JUNE 1991

WRL Research Report 91/8

Incremental, Generational Mostly-Copying Garbage Collection in Uncooperative Environments

G. May Yip

digital **Western Research Laboratory** 250 University Avenue Palo Alto, California 94301 USA

The Western Research Laboratory (WRL) is a computer systems research group that was founded by Digital Equipment Corporation in 1982. Our focus is computer science research relevant to the design and application of high performance scientific computers. We test our ideas by designing, building, and using real systems. The systems we build are research prototypes; they are not intended to become products.

There is a second research laboratory located in Palo Alto, the Systems Research Center (SRC). Other Digital research groups are located in Paris (PRL) and in Cambridge, Massachusetts (CRL).

Our research is directed towards mainstream high-performance computer systems. Our prototypes are intended to foreshadow the future computing environments used by many Digital customers. The long-term goal of WRL is to aid and accelerate the development of high-performance uni- and multi-processors. The research projects within WRL will address various aspects of high-performance computing.

We believe that significant advances in computer systems do not come from any single technological advance. Technologies, both hardware and software, do not all advance at the same pace. System design is the art of composing systems which use each level of technology in an appropriate balance. A major advance in overall system performance will require reexamination of all aspects of the system.

We do work in the design, fabrication and packaging of hardware; language processing and scaling issues in system software design; and the exploration of new applications areas that are opening up with the advent of higher performance systems. Researchers at WRL cooperate closely and move freely among the various levels of system design. This allows us to explore a wide range of tradeoffs to meet system goals.

We publish the results of our work in a variety of journals, conferences, research reports, and technical notes. This document is a research report. Research reports are normally accounts of completed research and may include material from earlier technical notes. We use technical notes for rapid distribution of technical material; usually this represents research in progress.

Research reports and technical notes may be ordered from us. You may mail your order to:

Technical Report Distribution
DEC Western Research Laboratory, WRL-2
250 University Avenue
Palo Alto, California 94301 USA

Reports and notes may also be ordered by electronic mail. Use one of the following addresses:

Digital E-net:	`DECWRL::WRL-TECHREPORTS`
Internet:	`WRL-Techreports@decwrl.dec.com`
UUCP:	`decwrl!wrl-techreports`

To obtain more details on ordering by electronic mail, send a message to one of these addresses with the word ‘‘`help`’’ in the Subject line; you will receive detailed instructions.

Incremental, Generational Mostly-Copying Garbage Collection in Uncooperative Environments

G. May Yip

June , 1991

digital **Western Research Laboratory** 250 University Avenue Palo Alto, California 94301 USA

INCREMENTAL, GENERATIONAL MOSTLY-COPYING GARBAGE COLLECTION IN UNCOOPERATIVE ENVIRONMENTS

by

G. May Yip

B.S. Computer Science, MIT (1990)
B.S. Electrical Engineering, MIT (1990)

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL

ENGINEERING AND COMPUTER SCIENCE IN PARTIAL

FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

AT THE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1991

Signature of Author _____
Department of Electrical Engineering and Computer Science
May 10, 1991

Certified by _____
David K. Gifford
Faculty Thesis Advisor

Certified by _____
Joel F. Bartlett
VI-A Company Supervisor

Accepted by _____
Arthur C. Smith
Chairman, Graduate Thesis Committee

INCREMENTAL, GENERATIONAL MOSTLY-COPYING GARBAGE COLLECTION IN UNCOOPERATIVE ENVIRONMENTS

by

G. May Yip

Submitted to the Department of Electrical Engineering and Computer Science
on May 10, 1991 in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

The thesis of this project is that incremental collection can be done feasibly and efficiently in an architecture and compiler independent manner. The design and implementation of an incremental, generational mostly-copying garbage collector for C++ is presented. The collector achieves, simultaneously, real-time performance (from incremental collection), low total garbage collection delay (from generational collection), and the ability to function without hardware and compiler support (from mostly-copying collection).

The incremental collector runs on commercially-available uniprocessors, such as the DECStation 3100, without any special hardware support. It uses UNIX's user controllable page protection facility (`mprotect`) to synchronize between the scanner (of the collector) and the mutator (of the application program). Its implementation does not require any modification to the C++ compiler. The maximum garbage collection pause is well within the 100-millisecond limit imposed by real-time applications executing on interactive workstations. Compared to its non-incremental version, the total execution time of the incremental collector is not adversely affected.

Faculty Thesis Advisor:	David K. Gifford
Title:	Professor of Electrical Engineering and Computer Science
VI-A Company Supervisor:	Joel F. Bartlett
Title:	Computer Research Scientist, Digital Equipment Corporation

Dedication

To Mom and Dad, who are always loving and supportive. Not words alone can adequately express the kind of appreciation and love I have for you.

Acknowledgment

My company supervisor, Joel Bartlett, has been very helpful to me during my three VI-A internship assignments at DEC Western Research Laboratory. Bartlett is not just an excellent boss, giving me advice and support whenever I need guidance, but he is also a terrific friend. I have never had a boss who would fix my bike and listen to my complaints about landlord/tenant problems. He is the kind of boss who is not just concerned with work performance, but is also concerned with other aspects of a summer intern's life. Bartlett debugged early drafts of this thesis and gave me useful feedback which improved the presentation of the final version significantly.

I would like to thank my faculty thesis advisor, David Gifford, who has been an active advisor even though he was at MIT, and I was doing my thesis project some three thousand miles away. He read my e-mail messages carefully and provided me with questions to further investigate. Gifford helped improve the presentation of this thesis. Thank you very much.

I would also like to thank all my friends who have helped preserve my sanity during my MIT days. They are the ones with whom I've spent most of my nocturnal days -- eating Wing-It, doing problem sets, or talking about personal problems. I can't imagine what campus life would be like without them.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction and Related Work

Garbage collection (GC) refers to the memory allocation and recycling mechanisms for the application program's data memory area. GC has always been associated with high-level, symbolic-processing languages such as Lisp and Scheme, where the concept of automatic storage management for objects of indefinite extent is embedded in the language. However, in other high-level languages, garbage collection has not gained wide acceptance because the extra bookkeeping needed often imposes a significant performance penalty. Traditional garbage collection schemes, like mark-and-sweep and stop-and-copy, lengthen a program's total execution time and adversely affect its interactive performance.

Although copying and generational collection algorithms applied to high-level languages are hardly new ideas [Fenichel&Yochelson 69] [Moon 84], garbage collection in C++ seems to be an odd idea on first sight. In C++, contrary to Lisp and Scheme, the storage for objects of indefinite extent must be explicitly managed. But the importance of garbage collection in C++ is increasing, because as programs become more complicated, GC can help alleviate the complexity in storage management. Efficient garbage collection techniques present an attractive alternative to the usual low-level, ad hoc approach to storage management, where object deallocation is explicitly managed. Indeed, advances in garbage collection technology have made possible the advent of efficient collectors that do not rely on special purpose hardware or compiler support.

Garbage collectors that can operate on various architectures and system environments contribute substantially to the language system's portability and interoperability. One example is Xerox PARC's Portable Common Runtime [Weiser 89]. It is a portable, multi-lingual programming environment where the support for threads and garbage collection is provided for all languages and is built in as part of the Common Runtime, rather than the individual language runtimes. PCR's collector is a parallel and non-copying storage manager that must be used as part of the runtime package. Although PCR is portable across many operating systems, it does have a fair amount of CPU specific code. My objective in this project, however, is to build a collector that can run in existing environments; a standalone, incremental, and copying collector that can be used with any C++ compiler the programmer desires.

This thesis work is based on the work done at Digital Equipment Corporation's Systems Research Center and Digital Equipment Corporation's Western Research Lab. Appel, Ellis and Li [Appel et al. 88] demonstrate that incremental garbage collection is feasible using standard virtual memory page protections, without a tagged architecture or additional hardware support. To detect pointers to from-space objects, a medium grain synchronization is established between the collector and the mutator at the virtual memory level. Bartlett invented mostly-copying collection, which makes substantial memory compaction possible without having to know the actual set of root pointers at the start of the collection [Bartlett 88]. The result is that a mostly-copying collector can perform in an environment where stack and register allocation disciplines are not known. Additionally, Bartlett has incorporated generational collection into his mostly-copying collector [Bartlett 89], and has shown that his strategy works well in systems running Scheme, C, and C++.

Detlefs has already attempted to combine Appel et al.'s and Bartlett's ideas into building a concurrent collector for C++ [Detlefs 90]. However, Detlefs's collector is not generational and not very portable. It requires modifications to be made to the C++ compiler and requires MACH. My thesis is that incremental, generational mostly-copying collection can be done efficiently in a compiler and architecture independent manner, without any modification made to the underlying operating system. By combining and expanding on ideas employed in previous work, I have built such a collector.

