Scheduling Real Time Garbage Collection

Roger Henriksson

LU-CS-TR:94-129

Also published in Proceedings of NWPER'94, Nordic Workshop on Programming Environment Research, Lund, Sweden, June 1994.

Department of Computer Science
Lund University

P.O. Box 118
S-221 00 Lund
Sweden

Scheduling Real Time Garbage Collection

Roger Henriksson

Dept of Computer Science, Lund University
Box 1 18, S-221 00 Lund, Sweden
e-mail: Roger .Henriksson@dna.lth.se

Abstract. This paper presents a new method for scheduling the work of an incremental garbage collector in a system with hard real time requirements. The method utilizes the fact that just some of the processes in the system have to meet hard real time requirements and that these processes typically run periodically , a fact that we can make use of when scheduling the garbage collection. The work of the collector is scheduled to be performed in the pauses between the critical processesis and is suspended when the processes with hard real time requirements run. It is shown that this approach is feasible for many real time systems and that it leaves the time-critical parts of the system undisturbed from garbage collection induced delays.

1 Introduction

During the last few decades, we have observed an increased use of computers in process control equipment and other types of real time systems, e.g robot control and telephone exchanges. The demands on the software of these systems have increased as advances have been made in the fields of automated control and computer engineering, forcing engineers to develop new techniques for writing real time software.

A *real time application* is characterized by the need to guarantee short response times to external events and to run continuously without faults for very long periods. Short response times are needed in order to keep up with the external process being controlled. Failure to do so, or having the system go down due to a software error , may cause expensive, and perhaps even dangerous, interruptions in the process being controlled. For example, failure of the software controlling a robot in a manufacturing plant may not only delay the production, but an uncontrolled robot may also injure nearby workers. It is thus of vital importance that real time software is able to keep up with the process, i.e guarantee short response times, and that it really is reliable.

One issue that affects how well the real time system meets the above requirements is how the memory of the computer is managed. The real time software can either use *statically allocated* memory only , or in addition use *dynamically allocated* memory . Statically allocated memory is allocated when the application starts and is never deallocated. Using only this type of memory may be fine for many real time applications, but it requires that it is possible to calculate the maximum sizes of all internal data structures in advance. Complicated data structures may be difficult to handle with statically allocated memory only . The ability to manage complex data structures becomes more important as *object-oriented technologies* are applied to the development and implementation of real time software. We thus see that the freedom of the programmer becomes limited to an undesirable degree, especially in the light of the increased complexity of real time software. Some method to dynamically allocate, and release, memory would be desirable.

The traditional way to implement *dynamical memory management* is to let the application manage the free store on its own, perhaps with some rudimentary help from the runtime system. This is typically done by building a linked list containing all free memory segments, called the free-list. When the application demands more memory, the free-list is searched for a memory segment with a suitable size. The free-list is updated and a memory area of the requested size is delivered to the application. When a memory segment is released by the application, it is again linked into the free-list.

There are two big disadvantages of using this method in real time applications. The first disadvantage stems from the fact that most applications allocate and deallocate memory chunks of variable sizes. This leads to *memory fragmentation* and an arbitrary long free-list. That in turns makes it difficult to calculate the worst-case response time for an allocation operation. In the worst case, the system must traverse a very long free-list in its entirety before finding a segment that is large enough. If the delays caused by searching the free-list become too long it may not be possible to guarantee short bounds on the response times of the system as a whole.

The second draw-back comes from that the application, and ultimately the programmer, must see to that memory is allocated and released in a correct fashion. Two types of programming errors that are very common in large software systems are that chunks of memory aren't deallocated when they should be, *memory leaks*, and that memory is deallocated when it shouldn't be, introducing *dangling pointers*.

Forgetting to release unused memory means that these segments aren't linked into the free-list and can thus not be reused. Sooner or later, the free store will be exhausted and the application will crash. Errors of this type are typically very difficult to locate and it is nearly impossible to prove that an application doesn't leak memory. A memory leak is much more severe in a real time system than in other applications since a real time system must be able to run without faults for very long periods, for some industrial processes months or years.

