Incremental Garbage Collection: The Train Algorithm

Thomas Würthinger

***Abstract*— Modern programming languages use an automated way of getting rid of unused objects called garbage collection. This paper describes the use of incremental garbage collection and discusses the Train Algorithm in particular. At first the approach to split memory into blocks is described and then the reason why trains are needed is revealed. Furthermore another not very obvious problem of the algorithm and its solution is shown. Optimization possibilities and some implementation details as well as a discussion of the Train Algorithm in distributed systems conclude the article. This paper is the last part of a three paper series about incremental garbage collection algorithms. The first two parts ([8], [6]) give an overview of different algorithms to achieve incrementality, whereas this paper concentrates on the Train Algorithm.**

***Index Terms*— Memory management**

I. Introduction

In traditional programming languages it was the responsability of the programmer to explicitely delete an object before all references to it are lost. This is the source for two very common errors: Firstly the usage of an already deleted object will probably lead to a memory exception. Secondly, if all references to an object are lost before it is deleted, it will remain in memory until program termination. Long living applications suffering from that kind of error would need more and more memory. Then it's just a matter of time until all memory is allocated and the program has to stop execution. The term "garbage" is commonly used for all objects that will never be reachable again and therefore can be deleted safely. To free the programmer from the responsability to delete unused objects, algorithms that automatically detect which objects are garbage and delete them were developed. This is called "garbage collection".

Automated memory management is a widely used approach in modern programming languages like Java or C# for example. The major drawback of this method to get rid of unused objects is, that non-incremental algorithms need to stop the main program while collecting all garbage. As memory requirements grow, also the time the program has to be stopped is getting longer. This is why incremental garbage collection has become more and more important. Realtime applications very often have to react within some milliseconds and programs with user interaction must not freeze for more than a small fraction of a second. It would be impossible to use garbage collection in online role playing games either, none of the players would accept a delay at each collection run.

Additionally there is another very important advantage of being able to collect the garbage in small steps. Whenever objects are distributed among different systems, non-incremental garbage collection algorithms would cause all of them to stop at each run. Ideas behind approaches like the Train Algorithm can be used to solve this problem too.

II. Basic Idea

The following chapter contains a description of the main ideas behind the most commonly used garbage collection algorithms and starts describing the Train Algorithm. Collecting garbage basically consists of two tasks:

- *Detection:* Somehow the collection need to distinguish between living objects and garbage. Instead of finding dead objects, very often algorithms use a different approach: All living objects are marked and rescued. References from the outside of object space to an object are called "root references". All objects pointed to by such a reference have to be marked as still alive. Recursively all objects referenced by any object marked as alive so far have to be marked too. When no more unmarked but referenced objects are found this algorithm terminates and ensures that all garbage objects are not marked or visited.
- *Freeing:* There are two different possibilities of deleting the garbage:
  - Rescue all objects that are still alive by copying them to a safe place and free the whole block filled with garbage objects.
  - Free the space of each object that is not marked as still living.

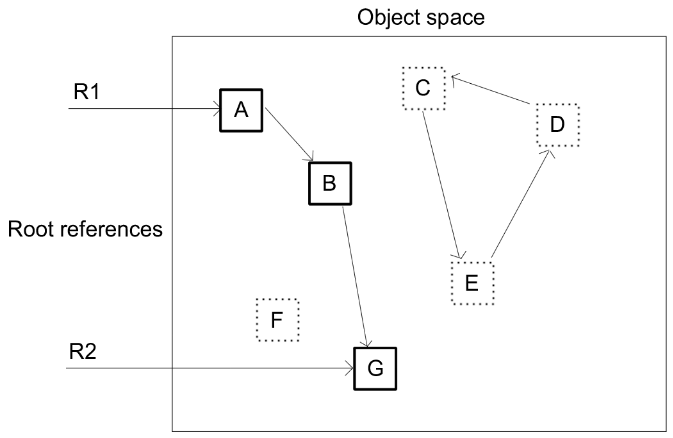

Fig. 1. Finding objects that are still alive.

In the figure above, the detection of all objects that are still alive is visualized. First A and G, which are directly referenced by root pointers, are marked. But also B, which is reachable from A has to be rescued. Note that the reference circle consisting of the objects C, D and E is correctly treated as garbage.

The idea of rescuing all referenced objects by copying them seems to be quite a performance loss at first sight. But there is also a big advantage of this technique: After each garbage collection run memory is compacted. Therefore there is no space between objects and execution is faster. Figure 2 shows the part of freeing memory by rescuing all living objects. The term “from space” is used for the memory area currently in use by the program before the gargabe collection run, whereas “to space” denotes the rescue area.

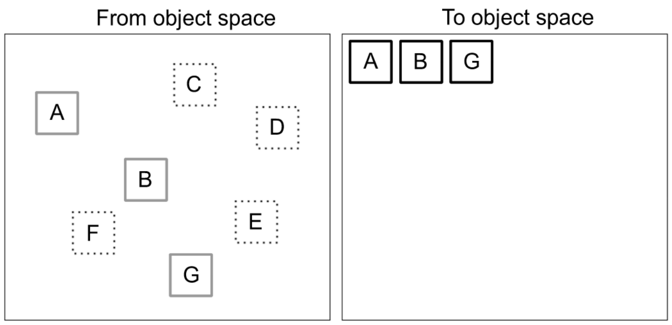

Fig. 2. Rescuing referenced objects.

When garbage collection passes must not interrupt for longer periods of time, the part of detection is the main problem. It’s difficult to find a good algorithm to do this in small steps. This is where incremental garbage collection techniques in general and the Train Algorithm in particular provide a solution.

A. Splitting Memory into Blocks

The main idea of the Train Algorithm is to split memory into small blocks and perform the mark and copy garbage collection strategy seperately for each block. The main program is only stopped while processing a single block and the interruptions are a lot shorter depending on the size of the memory pieces.

Processing only one block at a time the root references are not the only pointers coming from the outside of a block. Therefore we must keep track of all references to objects in that block that are not from other objects in the same block. The set of all such references is called the “remembered set” of a block. For simplification it is assumed that only three objects fit into one block. Figure 3 shows a possible distribution of the objects of the example shown in figure 1 in blocks.

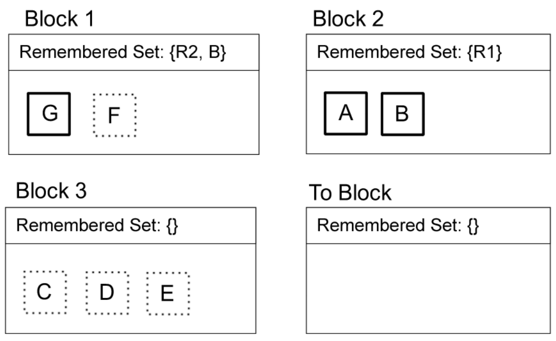

Fig. 3. Splitting object space into blocks.