1.1 Mostly-Copying Collection

Bartlett's mostly-copying collector is a compacting, conservative collector that has knowledge about the heap structure but does not need compiler support. *Conservative* means that the collector must assume that any value on the stack or in the registers that could be a pointer is a pointer. While this may retain objects that could otherwise be collected, it has the advantage that the collector does not need to know everything about the stack and registers. *Compacting* means that objects that can be safely relocated are moved into contiguous spaces. There are two benefits of compacting collection: (1) heap fragmentation can be controlled, so that the working set of physical pages occupied by the program in the virtual space is reduced, therefore lowering the overhead for virtual memory paging; and (2) compacting collector's execution time is proportional to the amount of space retained, and not proportional to the heap size.

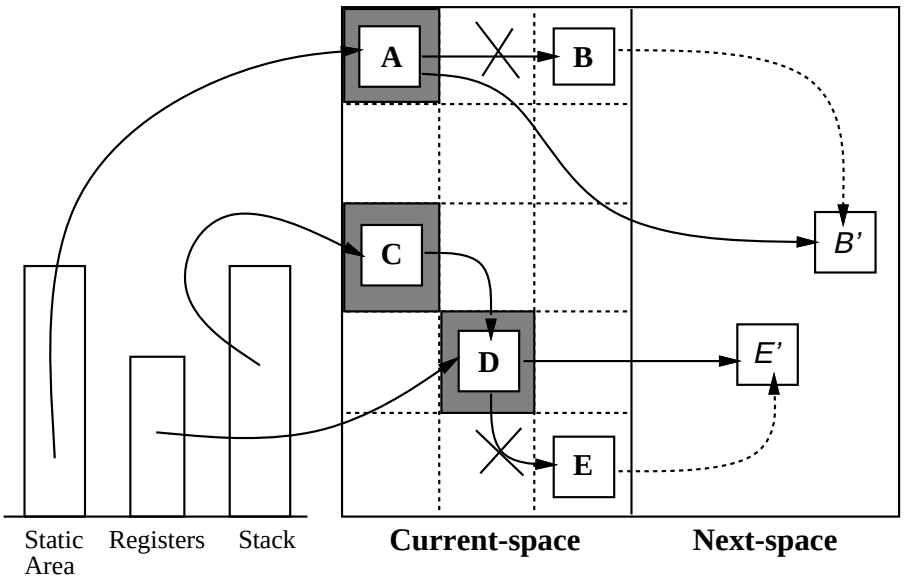


Figure 1-1: Bartlett's Mostly-Copying Collector

As shown in figure 1-1, Bartlett's collector is based upon an old-space/new-space approach. The heap is divided uniformly into a number of pages (called heap pages) whose size is independent of the actual physical page size of the underlying paging system. *Current-space* and *next-space* are conceptual counterparts of *from-space* and *to-space*, respectively, of the classical stop-and-copy collector. Note however, that the spaces here are not necessarily contiguous regions in memory. There is a special space identifier associated with each heap page to indicate the space it is in. In the simplest term, mostly-copying collection proceeds as follows. First, the collector ''guesses'' which heap pages contain objects that may be referenced from pointers in the processor stack, registers and the application program's static area. These pages are *promoted* to the next-space. In the figure, this refers to the pages containing objects A, C and D. Promoting a page means that the special space identifier of the page is flagged such that the page will be retained when the collection is over. The objects that point to A, C and D are called *ambiguous roots*, because not necessarily all of them are actually real pointers, but they contain the root objects through which all accessible objects can be traced. Because of this uncertainty, ambiguous roots cannot be changed. It is important that the heap pages pointed to by these ambiguous roots be ''locked,'' so the objects on these pages can be retained.

Once the initial promoted objects have been identified, the collector *scans* the

ambiguous root objects and *forwards* all objects referenced from these roots into a more compact area in next-space. Scanning refers to the process by which internal pointers (if any) of an object are examined; and forwarding refers to the copying of the objects referenced by these internal pointers into next-space. The forward pointers of the copied objects are updated in the object being scanned. As the figure illustrates, object B is therefore copied into B', and the old pointer in A which pointed to B is updated to point to B' instead. The same is also true for object E pointed to by object D. On the other hand, although D is referenced by object C, D is not forwarded because the heap page containing D is already a promoted page. Scanning continues until all promoted and forwarded objects have been scanned. GC is then complete.

In contrast with a conservative mark-and-sweep collector, which also makes ''guesses'' on stack and register pointers but leaves all retained storage fragmented, the mostly-copying collector is able to compact most of the heap because *it has knowledge about the heap structure.* By configuring the heap into a set of pages, the collector can preserve pointers in the root set whose values cannot be changed during a collection by simply promoting their corresponding page. It is then possible to compact other objects that are not in the root set.

1.2 Generational Collection

Bartlett has also incorporated generational collection into his mostly-copying collector. Generational collection works on the basis that newly created objects have the highest probability to be destroyed soon, while old objects that have survived collection(s) have a tendency to remain around for a longer time. Such observation is in line with the life times of (i) subroutine local variables, which are created when the subroutine is called and are immediately abandoned as soon as the subroutine exits; and (ii) entries in a large database, which are generally long-living objects.

Generational collection improves efficiency because repeated copying of retained objects is avoided. Bartlett's generational collector differentiates objects as either ''young'' or ''old.'' Each time garbage collection is invoked only pages containing young objects are collected. Old objects are retained and they are not scanned unless they have been mutated (or the collector thinks that they have been mutated) since the last time they were scanned. Old objects are not collected until more than a certain portion of the heap is allocated.

The mostly-copying algorithm facilitates generational collection as the space identifier of each heap page can be used to approximate the age of the object. In the current version, an even page space identifier represents an ‘‘old’’ page, and an odd one represents a ‘‘young’’ page. According to [Bartlett 89], when running with a generational mostly-copying collector, smaller programs that do not use garbage collection run a bit slower because of the overhead needed to keep track of mutated old objects, but large batch programs such as the Scheme compiler run faster, and interactive programs have shorter pauses during collection. Generational collection is worthwhile whenever the additional time needed to manage the remembered set -- the set of ‘‘old’’ pages that have references to new pages -- can be offset by the reduction in garbage collection time.

1.3 Incremental Collection

Traditionally, incremental collection is implemented on a tagged architecture with special hardware support, e.g. Symbolics Lisp Machines. For conventional incremental stop-and-copy collection, every pointer fetched from memory is examined to see if it points to a from-space object. If so, then just as in the case of a non-incremental collector, the from-space object is copied to to-space and the pointer which points to the object’s old location is updated. This hardware approach implements a fine grain synchronization between the mutator and the collector.

Appel, Ellis and Li have a different approach to incrementalize stop-and-copy collection. Using the standard virtual memory page protection mechanism, a medium grain synchronization is achieved: a virtual *page* in to-space is not scanned until an *object* within it is referenced. Their strategy is based upon the following invariants:

- At all times, newly-allocated objects contain to-space pointers only.
- At all times, user application accesses objects via to-space pointers only.
- During collection, scanned objects contain to-space pointers only.
- During collection, unscanned objects contain both from-space and to-space pointers.