Deallocating a chunk of memory that is still in use introduces dangling pointers into the data structures of the application. An attempt to dereference such a pointer produces strange, erratic, effects and will in many cases cause a crash. Manually managed memory must thus be considered less suitable for complex real time systems since it makes them unreliable.

To satisfy the reliability restriction one needs support for *automatic memory management*. Furthermore, in order to avoid fragmentation *memory compaction* is desirable. By including a compacting garbage collector in the runtime system and leave the task of identifying dead objects and deallocate them to it, the problem of memory management errors and fragmentation can be eliminated. This would be a largestep towards reliable real-time systems. The problem with this strategy is that existing garbage collectors can't guarantee short enough worst-case response times for systems with very hard real-time demands, such.

Solutions are available for *interactive systems*, i.e systems that only have to make garbage collection unnoticable to a human user. For *real time systems*, which have to satisfy much tougher demands, suitable approaches are lacking. We think, however, that this problem can be solved and this paper attempts to take a step in that direction.

2 Previous Work

Development of algorithms for garbage collection started in the late fifties when the first list-processing languages were implemented. These algorithms were of the *stop-and-compact* type. This means that the application program, called the *mutator* since it modifies the object graph, executes, allocating new objects, until it runs out of memory. At that time the mutator is halted and the garbage collector, abbreviated *collector*, goes into action, traversing the pointer graph, identifying garbage, and compacting the heap. When it is done, the mutator can continue execution. The time the collector needs to collect all garbage objects and to compact the heap may be very long, typically many seconds, but in extreme cases several minutes.

There are essentially three basic approaches to how garbage is identified and deallocated: reference-counting, mark-and-sweep, and copying.

Reference-counting relies on keeping a count on every object of how many pointers there are to the object in question. When the count reaches zero, there are no pointers to the object and it can be marked as garbage. Reference-counting can't identify and reclaim cyclic structures of garbage objects.

The *mark-and-sweep algorithms* leave the task of identifying the garbage to the collector. The collector cycle is divided into two parts: identifying garbage and compacting the heap. Live objects are found by recursively traversing the object graph starting from all the root pointers. All objects encountered during this traversal are reachable and thus alive. Objects not marked are considered garbage and are removed by compacting the heap.

Copying algorithms divides the heap into two equally sized areas, the *Fromspace* and the *Tospace*. Objects are allocated in one of the areas, the Fromspace, until it is full, after which the live objects are copied to a contiguous area in the Tospace by traversing the object graph starting with the root pointers. Dead objects are not reached when traversing the graph and are thus not copied. When all live objects are copied, the functions of Fromspace and Tospace are flipped. The drawback of copying algorithms is that only half of the heap size can be used for storing objects.

With the advent of *interactive* computer systems, the long delays caused by garbage collection became intolerable. The solution to this problem was to make the collector *incremental*, i.e dividing the collector cycle into many small pieces of work which are interleaved with the execution of the mutator. All three of the basic algorithms can be modified to be incremental, but a very popular algorithm is based on the copying approach and was published in [Bak78]. Several modifications of this algorithm have been presented since then, for example in [Bro84].

Using incremental garbage collectors has reduced the individual delays caused by garbage collection to tenths of a second, making it virtually unnoticable in most interactive systems. It is, however, still difficult to guarantee short worst-case delays, which makes the algorithms less suitable for real-time applications, which have to satisfy much harder requirements. In the past few years, researchers have tried to modify the incremental algorithms in various ways, decreasing the duration of the individual garbage collection induced delays. It's claimed that if the individual delays are small enough, these algorithm can be used in true real-time systems. An example is the work published in [EV92]. Since the worst-case behaviour for a longer sequence of actions is still quite unpredictable, we do not think these approaches are quite sufficient for systems with very hard real-time requirements.