When the garbage collector processes the first block, it uses the remembered set to find out that only C is referenced from the outside of the block. Therefore it rescues this object to the last block. After this the entire block is freed and the object F is deleted implicitely. The resulting object space after this first pass is shown in figure 4.

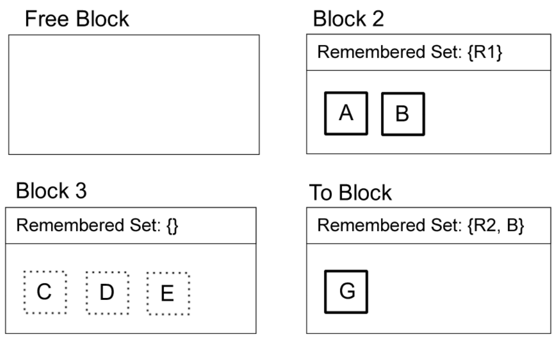

Fig. 4. Result after first pass.

During the next invocation the same procedure is applied to the next block. Object A is rescued and because object B is reachable from A it is rescued too. Because there is still some free space in the last block, no new block needs to be allocated. Note how the remembered set is updated to reflect the new situation.

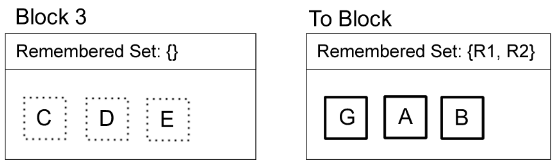

Fig. 5. After second pass.

The remembered set of the next block is empty therefore the whole block can be safely marked as free.

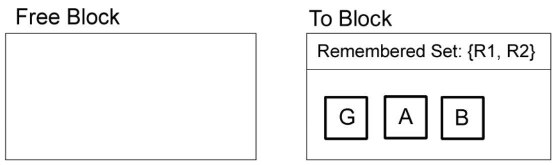

Fig. 6. Finally space is freed.

This seems to be a quite straight-forward incremental garbage collection algorithm. But there arise several problems which are discussed in the following subsections.

B. Write Barrier

First of all, how can we realize the remembered set efficiently? The answer is, without hardware support or at least with the help of the compiler there is no way. While some garbage collection algorithms need the use of a so-called “read barrier”, the Train Algorithm needs a “write barrier”. When a read barrier is used, at each pointer access some special code must be executed, whereas a write barrier only needs code execution at pointer assignments. The number of read accesses to a pointer is usually a lot higher than the number of write accesses.

Whenever a pointer asignment is made, the algorithm has to perform two steps:

- *Delete old reference:* If the old reference was registered in a remembered set we have to remove it.
- *Add new reference:* If the new reference points to an object in a block in the object space, we need to add the reference to the corresponding remembered set.

The following java pseudocode shows all four possible cases that can occur when a pointer assigment is made. It is assumed that a, b and c are put into different blocks:

```
class Pointer{
   Pointer p;
}

Pointer a = new Pointer();
Pointer b = new Pointer();
Pointer c = new Pointer();

// Case 1: Nothing has to be done no pointer to
// objects in the object space is affected.
a.p = null;

// Case 2: Add b.p to the remembered set of the
// block of a
b.p = a;

// Case 3: Remove b.p from the remembered set
// of the block of a and add it to the
// remembered set of the block c
b.p = c;

// Case 4: Remove b.p from the remembered set
// of the block of c
b.p = null;
```

There is a possible optimization if blocks are numbered and the algorithm always processes the lowest numbered block first. Then only pointers from higher numbered blocks to lower numbered blocks have to be recorded in the remembered set.

The remembered set is also used to update references as objects are copied. This can be quite a performance loss as discussed in the section about popular objects.

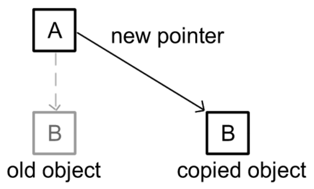

Fig. 7. Pointer needs to be updated.

C. Very large Objects

There is another problem that must be solved when dividing memory into smaller blocks. Because the block size is fixed, it is not possible to store big objects that don’t fit into one block. The solution to this problem is an extra large object space. Only the reference to the real object is written into the block. Figure 8 shows the resulting object space.

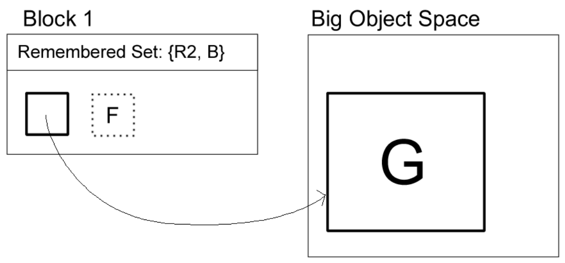

Fig. 8. How to treat big objects.

D. Problems with Cycles

Until now the presented algorithm has proven quite useful and small extension solved the problems. But there is still a very difficult problem to solve: Cycles.

So far all of the examples never had cycles that cover more than one block. Let’s look at a new one:

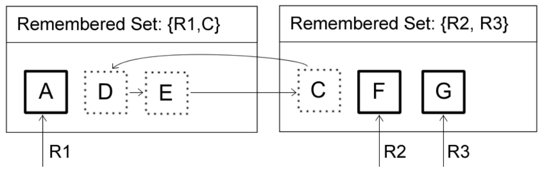

Fig. 9. Simple circle example.

Note that in this example the remembered set is optimized so it contains only references from higher numbered blocks. Now if the algorithm is applied once, one must admit that very little changes: The position of the blocks and therefore the remembered set, see figure 10. But if we do another garbage collection run, exactly the same object space as shown in figure 9 is the result! This is an endless loop, that can only be broken if the program frees A, F or G, or allocates a new object. Otherwise C, E and D are never deleted wasting memory. This is of course unacceptable!

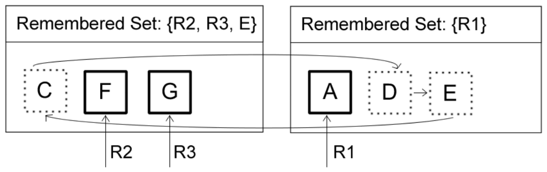

Fig. 10. After one invocation.