Figure 1-2 illustrates the basic framework. When garbage collection is initiated, the root set is identified and the objects pointed to by the root set are copied to to-space. But these forwarded root objects in to-space may still have references to from-space objects, and

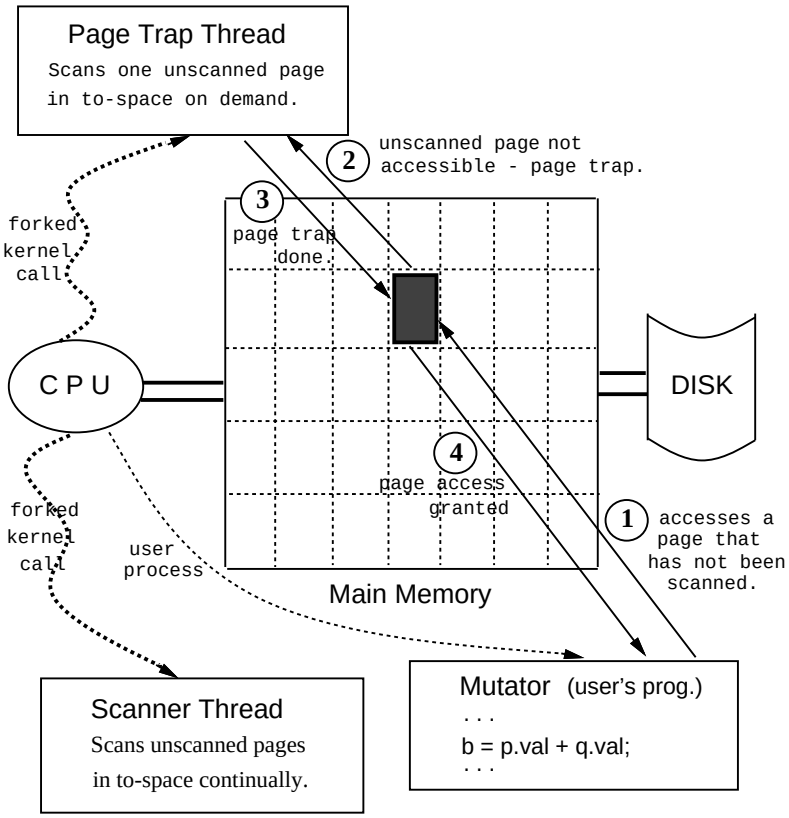


Figure 1-2: Appel, Ellis and Li's Incremental Collector

therefore must be scanned. Garbage collection is incrementalized by protecting unscanned pages in user mode. With incremental collection, an unscanned page in to-space can then be scanned when the mutator tries to access an object in it, by triggering a pre-established page trap thread running in kernel mode. After the page trap handler has scanned the page, the access mode of that page is adjusted so that execution of the mutator can proceed normally.

Appel et al. used this strategy to build a concurrent, real-time collector on the Firefly multiprocessor workstations. They added two kernel calls and use facilities particular to the Taos operating system. Taos extends Digital Equipment Corporation's ULTRIX[1] with virtual memory primitives, threads, and cheap synchronization [Thacker&Stewart 87].

[1]ULTRIX is a trademark of Digital Equipment Corporation.

The work presented in this thesis experiments with the application of this incremental collection strategy on the mostly-copying algorithm to build a real-time garbage collector on sequential workstations such as the DECStation 3100, running standard ULTRIX.

Chapter 2

Bartlett's Generational Mostly-Copying Collector

2.1 Programming Interface in C++

In languages such as Scheme and Lisp, which have their own native garbage collectors in the language system, objects with indeterminate extent are allocated in the heap and are garbage collected. In C++ there is no garbage collection facility native to the language system, and objects with indeterminate lifetime are given storage space by means of a general-purpose memory block allocation facility. Explicit storage management of these objects is necessary.

C++ is an object-oriented superset of C. It allows the application to define new object classes and provides compile time type checking for the classes. For each object class, the application supplies a constructor, a destructor, and creates other class operations. The constructor specifies how new objects are initialized. It is called when a new object is allocated. Class operations are procedures which clients of the object class use to manipulate objects. The destructor is called automatically when the object is going out of scope. Inside the destructor the application can free the memory occupied by the object. With Bartlett's generational mostly-copying garbage collector added to the C++ system, application programmers can decide which C++ classes are allocated on the garbage collected heap, and which are not. And for the garbage collected classes, the application does not need to provide a destructor.

Bartlett's collector is added to the language system like any other library because the collector is a self-contained module requiring no cooperation from the compiler. The application calls routines provided by the garbage collector by linking with the collector's library at compile time. This level of compiler independence allows the collector to be used with different C++ compilers.

Figure 2-1 shows an example of how a garbage collected C++ object class is declared. Only object classes with the special statement `GCCLASS` appearing in the class structure declaration are garbage collected. In the example, `GCCLASS` informs the garbage collector that the type `word` is garbage collected and has a ''pointer-finding'' callback method

```
struct  word  {
        word* lesser;
        word* greater;
        int   count;
        char  symbol[ 1 ];
        word( char* chars );
        GCCLASS( word );
};

word::word( char* chars )
{
        GCALLOCV( word, sizeof( word )+strlen( chars ) );
        lesser = NULL;
        greater = NULL;
        count = 1;
        strcpy( symbol, chars );
}

void  word::GCPointers( )  {
        gcpointer( lesser );
        gcpointer( greater );
}
```

Figure 2-1: C++ definition for the `word` object class

`word::GCPointers`. When the collector is scanning an object, it must be able to identify internal pointers of the object. In this case, the internal pointers of `word` are `word* lesser` and `word* greater`. In the `word::GCPointers` callback, there is a `gcpointer` statement for each internal pointer. During scanning, the garbage collector has an efficient way to call the `GCPointers` method of the object being scanned (described in the last paragraph of section 2.2.2, page 20). Then each of the `gcpointer` statements of the method passes an internal pointer of the object to the collector, so that *the object referenced by that internal pointer* can be forwarded. If an object class does not contain any internal pointer, the `GCPointers` method is an empty procedure.

Inside the constructor `word::word`, `GCALLOCV` is responsible for space allocation. The arguments of `GCALLOCV` are the class name and the number of bytes the class object occupies, respectively. `GCALLOCV` is designed for the allocation of variable-size objects; for an object class `t` whose size is known at compile time, a simpler statement, `GCALLOC(t)`, is used instead of `GCALLOCV(t,sizeof(t))`.

C++ supports class inheritance; it is possible that a collectible class is a subclass of a collectible superclass, or a non-collectible subclass has a collectible superclass, and vice versa. To ensure that the pointer-finding method is consistent and able to find all pointers to garbage collected objects, the application programmer must adhere to the following rules:

1. For class `C:P {}`, `C::GCPointers` must contain `P::GCPointers()`. That is, for a class `C` that is derived from a super class `P`, the pointer-finding method of `C` must contain the call to `P::GCPointers()`, which is class `P`'s pointer-finding method.
2. For class `C{X x;}`, `C::GCPointers` must contain `x.GCPointers()`. That is, for a class `C` which contains an object `x` of the garbage collected class `X`, `C`'s pointer-finding method must include the statement `x.GCPointers()`, which is the call to invoke the pointer-finding method of object `x`.
3. For class `C{X* x;}`, `C::GCPointers` must contain `gcpointer(x)`. That is, for a class `C` which contains a pointer `x` to an object of the garbage collected class `X`, `C`'s pointer-finding method must contain the statement `gcpointer(x)`, which informs the garbage collector about the existence of this internal pointer.

If a subclass does not have any internal pointers, then the `GCCLASS` statement in the declaration of the subclass can be omitted and the `GCPointer` callback method is not necessary.

2.2 Summary of Bartlett's Collection Algorithm

Bartlett's generational mostly-copying collector is a compacting, conservative collector. The heap is divided into a number of heap pages, each of which is `PAGEBYTES` bytes in size. `PAGEBYTES` is an adjustable parameter independent of the hardware page size. The collector conservatively treats all words in the processor stack and registers which could be pointers into the heap as root pointers. Heap pages which are referenced by these *ambiguous roots* are retained. Their contents are left intact because the ambiguous roots may reference their locations in the future. Other objects in the heap that are not referenced directly by the roots can be compacted, much like the way of a classical stop-and-copy collector.

Bartlett's collector uses a dual age group approach for generational collection. Newly-allocated objects are considered ''young'' while objects which have survived at least

one collection are ‘‘old.’’ Old objects are stable and not collected. During each collection, stable objects (which constitutes the stable set) that have been mutated to reference young, unstable objects have to be scanned, so that the unstable objects can be forwarded and made stable. The set of stable objects which the collector needs to scan is called the *remembered set*. For ease of implementation, the remembered set in Bartlett's collector is equal to the stable set, so no additional bookkeeping is necessary to remember the stable objects which have been mutated. Stable objects are retained at each collection, until the set of stable objects occupy a certain fraction of the heap. Then a total collection is initiated to collect all objects regardless of age.

2.2.1 Spaces

The classical stop-and-copy collector divides the heap into two contiguous regions, from-space and to-space. Objects are allocated in from-space until it is exhausted. Garbage collection is started, and objects that are salvageable are copied to to-space. When collection is over, the two spaces swap roles. Bartlett's collector divides the heap into two spaces also: current-space and next-space. They are analogous to from-space and to-space, respectively, of the classical algorithm; but Bartlett's spaces are not contiguous. Rather, each heap page has an associated space identifier to indicate the space it is in. There are two variable that the collector maintains, `curr_space` and `next_space`, to denote two important space identifiers. The following explains the partition of the spaces:

- *current-space* -- space where the application allocates heap pages. Heap pages in current-space have their space identifiers equal to `curr_space`.
- *next-space* -- space where retained heap pages are found during collection. The collector retains heap pages referenced by the ambiguous roots by setting (i.e. promoting) the space identifiers of those pages to be `next_space`. Heap pages allocated to hold forwarded objects also have their space identifiers equal to `next_space`.