3 The Anatomy of a Real Time System

In order to solve the problems of garbage collection in systems with hard real time requirements, it might be a good idea to study how such systems are designed. Many of these systems are controlling some kind of external process, such as the machines of a manufacturing plant. The state of the external process is sampled with regular intervals, several times a second, and appropriate actions are taken as a result of these samples. This part of the system could thus be said to be clock-driven, or periodic, in its nature. It is vital that this sampling is done regularly and that the time from sampling to the output of modified control signals is short in order for the system to stay in control of the external process in question. Investigating existing actual production systems show that typical cycle times are around 50ms. Some processes do, however, sample as often as every 10ms.

There are other tasks the system is expected to perform that are not as critical as the described tasks. This includes tasks such as presentation of data to the operator, communication with other computers, and long-time planning. It is still important that these tasks do not lag behind, but short, to a human unnoticable, delays are accepted. Even occasional longer delays can be accepted in many systems. We see that we have two levels of sensitivity to garbage collector induced delays (and of course other types of delays as well).

Real time software systems are in most cases implemented using a runtime environment supporting preemptively scheduled processes with priority. The different tasks are assigned to different concurrently running processes, which can be classified accordingly to the above observations into processes with high priority (sensitive to delays) and processes with low priority (not very sensitive to delays). Another observation that can be made is that the number of processes with high priority, and the amount of resources in the form of time and memory they need, are quite small compared to the demands of the system as a whole. Interviews with people from the process control industry support the observations made above.

4 The Proposed Scheduling Strategy

In this section we will study an approach to how the garbage collection work can be scheduled in order to take advantage of the properties of the real time system. First, a discussion of the general principles and assumptions will be presented, which will be followed by a more in-depth description of the proposed strategy.

4.1 Principles and Assumptions

The idea behind this paper is that our knowledge about the inner workings of real time systems can be used to schedule the garbage collector work in such a way that time-critical parts of the system are disturbed as little as possible. This means that the processes with high priority should run without being interrupted by the garbage collector. Thus, no garbage collection should be done when these processes are running. To be able to guarantee that this will be possible, there must always be enough memory free when the high priority processes are invoked for them to finish.

When the high priority processes run, garbage will be generated. To keep up with this, the processor time left must be divided between the collector and the processes with low priority. The garbage collector could be viewed as a separate process with a priority which is lower than for the high priority processes but higher than for the low priority ones. The scheduling principle is illustrated in figure 1. Since the high priority processes typically only use a small fraction of the total CPU time and have quite modest memory needs, there should practically always be plenty of time left for the low priority processes to run. To make sure that there always is enough free store to run the high priority processes, some garbage collector work may also have to be done while running the low priority processes as we will see later.

Scheduling the garbage collector work in this way is feasible if we can predict when the processes with high priority will be invoked. This is easy in many real time systems since these activities are periodic. Also, their memory demands are low and easy to estimate an upper bound to.

Now when the general idea is laid out, we are ready to study the proposed garbage collector strategy in more detail and how it can be implemented. As the starting point of the collector we will use an engine based on Brook's copying garbage collector algorithm. We furthermore assume that the real time system contains only one high priority process. The number of low priority processes is not restricted, however.

The heap is divided into two areas, the *Fromspace* and the *Tospace*, as shown in figure 2. New objects are allocated in Tospace beginning from the upper end (right side in the figure). We use the symbol A to denote the amount of newly allocated objects in Tospace. As the garbage collector runs, it evacuates objects from Fromspace and puts them in Tospace beginning from the lower end of

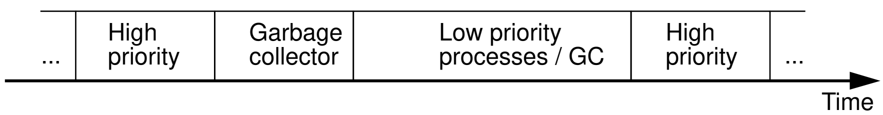

Figure 1 *How to divide the CPU time between the different processes.*

Tospace (left side in the figure). We denote the amount of evacuated objects at any one time E.

