. That is, a programmer may know that certain objects are never going to be accessed again, even though

references to the objects remain. An example from compiling will suggest the problem.

Example 7.17: We have seen that the lexical analyzer often manages a symbol table by creating an object for each identifier it sees. These objects may appear as lexical values attached to leaves of the parse tree representing those identifiers, for instance. However, it is also useful to create a hash table, keyed by the identifier's string, to locate these objects. That table makes it easier for the lexical analyzer to find the object when it encounters a lexeme that is an identifier.

When the compiler passes the scope of an identifier I, its symbol-table object no longer has any references from the parse tree, or probably any other intermediate structure used by the compiler. However, a reference to the object is still sitting in the hash table. Since the hash table is part of the root set of the compiler, the object cannot be garbage collected. If another identifier with the same lexeme as I is encountered, then it will be discovered that I is out of scope, and the reference to its object will be deleted. However, if no other identifier with this lexeme is encountered, then I's object may remain as uncollectable, yet useless, throughout compilation. □

If the problem suggested by Example 7.17 is important, then the compiler writer could arrange to delete from the hash table all references to objects as soon as their scope ends. However, a technique known as *weak references* allows the programmer to rely on automatic garbage collection, and yet not have the heap burdened with reachable, yet truly unused, objects. Such a system allows certain references to be declared "weak." An example would be all the references in the hash table we have been discussing. When the garbage collector scans an object, it does not follow weak references within that object, and does not make the objects they point to reachable. Of course, such an object may still be reachable if there is another reference to it that is not weak.

7.8.5 Exercises for Section 7.8

! **Exercise 7.8.1:** In Section 7.8.3 we suggested that it was possible to garbage collect for C programs that do not fabricate expressions that point to a place within a chunk unless there is an address that points somewhere within that same chunk. Thus, we rule out code like

```
p = 12345;
x = *p;
```

because, while p might point to some chunk accidentally, there could be no other pointer to that chunk. On the other hand, with the code above, it is more likely that p points nowhere, and executing that code will result in a segmentation fault. However, in C it is possible to write code such that a variable like p is guaranteed to point to some chunk, and yet there is no pointer to that chunk. Write such a program.