Normally `curr_space` and `next_space` are equal when garbage collection is not going on. Initially, `curr_space` and `next_space` are set to 3. When garbage collection is initiated, `next_space` is incremented by 1, to become an even number. When garbage collection ends, `curr_space` is incremented by 2 (i.e. it remains odd), and `next_space` is reset to be equal to `curr_space`.

The heap pages promoted into next-space as well as those allocated in next-space

during collection always have even identifiers, while heap pages allocated by the application in current-space always have odd identifiers. Generational collection takes advantage of this fact and treats heap pages with even identifiers as the stable set.

2.2.2 Allocation

Allocation in the mostly-copying collector is a two-part process: (i) allocate a free heap page, and then (ii) allocate space from it. A new heap page allocated in step (i) always has its space identifier set to `next_space` (recall that `curr_space == next_space` when garbage collection in not going on). The variable `freewords` holds the number of remaining allocatable words in the current free page. If the remaining `freewords` number of words is insufficient for an allocation request, then the trailing space on the current free page is discarded and left unused. Another heap page is allocated, and `freewords` is reset to reflect the status of this new free page.

Each heap page has a type identifier associated with it. When a heap page is allocated to hold objects smaller than one heap page in size, its type identifier is set to `OBJECT`. To accommodate a large object, more than one heap page has to be allocated. The first of such heap pages is of type `OBJECT`, while the remaining one(s) are of type `CONTINUED`. The trailing space (if any) on the last `CONTINUED` page is not used for another object. It is simply left unused.

The mostly-copying collector can function without hardware and compiler cooperation because it has perfect knowledge about the heap and the objects inside it. Besides knowing the space and type identifiers of the heap pages, the collector needs to know specific information about each object. This is achieved by allocating a header word at the start of each object. The header word contains information about the object size and an index into an array of `GCPointers` callback routines. Using this index the collector is able to access the appropriate `GCPointers` routine for the object type, and it is through this `GCPointers` method that the internal pointers (if any) of an object can be located and examined.

2.2.3 Collection

The collector maintains the variable `allocatedpages` so that it knows how many heap pages have been allocated. Every time the collector enters the allocation routine, it checks the condition:

```
allocatedpages >= heappages/2 && curr_space == next_space
```

which when true, means that at least half of the heap is already allocated[2] and that collection has not been triggered yet. In this situation the collector enters garbage collection mode and attempts to recycle memory before getting to the allocation routine.

Garbage collection starts with the following assignment:

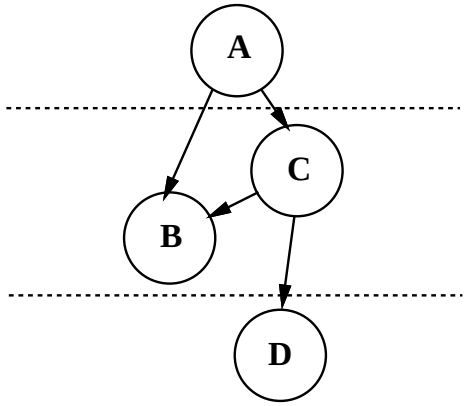
```
next_space = curr_space+1;
```

GC then proceeds with examining the stack, registers and static areas, looking for words which can be interpreted as pointers into current-space heap pages. These words are referred to as the ambiguous root pointers. The current-space heap pages which the ambiguous roots reference are promoted by changing their space identifiers to `next_space`. The page type identifier allows large objects to be recognized. When an ambiguous root points into one of the `CONTINUED` heap pages, all the heap pages which together make up the whole object must be promoted. When such an ambiguous root into an arbitrary `CONTINUED` page is encountered, it is easy to ‘‘search backward’’ to the beginning of the object by simply checking the type identifier of the page(s) preceding the `CONTINUED` page until a page of type `OBJECT` is discovered. After the leading `OBJECT` page is found, the header word gives the size of the object and the corresponding number of pages are promoted. If an ambiguous root references a stable heap page (one with an even space identifier), no promoting is necessary.

After the stack and registers are searched and promoting is done, the garbage collector sweeps across the set of stable and promoted heap pages and scans the objects inside them. Scanning is done using a breadth-first discipline. Referring to figure 2-2, object A is the first object to be scanned, and it contains pointers referencing objects B and C. Assume that all objects are in current-space before collection begins, and object A is promoted to next-space during collection. Scanning A discovers internal pointers to B and C (via the `GCPointers` method), and causes B and C to be forwarded, to B’ and C’ respectively, on another heap page allocated in next-space (see figure 2-3). As B and C are forwarded, their

---

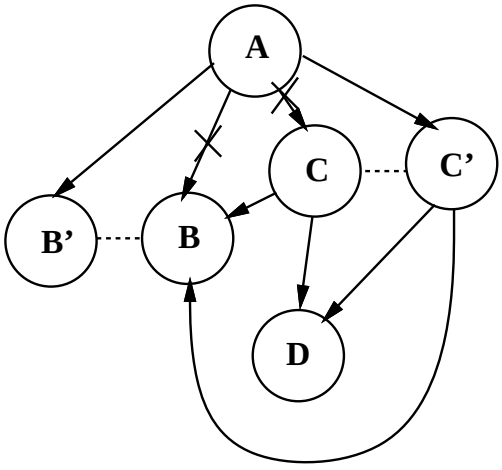
[2] `heappages` is the total number of heap pages in the heap.



The dotted lines show the ‘‘levels’’ of the data structure.

Scanning A causes B and C to be forwarded. Scanning C causes D to be forwarded (assuming that B is already forwarded).

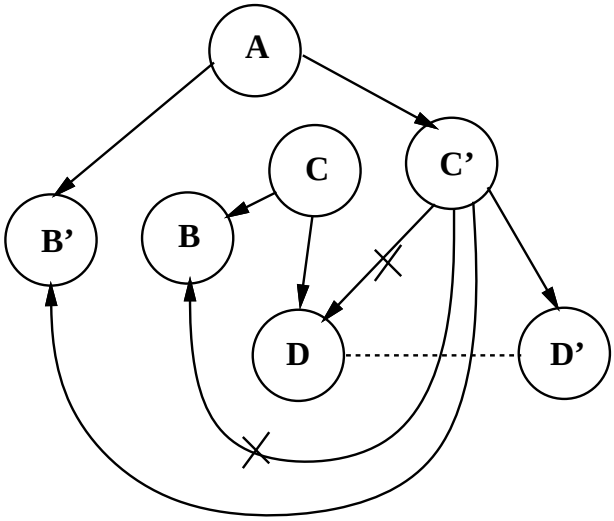
Figure 2-2: Scanning objects



Scanning A forwards B and C to B’ and C’, respectively.

Figure 2-3: Scanning objects - II

header words are replaced with the forward pointers to the location where the forwarded



Scanning C’ forwards D to D’.

Figure 2-4: Scanning objects - III

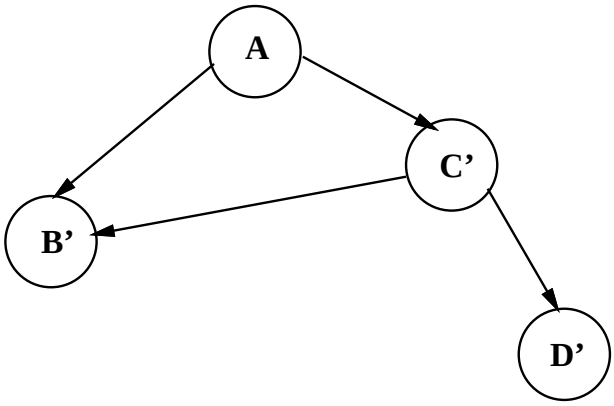


Figure 2-5: Scanning objects - IV

objects B’ and C’, respectively, are found. The flag bit of B’s and C’s header words are also set to indicate that these objects have been forwarded. The references to B and C found in A’s internal pointers are changed to reference B’ and C’ instead. Scanning of A is complete

after B and C are copied to B' and C'. Since B' and C' are in next-space, they are considered promoted objects and must be scanned. When B' is scanned, it does not cause any forwarding because it does not contain any internal pointers. When C' is scanned, the collector attempts to forward B and D (see figure 2-4). The flag bit in B's header shows that B has already been forwarded and therefore it is not forwarded again. The internal pointer in C is set to the value of B's forward pointer found in its header word. Afterwards, object D is forwarded the normal way -- assuming that it has not been already forwarded by another object in the heap. Figure 2-5 shows that as a result of scanning A, objects B, C and D are copied to B', C' and D', respectively, in next-space.