The evacuation is done incrementally. A *forwarding pointer* in each of the objects is used during pointer dereferencing. The forwarding pointers of the Fromspace copy of the objects point to the corresponding, evacuated, objects in Tospace. The forwarding pointer within a Tospace object points to the object itself. All references to objects are done via the forwarding pointer. This enables the program to always address the most recent copy of an object.

Evacuation is done by linearly scanning the pointers in the evacuated objects, evacuating referenced objects in Fromspace. Pointer assignments are checked. If an attempt is made to store a pointer referencing an object in Fromspace, the object is also evacuated. This is done to make sure we don't introduce any pointers into Fromspace in the already scanned objects. It also means that we don't have to scan the newly allocated objects. When all objects have been evacuated, we can temporarily suspend the garbage collector until Tospace is filled up. At that point, the functions of Fromspace and Tospace are swapped, and a new collector cycle begins. This is called a *flip*.

To guarantee that the system doesn't run out of memory, we must make sure that the evacuation of live Fromspace objects will be finished before the point in Tospace where new objects are allocated reaches the point where evacuated objects are stored. That can be avoided if we know how much of the Fromspace that consists of live objects, or more realistically an upper bound on the amount. The garbage collection work can then be scheduled as a function of A in a way such that all live objects are evacuated to Tospace before the system runs out of memory. One way to find out how many objects have to be evacuated is to have the application programmer specify an upper bound for the amount of objects that can be alive at any one time. The maximum amount of live objects is also the maximum amount of objects that may have to be evacuated. We use E_{max} to denote this upper bound. If all Fromspace objects are evacuated before the area denoted F in figure 2 becomes exhausted, the system will not run out of memory.

4.2 Description of the algorithm

In order to describe the proposed strategy, let us follow the execution of the application from the point at which the high priority process is invoked, through the execution of the garbage collector process and the low priority processes

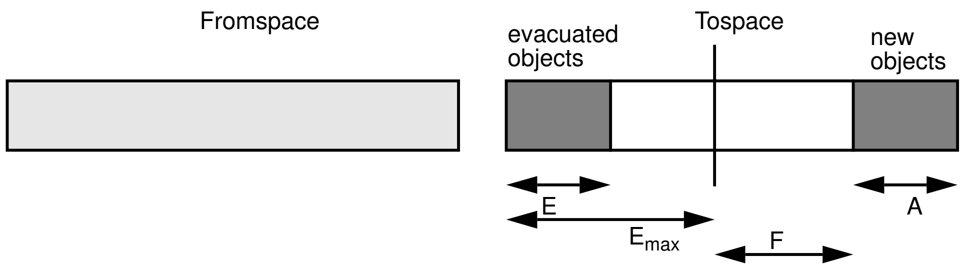

Figure 2 *Heap usage.*

until the high priority process is started next and the cycle starts all over again. See also figure 1.

In the first phase of the cycle, i.e when the process with high priority is running, no garbage collection is to be performed. In order to guarantee this, the size of the free memory, F in figure 2, must be at least as large as the worst-case allocation need of the program during one invocation of the high priority process. This parameter must be specified by the application programmer. We will later see how the garbage collector system makes sure that this condition always holds.

In order to guarantee that no Fromspace pointers are introduced into scanned objects, pointer assignments have to be checked and objects evacuated if needed. Since we have forbidden garbage collection induced delays during the execution of the high priority processes, it is advisable to delay this work until the process has finished. The addresses of all pointers that are modified during the first phase are thus pushed onto a stack to be checked later. When the high priority process blocks, the pointer assignment stack is traversed and objects are evacuated as necessary. Here we pay some of the price of not having the high-priority process disturbed by garbage collection.[1]

After having emptied the pointer assignment stack, an amount of time remains before the high priority process is to be started again. This time slice is to be divided between the garbage collector and the low priority processes. Varying strategies can be used to do this, but they must all guarantee that we don't run out of free memory, F, before all live objects in the Fromspace has been evacuated and we can do a flip.