Obviously garbage objects that form a loop over more than one block will never get deleted. This kind of structure is very likely to occur in real programs. For example if the reference to a double linked list is lost.

The intuitive solution to this problem is to try to put all circles into one single block. Because if we can reach a situation as shown in figure 11 C, E and D would be deleted correctly. But the upper bound for the size of linked structures is the size of the available memory, so we wouldn't be able to split the memory area into smaller blocks. Let's assume that blocks can grow as large as needed, then the problem with cycles could be solved easily.

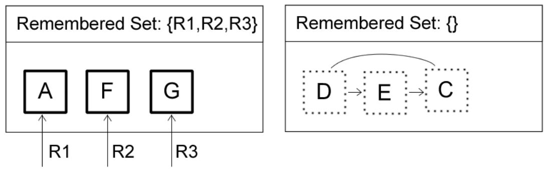

Fig. 11. In this case, the algorithm would work correct.

If an object A has to be rescued and it is referenced by an object of another block, move A into that block. Objects only pointed to by root references have to be moved to a new block. If a reference from more than one block points to an object, it can be moved to any of these blocks.

This strategy ensures that after some iterations dead linked structures are gathered in the same block. Let's see how the algorithm works on the example of figure 9:

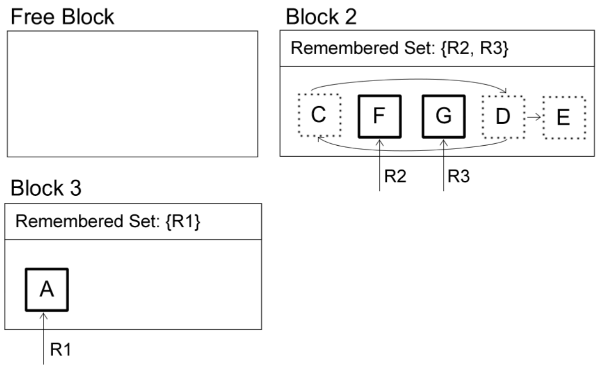

Fig. 12. Corrected algorithm, after one invocation.

Because the two objects D and E are only referenced by C they are not copied into the last block, but are added to block number 2. Note that this wouldn't work because of the assumption that a block can only hold three objects, but a solution for this problem is given in the next chapter.

The resulting object space after the next step is shown in figure 13. F and G are both reachable from root pointers so they are rescued, the rest of the objects are destroyed. The situation is very similar to figure 11. After this second invocation C, E and D are freed correctly.

After a finite number of invocations a circle consisting exclusively of garbage objects is finally collected. At least if the program does not continuously change the root pointer to two objects, this very subtle error is discussed in the section about the correct Train Algorithm. But now a way to rebuild the behaviour of arbitrary sized blocks has to be found. Just resizing the blocks is not a good solution, because at each iteration one block is processed and so the collection strategy would not be incremental.

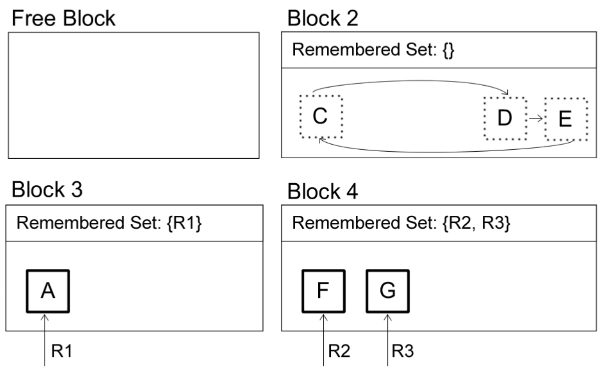

Fig. 13. Corrected algorithm, during second invocation.

III. Railway Station Metaphor

The Train Algorithm has a solution for the problem of arbitrary sized blocks and this solution is the reason for its name.

A. *Using Trains*

For a better understanding of the algorithm the object space can be seen as a big railway station. Blocks still have a fix size and are now called "cars". They are grouped and form "trains" that can have any number of cars. There is a remembered set for each car and also a remembered set for each train that contains only of references among trains.

B. *Example*

Figure 14 is based on the same object structure as figure 1, but now the object space is organized as a railway station. It consists of an ordered number of trains which can have an arbitrary number of cars, that are ordered too. In this example there are two trains. There can be a maximum number of three objects in a single car, but of course a train can consist of any number of cars.

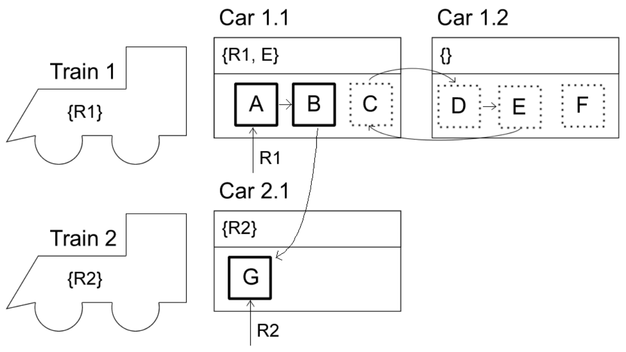

Fig. 14. The railway station.

The reference set of a train is the union of the reference sets of all its cars without any intra train reference. In figure 14 object E is for example a member of the reference set of car 1.1, but it is not a member of the set of train number 1. Because the algorithm processes always the lowest numbered car first, only references from higher numbered cars have to be considered when updating the sets. Therefore object E is in the remembered set of car 1.1, but C is not a member of the set of car 1.2.

When the garbage collector processes the first car, object A is rescued and copied to an entirely new train because it is referenced by a root pointer. B is only referenced by A and therefore copied to the *same* train as A. This is very important because this way cyclic dead garbage structures end up in a single train. Because C is referenced by an object of the same train it is copied to the end of the train. Now the first car is empty and can be freed. The state of the railway station after this first pass is shown in figure 15.

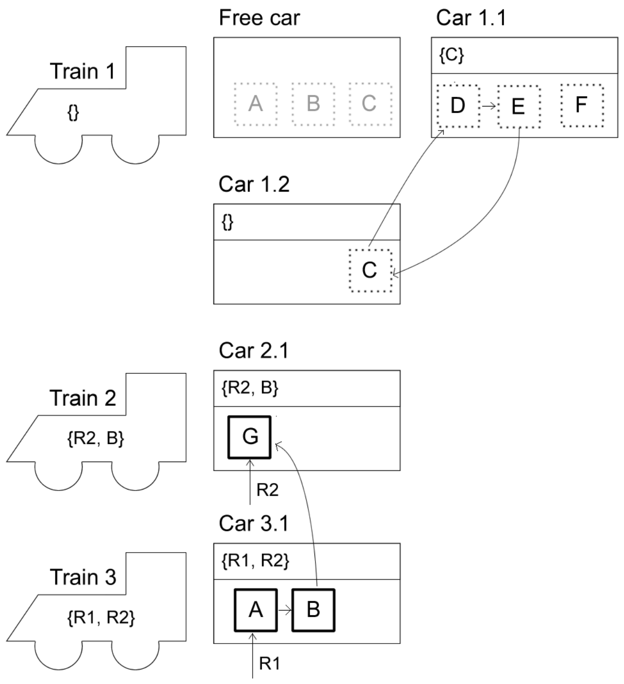

Fig. 15. After one invocation.