The collector does not attempt to forward objects that span more than one heap page. In the above example, objects B, C and D are assumed to be less than one heap page in size. For larger objects, the collector simply promotes all their heap pages to next-space, thereby saving the effort of having to find contiguous free heap pages to accommodate the large object and copying it.

During scanning, if a reference to a stable object is discovered, nothing needs to be done. Stable objects do not need to be forwarded or promoted because they are retained by the collector. With generational collection, heap pages in the remembered set are scanned also, in the same way that promoted heap pages are scanned.

Garbage collection is over when all the next-space and stable heap pages have been scanned. At this point all live pages have even space identifiers. These even pages constitute the stable set and are retained in the next collection. Heap pages in current-space can be recycled as free pages. Garbage collection is terminated by setting

```
curr_space = curr_space+2;
next_space = curr_space;
```

This way once again we have `curr_space == next_space`, just as it used to be before garbage collection was initiated. The space identifier `curr_space` is incremented by 2 so that the new heap pages to be allocated in current-space are distinguished from the reclaimed, used-to-be current-space pages, and from the retained stable heap pages.

Chapter 3

Incremental, Generational Mostly-Copying Collection

3.1 The Big Picture

The essence of incremental collection is to minimize garbage collection pauses by configuring the collector to scan only a bounded number of ‘‘live’’ objects at a time. After garbage collection is initiated, and after the necessary bookkeeping and setup work for incremental collection is completed, the application program can proceed normally even though garbage collection is not yet finished. While in the non-incremental scheme the application program can suffer a potentially long and detectable pause every time garbage collection takes place; under the incremental scheme the application program is interrupted intermittently to scan objects for a short duration of time. This way garbage collection is *incrementalized*, because the total work needed for garbage collection is divided into a number of comparable subtasks. The subtasks are spread out over time, each subtask being executed in a smaller time interval.

Incremental collection is synchronized at the granularity of physical pages. Incremental collection for applications running on commercially-available stock processors depends crucially on the virtual memory system’s provision to allow control for virtual memory page protections. At the start of incremental collection, the virtual memory pages on which objects referenced by the roots reside are read/write protected, i.e. they are neither readable nor writable. In the context of the mostly-copying strategy, this means that the *physical pages* containing one or more promoted *heap pages* are protected. The application is then allowed to resume as normal, while garbage collection is technically still "going on." When the application attempts to access a protected page, a virtual memory page fault is generated and program control is transferred to a special trap handler, which triggers the garbage collector to do all necessary work in order to make the physical page needed by the application accessible. To avoid the situation where the heap is exhausted by the application allocation before garbage collection is completed, the collector can randomly select a protected physical page to scan each time the application causes a new heap page to be allocated.

The GC work needed to be done at the page fault trap includes unprotecting the

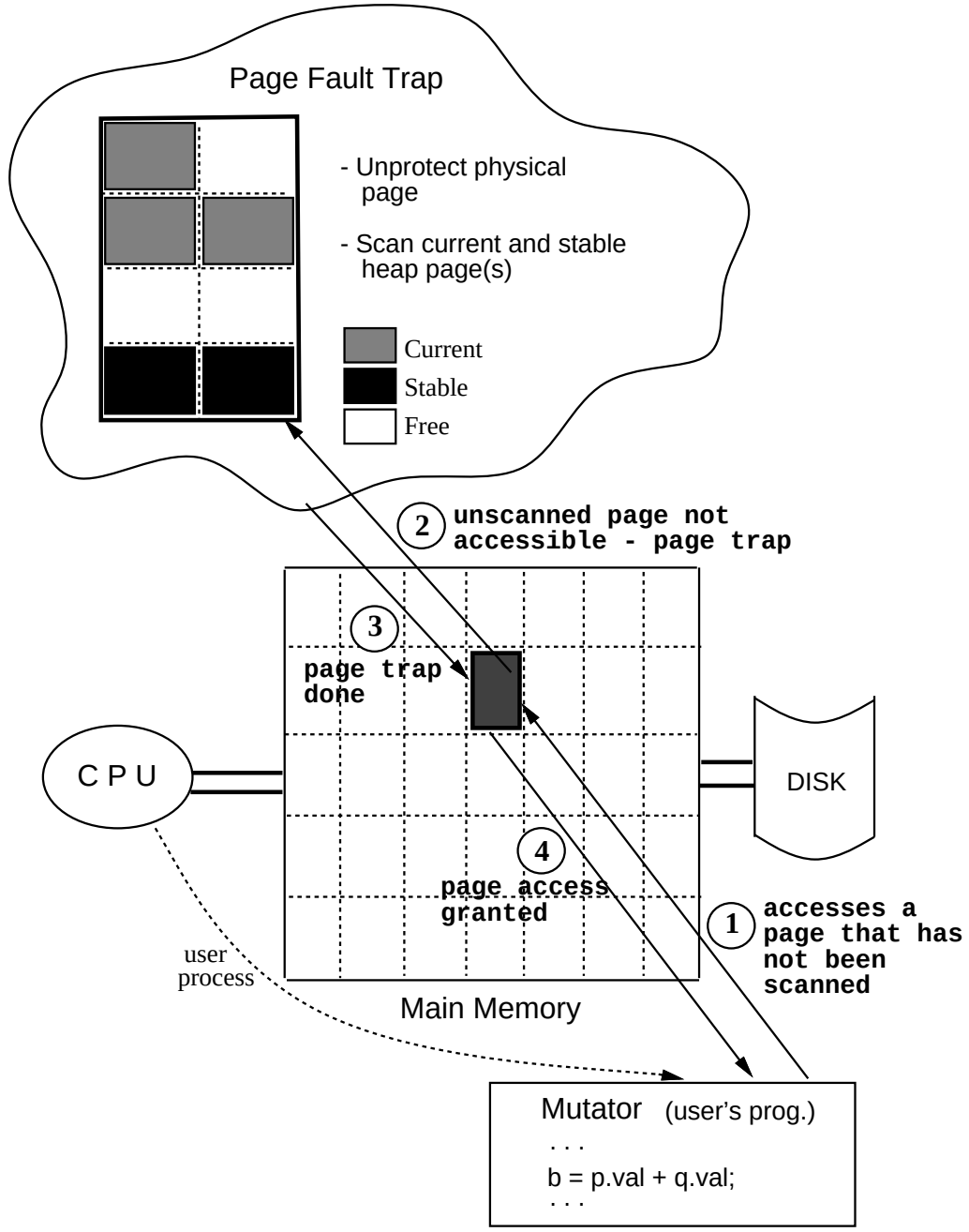


Figure 3-1: Incremental, Generational, Mostly-Copying Collection

faulted virtual page and scanning the promoted heap pages inside it.[3] This is termed ‘‘mostly-copying’’ collection because objects in the promoted heap pages are not moved. They are referenced by the ambiguous roots and therefore their contents cannot be changed. They are allowed to remain intact in the heap. Only objects not directly referenced by the root objects are copied into a more compact area in the heap, and their forward pointers are updated in the root objects appropriately.

When generational collection is added, not only do the *promoted* heap pages need to be scanned, heap pages in the remembered set have to be scanned also. The remembered set is the set of stable objects which have been mutated since the last time they were scanned. To avoid trapping writes into stable objects, this version of the generational collector in fact treats the entire stable set as the remembered set. Objects become stable once they have survived at least one collection. They have to be scanned because if they have been mutated since the last time they were scanned, they may contain pointers into newly-created (and hence unstable) objects. Similar to the scanning and forwarding strategy described in the previous paragraph, the unstable objects referenced by the stable ones are forwarded (and therefore become stable), and their forward pointers are updated in the stable objects. GC work on the physical page is complete after scanning and forwarding of promoted and stable objects are done. The physical page can be safely accessed by the application now, the trap handler can exit, and the application then resumes from the point where it was interrupted.

This process of faulting on a protected page, unprotecting it, and scanning and forwarding objects inside it repeats until there are no more protected pages left in the heap. Garbage collection is then complete, and the application executes with no interruptions until the next time garbage collection is initiated. Successive generational garbage collections enlarge the set of stable heap pages to be retained in the heap. When more than a certain fraction of the heap is retained after a generational collection is finished, a *total* collection will be carried out such that all live heap pages, whether they are ‘‘young’’ or ‘‘old,’’ are collected. In addition, the heap can be expanded, if it is discovered that more than a certain fraction of the heap is occupied after a total collection. Figure 3-1 illustrates this scheme for performing an incremental collection on a single-tasking stock processor utilizing a standard virtual memory system.