A trivial way to divide the time slices is to devote them entirely to garbage collection until all objects in Fromspace have been evacuated. After this, the garbage collector is suspended until Tospace is filled up, after which we do a flip and a new collector cycle starts by again using all available time to evacuate Fromspace objects. The behaviour of the system from the view of the low priority processes would be similar to the stop-and-compact type of garbage collectors.

In practice, however, a more even distribution of the garbage collector work is probably desired. By using our knowledge about the maximum amount of objects that may have to be evacuated from Fromspace after a flip, we can make a plan at the time of the flip of how much objects have to be evacuated when we have allocated a given amount of new objects. We calculate a garbage collector/allocation rate. When the system gets behind this planned schedule, the time left over after running the high priority process is devoted to garbage collection until the evacuation of objects catches up with the allocation of new objects.

The two above described approaches will be studied below in a more formal way and we will see that it will in both cases be possible to work out garbage col-

1. The cost of the proposed strategy, as seen from the point of the high priority process, is practically equal to the cost of manually managing the memory. The cost of a pointer assignment is increased only with having to push the address of the pointer onto a stack, whereas memory allocation is actually cheaper than if a system based on free-lists would be used. In previous approaches to automatic memory management for real time systems, each individual pointer assignment and object allocation could trigger a garbage collection induced delay in the order of 10 milliseconds.

lection parameters and conditions to guarantee that the system doesn't run out of memory or disturbs the high priority process.

5 Analysing the Strategy

In this section we will make a more formal analysis of the proposed strategies. To do this we will need to have a notation describing the involved entities. This section will therefore start by introducing such a notation. Next, we will make an analysis based on the assumption that garbage collection is done in batch with respect to the low-priority processes. In the last part of the section we will sketch on a simple strategy of how the garbage collector work can be distributed more evenly.

5.1 Notation

The notation needed for our analysis is shown in the table below. It consists of symbols for application parameters given by the programmer, sizes of varying memory areas, and times needed to perform different tasks. The meanings of some of the symbols are also illustrated graphically in figure 2 and figure 3.

Symbol	*Description*	*Figure*
p	Period for the process with high priority, i.e the time between two invokations of the process. Specified by the programmer.	3
t_{max}	Maximum time required by the high priority process during one period. Specified by the programmer.	-
a_{max}	Maximum amount of memory that can be allocated by the high priority process during one period. Specified by the programmer.	-
pa_{max}	Maximum number of pointer assignments that can be performed by the high priority process during one period. Specified by the programmer.	-
S	Size of Tospace.	-
E	Amount of currently evacuated objects.	2
E_{max}	Worst-case amount of objects that may have to be evacuated.	2
A	Amount of newly allocated objects.	2
F	Amount of free memory that can be used to allocate new objects.	2
L	Time remaining after having run the high priority process and cleared the pointer assignment stack until the high priority process is to be started again.	3
L_{min}	Minimum value of L, i.e worst-case behaviour.	-

Symbol	Description	Figure
t_{Evac}	Worst-case remaining time needed by the garbage collector to finish the evacuation of Fromspace.	-
c_{Evac}	Number of periods it may take to finish the evacuation of Fromspace given that all available time is allotted to the garbage collector .	-
t_{GC}	Length of the time slice assigned for garbage collection during one period.	3
t_h	Time used by the high priority process during the current period.	3
t_{Copy}	Maximum time to evacuate one object, i.e to copy it.	-
t_{low}	Time remaining for the low priority processes to run before it is time to restart the high priority process.	

5.2 Batch-like Garbage Collection

A trivial way to divide the spare time between the garbage collector and the low priority processes is, as we mentioned earlier , to devote all available time to the garbage collector until the Fromspace is empty . The low-priority processes are then allowed to run, interrupted of course by the high priority process, until Tospace fills up. When this happens, we do a flip and the process starts all over again. What does this strategy require to succeed? We will calculate the worst-case load and derive necessary conditions for the system not to run out of memory or having to disturb the high priority process.