The remembered sets have been updated accordingly. Now the first train is not referenced from anywhere outside and so during the next iteration the garbage collector will safely free the whole train, resulting in an object space as in figure 16.

Whenever there is a cyclic garbage structure in the first train, it will never be copied out of this train. After all objects that are not part of the circle are copied into other trains, it will be freed. This is quite easy to understand, but will really every cyclic structure end up in the first train? If a cycle is split upon serveral trains, the first of those trains will become the first train after some iterations. After this train is processed all members of the cyclic structure are distributed among the other trains containing some structure members. Therefore the number of trains containing members is decreased by one.

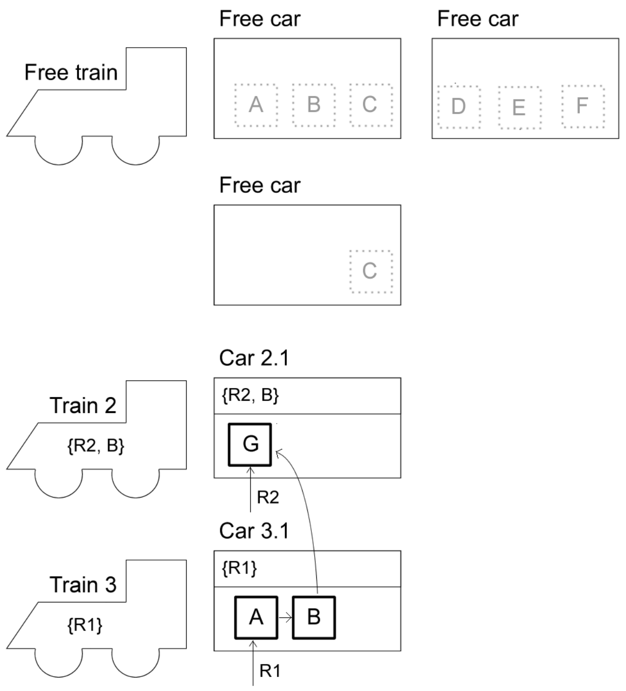

Fig. 16. After two invocations.

When this number reaches one and the train containing now all members becomes the first train, the garbage structure will be correctly collected.

Figure 17 shows the example of a cycle consisting of four objects A, B, C and D. After the first invocation of the algorithm train 1 is freed and object A is moved to train 2. The next time A and B are moved to train 3 and during the next step A, B and C are moved to train 3. Now all members of the cycle are in one train and the next invocation will free the structure.

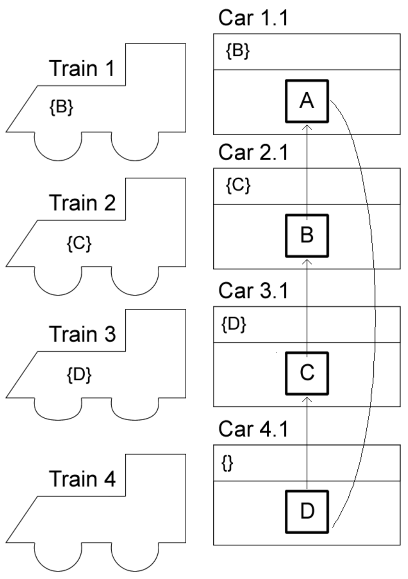

Fig. 17. Example for a cyclic structure.

As it seems this algorithm works fine.

C. Algorithm as Text

The following steps describe the train algorithm in textform.

1) Select the lowest numbered train.
2) Free the whole train if the remembered set of the train is empty and terminate, otherwise go to step 3.
3) Select the lowest numbered car within the train.
4) For each element of the remembered set of that car:
    a) If the object wasn't processed earlier, rescue it by copying it to a new train if it is a root reference or to the train of the source object of the pointer.
    b) Move all objects within the same car that are reachable from the rescued object to the same train.

    In this step it is necessary to update the affected reference sets accordingly. If an object is referenced from more than one train it can be copied to any of them.
5) Free the car and terminate.

D. Algorithm in Java

The following Java pseudocode assumes the availability of the following classes and functions:

- ObjectSpace
  - Car getLowestNumberedCar();
    Returns car number 1.1 or null if there is no car in the object space.
  - Train createTrain();
    Returns a newly created train that is appended at the end of the list of trains.
- Car
  - Train getTrain();
    Returns the train that corresponds to this car.
  - Car hasLowerNumber(Car c);
    Returns true if this car will be processed before Car c.
  - IList<Reference> getRememberedSet();
    Returns a list of references that represent the remembered set of this car.
  - void free();
    Frees this car.
- Train
  - int getRememberedSetSize();
    Returns the size of the rembered set of that train.
  - void addObject(Object o);
    Adds an object to the last car of the train. If the last car is full, a new car is automatically appended. The car pointer of the train has to be updated. Also the reference sets of the old and new car and reference set sizes of their corresponding trains.
  - void free();
    Frees the entire train.
- Reference
  - Object getSourceObject();
    Returns the source of the reference.
  - Object getReferencedObject();
    Returns the referenced object.
- Object
  - IList¡Reference¿ getReferences();
    Returns all references that are contained in the object.
  - Car getCar();
    Returns the car that corresponds to this object or null if this object is not part of the object space.
  - bool isReferencedFromTheOutside();
    Returns true if this object is referenced from outside its train.

```
public void doCollectionRun(ObjectSpace objectSpace){

   Car car = objectSpace.getLowestNumberedCar();
   if(car == null) return;

   Train train = car.getTrain();
   if(train.getRememberedSetSize() == 0){
      // The entire train may be freed.
      train.free();
      return;
   }

   IList<Reference> rememberedSet = car.getRememberedSet();
   for(Reference r : rememberedSet){

      Object o = r.getReferencedObject();
      Object source = r.getSourceObject();

      if(o.getCar() != car){

         // Object has been already processed
         if(o.getCar() != source.getCar(){

            // Update remembered sets
            if(source.getCar() == null /* root reference */
               || o.getCar().hasLowerNumber(source.getCar()){
               o.getCar().getRememberedSet().add(r);
            }else{
               source.getCar().getRememberedSet().add(r);
            }
         }

         continue;
      }

      if(source.getCar() == null){
         // Root reference
         objectSpace.createTrain().addObject(o);
      }else if(source.getCar().getTrain() == car.getTrain()){
         if(!o.isReferencedFromTheOutside()){
            // Move to the end of the same train
            // only if there is no reference from the outside
            source.getCar().getTrain().addObject(o);
         }
      }else{
         // Move to other train
         source.getCar().getTrain().addObject(o);
      }

      // Rescue all objects reachable from o
      Queue<Object> queue;
      queue.add(o);
      while(!queue.isEmpty()){

         Object cur = queue.get();
         IList<Reference> references = cur.getReferences();

         for(Reference r : references){
            if(r.getSourceObject().getCar() == car &&
               !r.getSourceObject().isReferencedFromTheOutside()){

               // Add reachable object to same train
               o.getCar().getTrain().addObject(r.getSourceObject());
            }
         }
      }
   }

   car.free();
}
```