---

[3]For the moment consider only the case in which all the objects in the faulted page lie within the physical page boundary, that is, only the faulted page will have to be unprotected and scanned. A more elaborate scheme for dealing with bigger objects spanning across physical pages will be discussed in section 3.2.4.1.

The following section discusses how the incremental version of the generational mostly-copying garbage collector is implemented in more detail. In particular, the state-to-state transitions of the different stages of the collection algorithm are described, along with various special considerations that have to be taken into account in order to optimize performance.

3.2 An Incremental, Generational Mostly-Copying Collector

3.2.1 Special Considerations

To integrate Bartlett's generational mostly-copying collection algorithm into the incremental framework, a few special considerations have to be taken into account. Section 3.2.1.1 explains why the current-space/next-space partition of the heap in the non-incremental collector must be adjusted to a current-space/previous-space/forward-space partition. Section 3.2.1.2 states a new restriction on the choice of heap size and heap page size. Section 3.2.1.3 discusses the effects of the allocation algorithm on real-time performance. Section 3.2.1.4 describes how the miscellaneous page status information is handled.

3.2.1.1 Space Numbers

The current-space/next-space approach employed by Bartlett's collector works well in the non-incremental framework because application allocation for a particular value of `curr_space` is not intermixed with collector allocation for a particular value of `next_space`. This two-space framework is not sufficient for the incremental collector, because application allocation and collector allocation are intermixed. After garbage collection is initiated, all the heap pages that used to be in current-space and were not promoted are now occupying memory which the collector is trying to reclaim. But the memory cannot be reclaimed until GC is completely finished, because some of these objects may be referenced by objects in the promoted heap pages, i.e. objects in this region may still be salvaged. While the collector is intermittently salvaging objects in this otherwise reclaimable area, application allocation is going on. Therefore, the reclaimable area which used to be in current-space must be distinguished from the current-space in which the application is allocating new heap pages. A three-space approach is necessary because there are three kinds of heap pages to differentiate: the newly-allocated heap pages, the salvageable and used-to-be current-space heap pages, and the heap pages containing forwarded/promoted objects. The following is the modified space designation strategy designed for incremental collection:

- *current-space* -- space where newly-allocated heap pages are found. The application allocates in the current-space, by setting the space number of the allocated page to be `curr_space`.
- *previous-space* -- space where unscanned and (possibly) reclaimable heap pages are found. Heap pages which used to be in current-space before collection begins and which are not promoted after collection has been initiated are entered into previous-space (denoted as `prev_space`), by asserting that `prev_space` be equal `curr_space` at the start of collection, and subsequently advancing the value of `curr_space`.
- *forward-space* -- space where promoted heap pages and heap pages containing forwarded objects are found. The collector promotes pages into or allocates pages in forward-space by setting the space number of the allocated page to be `forw_space`.

Collection in progress is indicated by the inequality `curr_space != forw_space`. Current-space and forward-space in the three-space scheme are the same as current-space and next-space, respectively, in the two-space scheme. The addition of previous-space holds (temporarily) the used-to-be current-space objects, and allows application allocation to continue while previous-space objects are being scanned. At the end of collection, the collector will be able to reclaim the previous-space pages by simply changing the value of `prev_space` to be equal to `curr_space`.

3.2.1.2 Heap Size and Heap Page Size

The incremental collector performs GC work on the granularity of a unit of page protection. On the DECStation 3100, a unit of page protection is a physical page. On other machines, this may be different. Since it is undesirable to have the collector protect *any* part of memory that does not belong to the garbage collected heap, the heap size must be a multiple of the protection unit, meaning that for the DECStation implementation the heap size is rounded to the nearest number of physical pages.

To simplify the collector, the size of a protection unit must be evenly divisible by the size of a heap page. Bartlett has shown in [Bartlett 88] that configuring the collector to have smaller heap pages generally yields better performance. He chose 512-byte heap pages for the mostly-copying collector and its generational version, for both Scheme and C++. The incremental collector presented here also uses 512-byte heap pages, and the heap page and physical page boundaries are aligned such that there are eight (4096/512=8) heap pages on each physical page.

3.2.1.3 Allocation

During each garbage collection pause the collector scans a certain amount of memory and returns control to the application. The less there is to scan at each GC pause, the better the real-time performance is. This is because when there is no object crossing page boundaries at both ends of a physical page, then that physical page can be unprotected and scanned as a unit. But when there is an object crossing either boundary, then instead of unprotecting just one physical page, at least two physical pages will need to be unprotected, and subsequently scanned.

It is important to allocate objects in such a way that will minimize the amount of memory that needs to be scanned each time. The amount of memory to scan is dependent on how the objects are laid out on the physical pages. Scanning must be done in units of a physical page, since a physical page is a page protection unit. To illustrate this point with an extreme example, assume that current-space consists of a set of contiguous physical pages, and there are *objects lying across each physical page boundary*. Suppose that all the objects are promoted to forward-space at the beginning of collection, and all the physical pages they lie on are protected. When it is necessary to scan any one of these objects, it will be necessary to *scan all of the protected physical pages* as one unit. Thus, the collector cannot incrementally collect.

3.2.1.4 Miscellaneous Bookkeeping

The incremental collector often checks whether a physical page in the heap is protected. This information is kept in an array indexed by physical page. When a physical page is protected, its corresponding protect map entry is set; and when the page is unprotected, the same entry is cleared. The variable `protectedpages` holds the number of physical pages that are still protected, and is maintained to be consistent with the protect map.

In the non-incremental collector, `allocatedpages` holds the number of ‘‘live’’ heap pages, and `stablepages` holds the number of heap pages with even space numbers. For the incremental collector, this no longer works because as mentioned in section 3.2.1.1 we have added the notion of previous-space. Instead, the following variables are used to facilitate accounting:

- `currentpages` -- number of heap pages currently allocated in `curr_space`.

- `forwardedpages` -- total number of stable heap pages.
- `allocatedpages` -- total number of ‘‘live’’ heap pages, i.e. pages that are not free for allocation.

When the heap is first configured, all these variables are initialized to zero. Application allocation causes both `currentpages` and `allocatedpages` to increment. When collection is initiated, what used to be in current-space is ‘‘demoted’’ to previous-space, and `currentpages` is reset to zero. Promoting pages causes `forwardedpages` to increment. Subsequently application allocation increments `currentpages` and `allocatedpages`, just as before. When the collector allocates a page in forward-space, both `forwardedpages` and `allocatedpages` are incremented. When collection is finished, the following assignment happens:

```
allocatedpages=currentpages + forwardedpages;
```

which correctly exclude the reclaimed pages in the statistics.

3.2.2 Before GC

When the heap is first configured, all the heap pages are free and are given a space number equal to 1. The current-space marker `curr_space` is set to 3 initially; and the following condition is true

```
curr_space==forw_space==prev_space
```

whenever garbage collection is not taking place. Heap pages are allocated in current-space until one-third of the heap is exhausted. Unlike the non-incremental collector, the incremental collector does not wait until collection is completely done before allowing the application to resume. Therefore collection must start sooner to allow the intermixing of application allocation and collector allocation (for forwarded objects). If incremental collection starts when one-half of the heap is exhausted, and assuming the worst case in which all the previous-space objects have to be forwarded -- half of the heap will be occupied by previous-space objects and the other half by their forwarded copies -- then there will not be enough memory for the application to continue.

3.2.3 Start GC

The gist of incremental collection is to protect all the root objects at the start of collection, and delay the process of scanning them until it is convenient to do so. All the ambiguous root pointers in the processor stack, registers and the program’s static area are

identified. The heap pages which the ambiguous root pointers reference are ‘‘promoted’’ to forward-space, by setting the space numbers of those pages to be `forw_space`. For incremental collection to be correct, such that before garbage collection is over all the objects accessible from the roots will be scanned and properly forwarded, the physical pages containing the ambiguous root objects must be protected.