We start by calculating the minimum amount of time available after having run the high priority process:

$$L_{min} = p - t_{max} - pa_{max} \cdot t_{Copy}$$

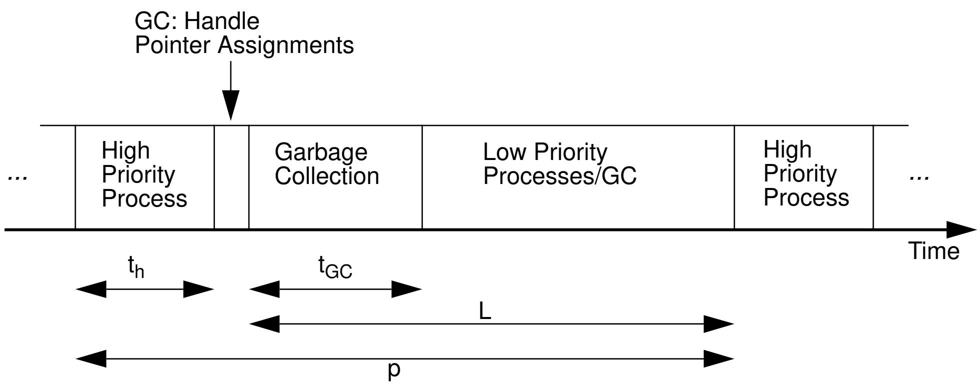

Figure 3 *Denoting time intervals.*

The time required to finish the evacuation is t_{Evac}. To simplify the analysis, we skip how this is calculated. It is heavily dependent on the details of the actual algorithm used, but in our case it is a function of how many live objects that are present, how many pointers they contain, the time it takes to evacuate an object, the time it takes to scan an object, and so forth. Since we don't know some of these parameters, a worst-case estimation has to be done.

Having calculated t_{Evac}, we can calculate the maximum number of periods needed to complete the garbage collection cycle, assuming that all spare time is allotted to the garbage collector. This number is denoted by c_{Evac}[1]:

$$c_{Evac} = \lceil t_{Evac} / L_{min} \rceil$$

During the time the evacuation of Fromspace is finished, the high priority process can be invoked c_{Evac} times. It may then in the worst case allocate $c_{Evac} \cdot a_{max}$ bytes of memory. This in turn means that F must always be at least as large as $c_{Evac} \cdot a_{max}$ in order for us to avoid running out of memory. F is calculated as follows:

$$F = S - E_{max} - A$$

The value of A directly after a flip is zero and we can thus set up the following inequality which, if satisfied immediately after a flip, tells us that the strategy will succeed:

$$S - E_{max} \geq c_{Evac} \cdot a_{max}$$

If the two sides of the equation are equal, the low priority processes cannot be guaranteed to get an opportunity to run. The larger the left-hand side of the equation is compared to the right-hand side, the more time will be available to the processes with low priority. In practice however, the low priority processes will get plenty of opportunity to run even in the case of equality, since the calculation of c_{Evac} is based on extremely pessimistic assumptions. It is also very pessimistic to assume that the high priority process will allocate a_{max} bytes every time it is invoked.

5.3 Distributing the Garbage Collection Work

The analysis of the strategy described in the previous section showed that it is possible to schedule the garbage collection work in such a manner that the high priority process isn't disturbed. We did, however, probably not use a very practical strategy due to the fact that the low priority processes are completely blocked for an extended period of time. In a real world application, the low priority processes must probably be able to run more smoothly than that. We must then interleave the execution of the garbage collector better with the low priority processes. A simple approach to doing this is described below.

1. The symbols $\lceil\ \rceil$ in the expression mean that the enclosed value is rounded upwards to the next integer bigger or equal to the given value.