The code listing shows a simple version of the Train Algorithm, but still it contains an error and some implementation details of the assumed functions are in need of an explanation. The following chapters discuss these topics.

IV. Correct Train Algorithm

One of the most important properties of a garbage collection algorithm is that all garbage is collected and there cannot occur any infinite loop. Cyclic structures are handled correctly by the shown algorithm, but there exists still a more sophisticated problem.

A. Error

The program can do anything between two garbage collection runs, for example changing a pointer. For the sake of argument it is assumed that only one object fits into a single car. Consider the following Java pseudocode:

```
class Pointer{
    Pointer p;
}

Pointer objectA = new Pointer();
Pointer objectB = new Pointer();

Pointer R1 = objectB; // Object A
R1.p = objectA; // Object B

// Ensure that there are no other
// root references to A or B
objectA = null;
objectB = null;

while(true){

    // Garbage collection run

    Pointer tmp = R1.p;
    R1.p = R1;
    R1 = tmp;
    tmp = null;

    new Pointer(); // This object is never collected!
}
```

If the garbage collection run is always exactly at the marked position, there will be never any garbage collected and the program will use more and more memory at each loop pass.

Figure 18 shows the object space after the configuration. Object B is referenced by a root pointer and object A is referenced by B.

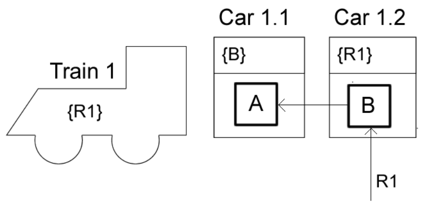

Fig. 18. Initial object space.

After performing a single garbage collection pass, the resulting object space is shown in figure 18. Until now, there are no surprises and if the program wouldn't change anything, the next run of the garbage collector would move object B to the last train. All of the other dead objects would be correctly collected.

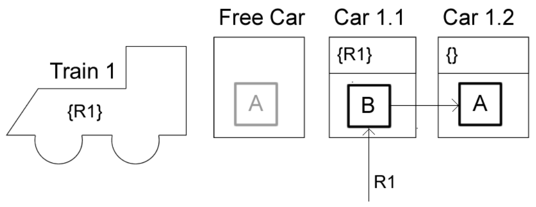

Fig. 19. Object space after first garbage collection run.

But now the program changes the pointers so object A is now referenced by the root pointer and object B is referenced by A. Figure 20 shows the resulting configuration.

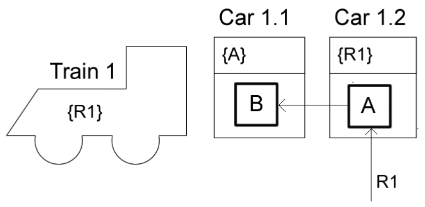

Fig. 20. Object space after program execution.

When the garbage collecter processes the next car, instead of moving B to the last train, it notices that B is referenced from inside the same train and moves it into a new car at the end of train 1. Now the program changes the pointer again and the resulting objects space has exactly the same first train as the one in figure 18, the only difference is that the object space contains also all newly allocated objects. Therefore the garbage collector will never delete any unused objects.

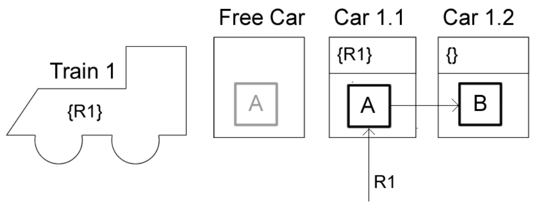

Fig. 21. Object space after next garbage collection run.

Obviously this is a serious problem of the algorithm. Of course it's very unlikely for a program to behave like that, but there is still a possibility that this can occur. Fortunately there is also a solution for this strange case.

Just remember that B was pointed to by a reference from outside the train and ensure that it is copied out of train number one even if there are no external references. This is of course only needed if there were no objects copied out of the first train in the current pass. Then the first member of the remembered set of the train is saved. When processing a car the algorithm has to look if there is such a special reference and treat it as if it would still exist.

B. Corrected Algorithm

Now let's look how the algorithm of the previous chapter can be corrected to work even for the subtle case discussed in the former section.

In the textual representation we just need to add that if the remembered set of the current car is empty and there exists a special saved reference which points into the car, this reference is added to the remembered set. If the special reference points into the lowest numbered car it has to be removed after the pass.

The Java pseudocode model of the algorithm needs a new member variable in the ObjectSpace class which is the special reference or null if no such reference exists. When the special reference points into the current car and there is no other reference to that car from the outside of the current train, this reference has to be added to the reference list.

Whenever none of the objects of the first car is copied out of the train, a member of the trains remembered set has to be saved as the special reference. The remembered set of a train normally isn't explicitely saved because it is just the union of the remembered set of all cars without any intra train references. Therefore the collector would have to iterate over all cars, but this could be quite a performance loss. Fortunately there is a solution: Whenever no objects are copied out of the first train, set a flag. If the flag is set, the write barrier algorithm will write the old reference as the special value and clear the flag again when the next pointer assignment that affects the first train is made.

V. Optimization

This chapter discusses the performance of the Train Algorithm and shows some possible solutions for implemenation problems as proposed in [1].

A. Getting the Train from an Address

First of all, we need to find a very fast way to get the car of an object and the train of a car. Obviously this is needed quite often as shown in the Java pseudocode.