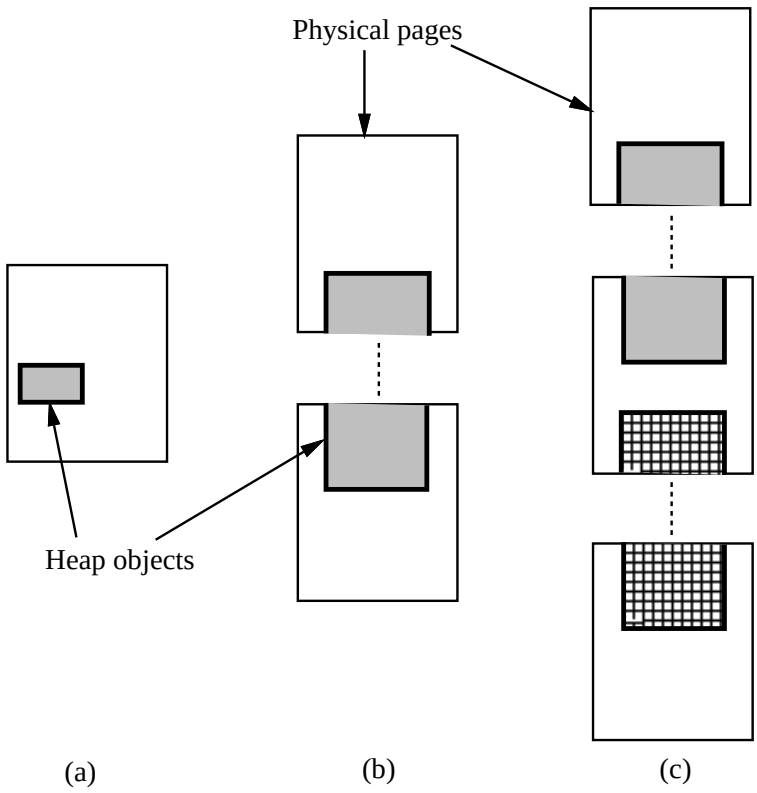
3.2.3.1 Protecting Objects

When there is an ambiguous root pointer in the stack or registers referencing an object in the heap, not only the physical page containing that part of the object being referenced needs to be protected. If the object is big and spans across several physical pages, the entire object has to be protected by protecting *all the physical pages* that it lies on. It would be incorrect to protect an object only partially, since the application would be able to access the unprotected part, and might therefore access a previous-space pointer.

The collector needs to protect the minimum number of physical pages containing the ambiguous root object such that no object is protected partially. This set of physical pages is called a *physical page cluster*, and is defined as the set of contiguous physical pages which have to be protected if any one object in any one of the pages is referenced by a root pointer.

Figure 3-2 illustrates three situations the collector has to consider when protecting root objects. Figure (a) shows the case in which an ambiguous root pointer is pointing at an object in a physical page which does not have any object crossing its page boundary; there is only one physical page in the page cluster. In figure (b) there is one object spanning across more than one physical pages, so the page cluster contains the set of physical pages from the one containing the beginning of the object to the one containing the end. In figure (c) there is more than one object crossing more than one physical page boundary, forming a ‘‘chain’’ of pages which must be protected.

The type identifier of the heap pages is used to determine the extent of the physical page cluster. There should be no object crossing the page boundaries (one at each end) of the physical page cluster. The first heap page of the cluster must be of type `OBJECT`, because a `CONTINUED` heap page would mean the head of the object is outside of the cluster. The first heap page just after the end of the cluster must be of type `OBJECT` also, because a `CONTINUED` page there would have meant the continuation of an object is outside



(a) Objects do not cross physical page boundary, i.e. page cluster has only one page.
(b) Page cluster has one object crossing physical page boundary.
(c) Page cluster has more than one object crossing physical page boundary.

Figure 3-2: Objects and Physical Page Clusters

of the cluster.[4] Figure 3-3 illustrates the idea.

3.2.3.2 Forward Region

One reason for using a copying garbage collector is to compact memory. When objects are copied into `forw_space`, it is desirable to have them close to each other. Because incremental collection distributes the task of scanning objects over several intervals

[4]Invalid heap pages outside of the heap are of type OBJECT, so this method of determining page clusters never mistakes an invalid page to be in the cluster.

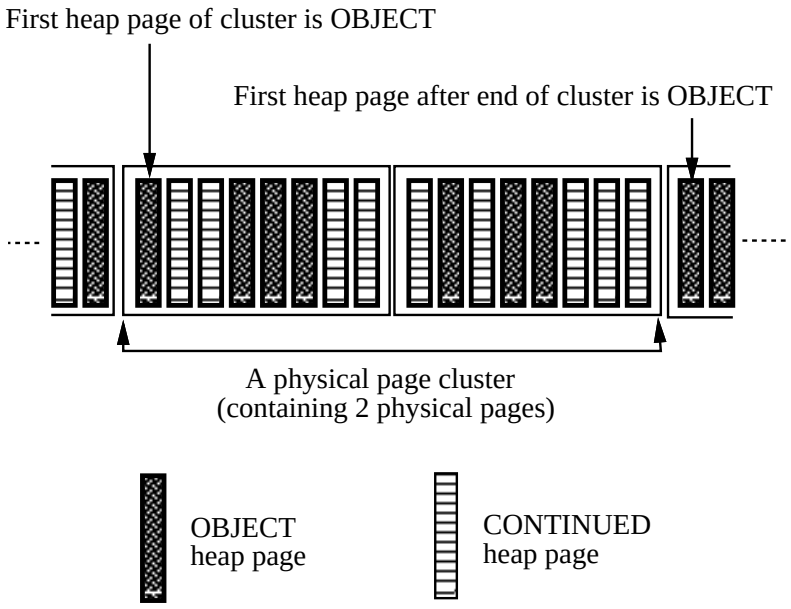


Figure 3-3: Determining physical page cluster

in time, forwarding objects in garbage collection mode occurs intermittently with allocating new objects in application mode. If special care is not taken to treat the two kinds of allocations differently, the resulting heap is likely to be very ‘‘fragmented’’, with a few heap pages of forwarded objects, followed by a few of newly-allocated objects, and so on. To deal with this problem, a ‘‘forward region’’ is reserved at the start of collection, by setting `forw_freepage` to be equal to `curr_freepage` and subsequently advancing `curr_freepage` to a number of free heap pages beyond the current value. The variable `curr_freepage` holds the page number of the first free page that will be allocated for an application heap page in current-space; likewise, `forw_freepage` is the first free page that will be allocated in forward-space. By spacing out `curr_freepage` and `forw_freepage`, a ‘‘forward region’’ is set up consisting of a group of heap pages where the forwarded objects are likely to be copied.

3.2.4 During GC

3.2.4.1 Page Fault Trap

When the application accesses an object on a protected physical page, the virtual memory system generates a page fault signal. This signal interrupts the application process and causes program control to be transferred to a fault handler. The fault handler finds out the faulting address of the protected page, and figures out the physical page cluster that needs to be unprotected and scanned (see section 3.2.3.1). The handler unprotects the physical page(s) in the page cluster and then calls the garbage collector's procedure to scan the page cluster. All the physical page(s) in the page cluster are unprotected at the beginning of the scanning, and *stay unprotected* throughout.

3.2.4.2 Scanning Objects

The basic scanning algorithm for the incremental collector is like that for the non-incremental collector described in section 2.2.3. The page cluster represents a unit of scanning work and is therefore also called the *scan region*. The scanning procedure looks at all the promoted and stable heap pages in the scan region, and scans the objects by proceeding in a breadth-first manner. However, there are additional concerns in the incremental collector which do not arise in the non-incremental one.

The concerns are primarily due to efficiency requirements imposed by the real-time aspect of the incremental collector. To achieve satisfactory real-time performance, the collector must spend as little time as necessary in scanning the scan region, and return control to the application as quickly as possible. Therefore, the collector should scan all the necessary objects, but not any more than necessary. This criterion directly affects the allocation of forward-space heap pages for the forwarded objects. Because the forwarded objects must be scanned eventually, they will have to be protected. If the forwarded objects were allowed to be allocated in the scan region, then the scanning routine will have to finish scanning these newly-forwarded objects before it can revert control to the application. Additionally, scanning the forwarded objects might in turn cause more to be forwarded into the scan region.