We observed in our analysis of the scheduling algorithm that if the amount of free memory was larger than the worst-case need of the high priority process during the evacuation phase, we were able to guarantee that the low priority processes would run. Why not use this knowledge to do the garbage collection work in a more slow pace?

When the garbage collector is invoked after the high priority has blocked, we start as before, by emptying the pointer assignment stack, evacuating referenced objects if necessary. Now, L time units remain before the high priority process is to be invoked again. This amount of time is to be allocated to garbage collection and running low priority processes. To decide how this is to be done, we look at how much garbage collector work remains in the worst case. We calculate t_{Evac} as in the previous section, but in doing so we take into account the amount of objects already evacuated. When we have already concluded that all live objects in Fromspace have been evacuated, t_{Evac} will be zero. As before, we do not perform a flip until Tospace fills up, i.e when $S\text{-}A\text{-}E<a_{max}$. The minimum amount of periods, c_{Evac}, required to perform the collector work is as before:

$$c_{Evac} = \lceil t_{Evac} / L_{min} \rceil$$

The actual number of periods, ac_{Evac}, remaining before we have to do a flip is however given by dividing the amount of free store, F, with the worst-case need of the high priority process during one period[1]:

$$F = S - E_{max} - A$$

$$ac_{Evac} = \lfloor F / a_{max} \rfloor$$

Attempting to distribute the time t_{Evac} evenly over these periods gives us the time to assign to garbage collection during the present period:

$$t_{gc} = t_{Evac} / ac_{Evac}$$

As long as ac_{Evac} is larger or equal to c_{Evac}, we can guarantee that the system doesn't run out of memory. The time remaining for low priority processes to run will be:

$$t_{low} = L - t_{gc}$$

It may seem that we ignore the fact that the low-priority processes also allocates memory. The calculations we have done up to now have only taken into account the memory allocation rate of the high priority process. The actual total allocation rate of the system is higher than that. To understand the solution to this problem we must look closer on how the garbage collector cooperates with the low priority processes.

What we in effect are doing when we calculate how to divide the time L between the garbage collector and the low-priority processes is that we make a

1. The symbols $\lfloor\ \rfloor$ in the expression mean that the enclosed value is rounded downwards to the next integer smaller or equal to the given value.

plan on how much collector work that has to be done given the current size of F. If the garbage collector gets behind this schedule, time is devoted to collecting garbage. When the high priority process runs we might get behind with the collector work and we thus devote some time to catching up. When the low priority processes run, they too might cause the collector to get behind. The low priority processes are then suspended and the garbage collector is again invoked to catch up. The lower limit on the size of F that the low priority processes are never allowed to pass is $c_{Evac} \cdot a_{max}$, since that is the minimum amount of free memory needed to conclude the evacuation of Fromspace without disturbing the high priority process. In order for the low priority processes to run smoothly, some time should be devoted to garbage collection even though we haven't yet reached the lower limit on F.

Another occasion when the garbage collector is to be invoked is when there are no other processes at all that want to run. Instead of wasting CPU cycles, we take the opportunity to do some garbage collection work in order to get ahead of the schedule. This does not improve the worst-case behaviour of the algorithm, but it is an obvious way of improving the practical, average-case, behaviour of the system. In a real-world system, this optimization will probably lead to that the garbage collector almost always is ahead of the schedule. The garbage collector will thus intrude minimally on the low priority processes.

6 Future Work

The work described in this paper is at the time of writing in a very early stage, so much work remains to be done in the discussed field. The very first thing that should be done is to implement the described scheduling strategy and to verify that it actually works in practice. A prototype implementation will be made within the Mjølner/Orm programming environment [M+90], making it possible to use the system to develop real time applications. The performance of applications using the described strategy will then be investigated. A traditional incremental garbage collector based on Brook's algorithm is already included in the Mjølner/Orm system. We might also get the opportunity to test the strategy within a real commercial system in cooperation with the local process industry.