The solution is to choose a car size of the form 2^n and also align cars on a 2^n boundary. Then the index of the car corresponding to an object can be calculated as follows:

```
int address = object.getAddress();
int carIndex = address >> n;
```

A simple right shift does the job and is quite high-performance. Now we need an array that has an entry for each possible car index. The size of this array is the amount of memory available divided by 2^n. For each car we save the number of its train and the number of the car itself.

```
class Car{
    int trainNumber;
    int carNumber;
}

Car[] indexTable = new Car[MEMORY_AVAILABLE >> n];
```

The next table shows a possible configuration of the index table corresponding to the object space shown in figure 22. It is assumed that the block size is 0xfff (4096).

index	train	car	memory
0	3	2	0x0000 - 0x0FFF
1			0x1000 - 0x1FFF
2	1	1	0x2000 - 0x2FFF
3	2	1	0x3000 - 0x3FFF
4			0x4000 - 0x4FFF
5	3	3	0x5000 - 0x5FFF
6			0x6000 - 0x6FFF
7	1	2	0x7000 - 0x7FFF
8	3	1	0x8000 - 0x8FFF

Note that a simple right shift by 12 gives the table index of any memory address.

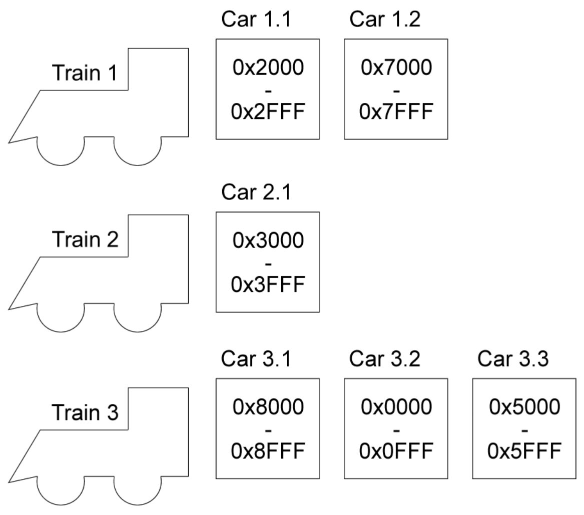

Fig. 22. Possible distribution of cars in memory.

Only the first train and only the first car of a train can be removed. So a simple linked list of all trains and one for each train containing all cars will ensure that finding the next car or the next train will be very fast.

Whenever a car is freed it is removed from the linked list of the train and added to a freelist. This way it even takes only a constant amount of time to free an entire train, because the linked cars can be added entirely to the freelist.

B. Popular Objects

A possible performance problem of the algorithm are so-called popular objects. Whenever an object is referenced by lots of other objects it is called popular. Cars containing popular objects have a very large remembered set and copying such objects requires lots of pointers to be updated. The problem hereby is that the number of references pointing to an object is only bounded by the entire number of references. So collecting a very popular object can mean a lot of work for the garbage collector and this would cause the normal program to stop for an unacceptable long time.

A possible solution to this problem would be to use indirect addressing. All references to an object go to a place where the real address of the object is saved. This way moving the object around can be done very fast by simply updating the pointer. Figure 23 shows this strategy at work.

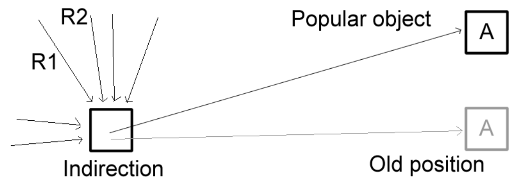

Fig. 23. Using indirect addressing.

But there is a drawback of this solution. As discussed earlier pointer access occurs a lot more often than pointer assignments in the majority of programs. The indirection requires an additional performance overhead at each read access to a pointer.

So is there any other chance of handling popular objects? Simply don't copy them! Whenever a car contains a popular object don't collect it and move it to the end of the train. Unfortunately, a lot of problems occur if this is done, because now endless loops and not collected garbage are again possible. But all of these problems can be solved, however a lot of effort is needed.

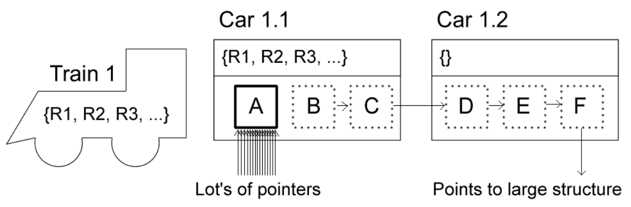

Fig. 24. Popular objects, first problem.

Let's discuss the first problem: If there is an object in a car together with a popular object, it will never get collected. And even worse, all objects that are only referenced by this object won't be freed either. Figure 24 shows this problematic situation. B, C, D, E and F would be garbage but will never get collected before the popular object A is collected. But because of the popularity of A this is very likely to take a long time, maybe even until the end of the program.

There is an additional problem if two popular objects form a cycle as shown in figure 25. If a new train is started whenever a popular object has to be moved logically, there is again an endless loop situation. The whole structure would be garbage but is never collected.

So first of all, a way to rescue all non-popular objects out of a car has to be found before moving it. This is not as easy as it seems at first sight, because if there are popular objects in a car, the remembered set that has to be processed can be very large. A seperate remembered set for each object in a car is needed. Whenever the number of reference to an object is higher than a certain number it is called popular and the remembered set for this object is discarded. A lot of memory is used, but the first problem is solved.

To which train should the car be appended after rescuing all normal objects? As there is no remembered set for popular objects, to know this, an additional entry has to be saved for each object: The train with the highest number that contains a reference that points to this object. This value can be easily updated instead of adding an entry to the remembered set.

But the problem prooves to be even more tricky, because what to do if a car contains more than one popular object? Cars have to be able to get splitted. After rescuing all non-popular objects, the physical car is divided into smaller parts, each part containing a popular object.

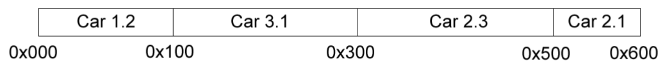

Fig. 26. A car split logically into four pieces.

Now special care has to be taken to find out to which car an object belongs to. The very efficient and simply strategy as presented in the previous section does not work any more for split cars. In fact a binary search tree is needed to find out to which car a given memory address belongs to. Figure 26 shows a car split into four pieces and figure 27 shows the corresponding binary search tree.