An observation that can be made about the scheduling strategy described in this paper is that individual worst-case delays are not as central when evaluating the algorithm as they are when using traditional, allocation-driven, variants. Since the garbage collector work is done in larger continuous chunks, the average-case behaviour becomes more interesting. One garbage collector architecture with very good average-case performance is the generation-based architecture. This idea has been described by [LH83] and [Ung84] among others.

In collectors of this type, the heap is divided into several areas, or generations, containing objects of different ages. Since research has shown that the mortality for objects is highest for young objects, most of the garbage collector work can be confined to the youngest generation, avoiding evacuation of the bulk of the live objects, which are located in the older generation(s). The worst-case behaviour, i.e worst-case delay, of these algorithms is not as good as for algorithms using only one generation, however. In spite of this, it seems reasonable that a generation-based algorithm would be suitable when scheduling the garbage collector work according to the ideas presented in this paper. We believe this since the collector is running for longer continuous periods and since it doesn't rely on having small-enough individual delays. The scheduling algorithm must however be modified to suit this type of collector. An implementation would be necessary in this case also and a variant of the application program runtime system of the Mjølner/Orm system already today contains a collector based on two generations as described in [Ben90], which would be suitable for experiments.

Another field of research will focus on the problem of many concurrently running processes with high priority. However, this doesn't seem to be a too difficult problem, but it should nevertheless be investigated and formalised.

The strategy presented in this paper take advantage of the fact that the high priority processes of many real time systems are clock-driven, or periodic, and that it is possible to predict when they are to be invoked next. Knowing this, it is possible to adapt the garbage collector parameters in such a way that the system never runs out of memory. If the high priority processes instead are driven by hardware interrupts caused by external events, this will be much more difficult to do. Future work could therefore be aimed at studying systems of this kind and adapting the ideas presented in this paper to suit the new demands.

7 Summary

Due to the high demands on real time systems to be reliable, automatic memory management is needed. Traditional approaches to real time garbage collection have, however, had trouble with guaranteeing short enough worst-case response times for the system as a whole. By using knowledge of the design of real time applications, some of the problems can be solved. This paper advocates dividing the processes of the real time application into two groups; those with high real time requirements and those with lower demands. Suspending garbage collection entirely when the high priority processes are running leads to extremely good performance for these processes. The cost of this scheme is paid by the processes with lower priority, which executes in a somewhat more jerky fashion. The fact that the high priority processes are periodic makes it easy to schedule the garbage collector in a way such that the system do not run out of memory. This also implies that the garbage collector should be closely coupled to the process scheduler in real time systems of this type.

Acknowledgements
Much of the credit of the strategy described above should be given to my supervisor, Boris Magnusson, who came up with the original idea and who has supported me in writing this paper. I would also like to thank the rest of the members of the programming environment research group at Dept. of Computer Science, Lund University for their support.

References

[Bak78] Henry G. Baker. List Processing in Real Time on a Serial Computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[Ben90] Mats Bengtsson. *Real-Time Compacting Garbage Collection Algorithms*. Licentiate thesis, Department of Computer Science, Lund University, 1990.

[Bro84] Rodney A. Brooks. Trading Data Space for Reduced Time and Code Space in Real-Time Garbage Collection on Stock Hardware. In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming*, August 1984.

[EV92] Steven L. Engelstad and James E. Vandendorpe. Automatic Storage Management for Systems with Real Time Constraints. In *OOPSLA '91 Workshop: Garbage Collection in Object Oriented Systems*, OOPS Messenger, October 1992. Position paper.

[LH83] H. Lieberman and C. Hewitt. A Real-Time Garbage Collector Based on the Lifetime of Objects. *Communications of the ACM*, June 1983.

[M+90] Boris Magnusson et al. An Overview of the Mjølner/Orm Environment. In *Proceedings of TOOLS 90 (Technology of Object-Oriented Languages and Systems)*, June 1990.

[Ung84] David Ungar. Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, May 1984.