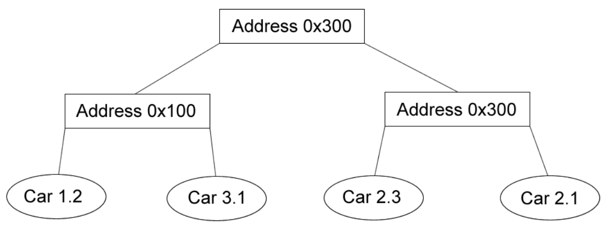

Fig. 27. A car search tree.

Popular objects are rather seldom in normal programs so the number of popular objects in a car should be small. If the search tree is balanced, the height of the tree will never be higher than $\log_2(\text{Maximal number of objects per car})$.

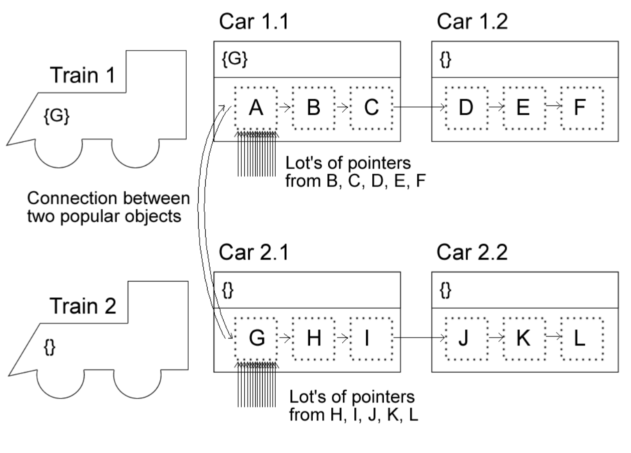

Fig. 25. Popular objects, second problem.

VI. Distributed Garbage Collection

The following chapter gives an overview of the DMOS (Distributed Mature Object Space) garbage collection algorithm as presented by Richard L. Hudson, Ron Morrison, J. Elliot B. Moss and David S. Munro in 1997. It uses the idea of the Train Algorithm extended to work in a distributed system. The term "node" is used for a member of the system and it is assumed that all nodes can communicate via messages. Furthermore for simplicity problems that arise if a message does not reach its destination or if a node does not react are not discussed.

A. Pointer Tracking

The first thing we must worry about when using garbage collection algorithms in distributed systems is, how to know about external references from other nodes? Messages have to be used to keep the knowlegde of each node up to date.

Every object has a so-called home node. The physical representation of the object resides at the home node and this node is also responsible for the deletion of the object. All other nodes may only hold references to the object. A message protocol is needed for the home node to know which external references point to its own objects.

- *Send event:* Whenever A sends a pointer to an object to another node B it has to inform the home node of the object.
- *Receive event:* When B receives the pointer the home node wants to be informed too.
- *Delete event:* After B deletes the pointer from its message buffers it sends another event.
- *Add pointer event:* For each newly created reference to an object on a node not being its home node an event is created.
- *Remove pointer event:* Analogically for each lost pointer an event is sent.

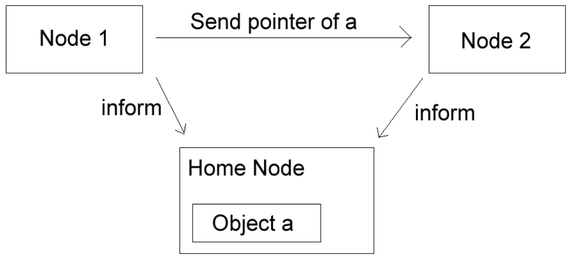

Fig. 28. Node 1 sends a pointer to Node 2.

Figure 28 shows a possible scenario. The steps taken by the nodes in order are:

1) Node 1 sends a message to the home node imforming it about the pointer transfer.
2) Node 1 sends the pointer to Node 2.
3) Node 2 receives the pointer and sends a message to the home node about the successful reception.
4) Node 2 tells the home node that another pointer to object a is installed.
5) Node 2 deletes the pointer from its reception buffer and sends another message to the home node about this fact.

Whenever the pointer is copied or overwritten at Node 2 the home node is again informed. There are of course several possible optimizations that minimize the number of messages sent. Those are discussed in detail in [2].

B. Distributed Trains

Using the Train Algorithm in a distributed system, trains spanning more than one node are needed. Of course we could copy objects around and by this ensure that a all objects of a certain train are on the same node. But this is quite a performance-loss and very often unwanted. Therefore a system must be designed to allow cars of trains to be spread upon nodes.

Each train has a so-called master node. This node created the train and is responsible for managing the train and adding new nodes that need to create cars of that train. All nodes using a certain train are connected using a token ring scheme. Whenever node wants to join the ring it sends a message to the master node and then it is added to the ring right after the master. Each node in the ring knows its successor. The following Java pseudocode does the job.

```
class Node{

   Node successor;

   void wantToJoin(Node n)
   {
      n.successor = this.successor;
       this.successor = n;
   }
}
```

Figure 29 shows the scenario when a node wants to enter the ring.

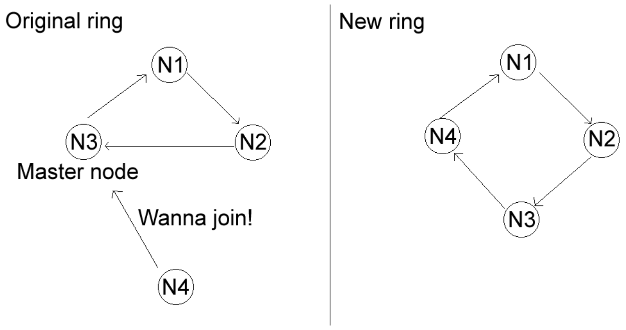

Fig. 29. Node 4 wants to enter the ring.

A node that does not contain any cars of the train any more may leave the ring. Note that the master node must never leave until the train is destroyed, because it is responsible for welcoming new nodes.

The idea of how a node can leave the ring is as follows: Pass a token around the ring and when it reaches the predecessor of the node that wants to leave, this predecessor sets its successor accordingly. Special care is given to the possibility that more than one node in a row wants to leave the ring. Then they will be able to leave simultaneously. A sample implementation is shown as the Java pseudocode below, figure 30 gives a visualization.

```
class Node{

   Node successor;

   void processLeaveMessage(Node leaver, Node newSuccessor)
   {
      if(this.successor == leaver){
         // We are predecessor
         this.successor = newSuccessor;
      }else{
         if(newSuccessor == this && weWantToLeave()){
            successor.processLeaveMessage(leaver, successor);
         }else{
            successor.processLeaveMessage(leaver, newSuccessor);
         }
      }
   }
}
```

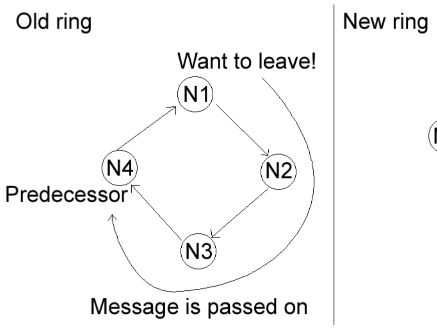

Fig. 30. Node 1 wants to leave the ring.

One of the main problems that must be solved when there is no global synchronization is, how to find out when can a train be discarded? Again the token ring system is used. An important fact has to be kept in mind when designing an algorithm for this: Whenever there are no external references into a train, whatever happens there will be never any external reference to it. This is simply ensured by the fact that nobody knows anything about any object of the train, as if the whole train would not exist.

The first approach would be to simply let the token only go on to the next node when there are no external references to the train on this node. After a full circle of the token, the train is discarded. But this is simply wrong, because even if there are no external references to the cars at some Node A at a certain time, because of references to that train at other nodes new external references to objects at A could be created.

So we can only discard the train if the token passed one round as proposed and passes an additional round to ensure that no new external references were added to any node meanwhile. This results in the following algorithm:

```
class Train{
   boolean changed;
   int numberOfExternalReferences;

   void removeExternalReference(){
      numberOfExternalReferences--;
   }

   void newExternalReference(){
      numberOfExternalReferences++;
      changed = true;
   }
}

class Node{

   Node successor;

   void processDiscardTrain(Node sender, Train train)
   {
      if(!train.changed){

         if(sender == this){

            // The train can be safely discarded
            train.discard();
         }else{

            successor.processDiscardTrain(sender, train);
         }

      }else{
         while(train.numberOfExternalReferences > 0){
            wait();
         }

         train.changed = false;
         successor.processDiscardTrain(this, train);
      }
   }
}
```

Any node that knows that there are no external references into its part of the train can start the process. The table shows as the token progresses if Node 1 starts by sending a message to Node 2:

current node	N2	N3	N1	N2	N3	N1
sender	N1	N2	N3	N1	N1	N1
old changed	T	T	T	F	F	F
new changed	F	F	F	F	F	F

Now the changed value of the sender is false and it is equal to the current node, so the train may be deleted. If for example a new reference would have been added to Node 3 in the meantime, the train would not be freed. When the token reaches the node, source is set to N3 and the token has to wait until the external reference is deleted.

C. Unwanted Relative Problem

Because the train that is processed is not always the first train as in the normal Train Algorithm and more than one train may be processed at a time, it is possible that new objects are inserted into the train while trying to look if it can be discarded. This can only occur if an object has an unwanted relative in a lower numbered train. Figure 31 shows the problematic case.

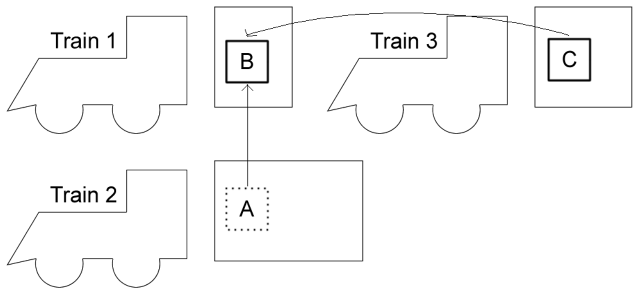

Fig. 31. A has an unwanted relative B.

The train of object A could be collected, but if the car of B is processed meanwhile the situation is as shown in figure 32. Because there is an external reference to B, the train cannot be deleted any more!

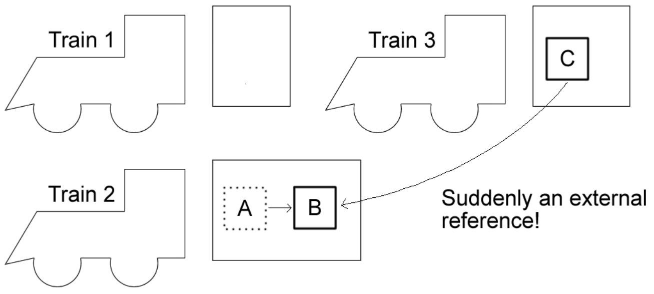

Fig. 32. Now the train has an external reference.

There is no easy possibility to bar the other collectors from adding cars to a certain train. A possible solution is to track all objects that are added to the train after the changed bit is set to false. If the train is really collected those objects must be rescued by forming a new train.

VII. Conclusion

The Train Algorithm is nowadays used very often in combination with other garbage collection algorithms. The Java Hotspot virtual machine for example uses a collection strategy as shown in figure 33.

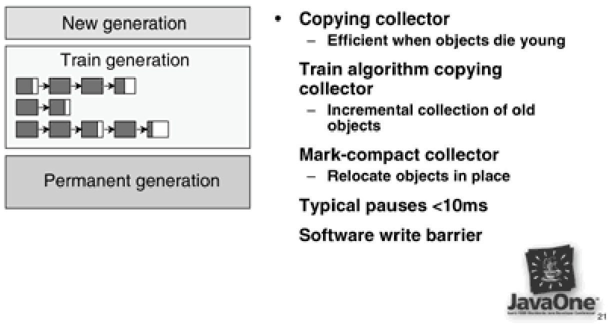

Fig. 33. Garbage collection, Java Hotspot VM [4].

Whenever a system needs to react very fast it is indispensable to use an incremental strategy. The Train Algorithm gives such an incremental garbage collection strategy that can also be used in distributed systems.

List of Figures

References

[1] S. Grarup and J. Seligmann, *Incremental Mature Garbage Collection*. Department of Comp. Science, Aarhus University, 1993.

[2] R. L. Hudson, R. Morrison, J. E. B. Moss, D. S. Munro, *Garbage Collecting the World: One Car at a Time*. Proc. OOPSLA 97, pp.162-175.

[3] R. Jones, R. Lins, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.*. John Wiley, 1996.

[4] S. Meloan, *The Java HotSpot Performance Engine: An In-Depth Look*. http://java.sun.com/developer/technicalArticles/Networking/HotSpot/, 1999.

[5] M. C. Lowry, *A New Approach to The Train Algorithm For Distributed Garbage Collection*. School of Comp. Science, University of Adelaide, 2004.

[6] R. Schatz, *Incremental Garbage Collection II*. Seminar aus Softwareentwicklung: Garbage Collection, 2006.

[7] R. Sedgewick, *Algorithmen*. Pearson Studium, 2002.

[8] C. Wirth, *Incremental Garbage Collection I*. Seminar aus Softwareentwicklung: Garbage Collection, 2006.