To avoid scanning more objects than absolutely necessary, a special forward-space allocation discipline is enforced where *the forwarded objects are always allocated outside of the scan region*. Figure 3-4 shows the same data structure as in figure 2-2, but with object A being on a different physical page from the rest of the objects. The bold outline of the

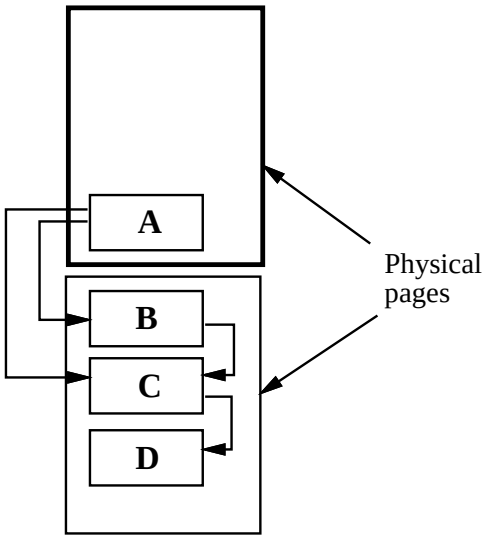


Figure 3-4: Scanning object A

rectangle enclosing A indicates that the physical page containing A is protected and currently causing a virtual memory page fault.[5] Therefore the page cluster has to be unprotected and scanned. Assume again that objects B, C, and D are less than one heap page in size, so that they will have to be copied. Figure 3-5 shows the result of scanning A: the objects B and C that A directly references are forwarded to another physical page (outside the scan region), and their copies B' and C' are subsequently protected. The forward pointers of B and C can be found in B's and C's header words. A forward pointer indicates the location of the copied object, so that other objects containing reference to the forwarded object can subsequently update the reference with the new location. In addition, A's internal pointers to B and C are updated to point to the forwarded copies. This completes scanning, and the physical page containing A can be accessible to the application now.

Scanning is triggered again when the physical page containing B' and C' causes a page fault. Scanning B' corrects its internal pointer (which still points to C when the fault is

[5]Note that although the page containing B, C and D is not protected, by the invariant of the incremental collector, the application program will never access these objects without going through the pointers in the forw-space (or stable) objects (see invariants listed in 1.3).

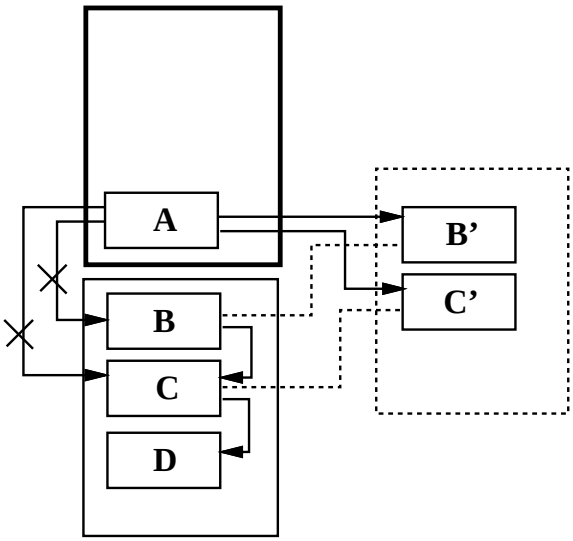


Figure 3-5: Scanning object A - II

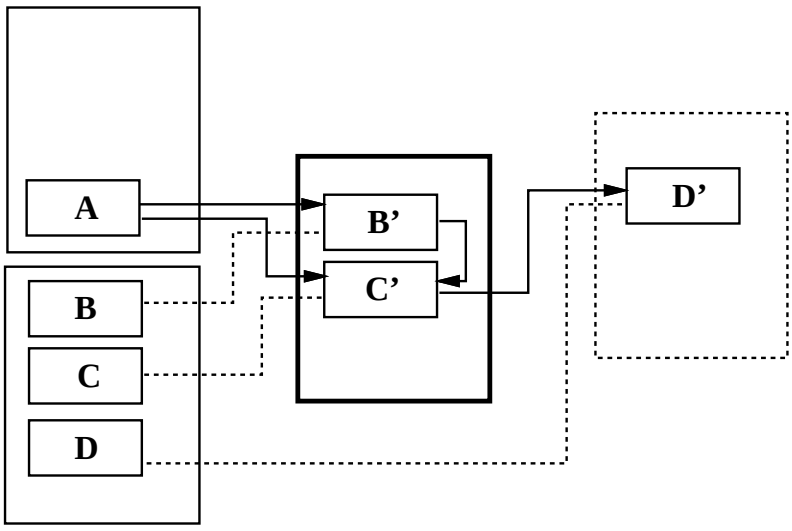


Figure 3-6: Scanning object A - III

generated) to point to C' instead, as illustrated in figure 3-6. C' is not copied (again)

because it is already forwarded and therefore stable. Stable objects are retained by the collector and their locations do not change. They are also guaranteed to be scanned before collection is over. Scanning C' is similar to that for scanning A: D is copied into D', on yet another physical page outside of the current scan region containing B' and C'. The forward pointer in D's header is updated to point to D', and the internal pointer in C' is updated to point to D' as well. D' will be protected after this level of scanning is complete. Since D' does not have any reference to other objects, scanning D' later in collection does cause any additional forwarding. (The above describes the basics of the scanning strategy, section 3.2.4.4 will discuss a memory fragmentation problem entailed by this scheme and present a revised design.)

In general, if a stable pointer is discovered when scanning an object, no further work needs to be done. For an internal pointer to an unstable object, there are a few possibilities, as outlined in figure 3-7. In this example, scanning C' discovers a pointer to D, which is a small object outside of the scan region, so D is forwarded. All (unstable) small objects, whether outside or inside the scan region, are forwarded in the hope of compacting memory.

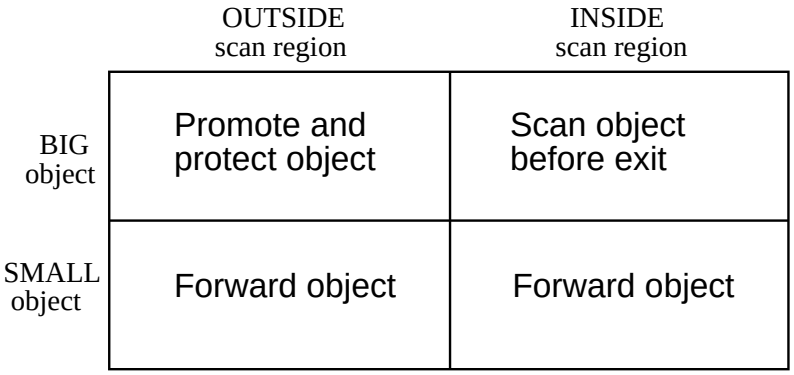


Figure 3-7: Scanning strategy for unstable objects

However, when a pointer to a large object is discovered, the location of the object is important. If the large object resides outside of the scan region, then its heap pages are promoted to forward-space, and the physical page(s) where it reside on is/are protected. It involves more work if the large object is inside the scan region, because no part of the scan region can be protected. Therefore the object must be promoted *and* scanned. Since the

scan region is always scanned in order -- the scan pointer makes one pass through the region and scans all objects which need to be scanned -- it can be determined with certainty whether an object in the scan region has had its chance to be scanned by comparing the address of that object with the scanning pointer. If the large object about to be promoted has not been swept by the scanning pointer, then it can be safely assumed that the scanning pointer will get to it later, realize that it has been made a stable object, and will therefore scan it. If on the other hand the object has already been swept by the scanning pointer, then the scanning procedure will have to ‘‘backtrack’’ to scan it properly. Correct operation requires that all the promoted objects in the scan region be scanned before the scanning procedure exits.

Depending upon the operating system the collector runs on, the cost of physical page protection facility may dominate the overhead of incremental collection. For this reason, it is desirable to not only minimize the amount of scanning each time the program pauses for garbage collection, but to minimize the number of memory protection calls. The naive way of implementing the forwarding strategy would be:

1. Unprotect the physical page containing the object to be forwarded, if it is protected.
2. Allocate a heap page in forward-space if necessary.
3. Unprotect the enclosing physical page if it is protected.
4. Copy the object onto the current actively forward heap page.
5. Protect the physical page containing this forward heap page.
6. Protect the physical page containing the forwarded object if it was unprotected in step 1.

Although this is a correct implementation, it is unnecessarily inefficient because more calls to protect/unprotect physical pages would have to be made. It is usually the case that a few forward-space heap pages are allocated on the same physical page, and a few forwarded objects are copied on the same forward-space heap page. Therefore it is wasteful to have to go through steps 3 and 5 on each copying operation. In the case where many forwarded objects tend to scatter on only a few physical pages, it would be costly to repeat steps 1 and 6 for every forwarded object. To minimize the number of memory protection calls, the scanning procedure registers those physical pages that are unprotected in steps 1 and 3, and remember to protect them before scanning exits. This strategy effectively unprotects all the necessary pages demanded by the scanning process and delays having to reprotect them until the very end.

3.2.4.3 Application Allocation

With only minor refinement, the incremental collector's allocation algorithm for newly-created current-space (application) objects follows closely from the one for the non-incremental collector described in section 2.2.2. Regardless of whether there is garbage collection ''going on,'' the collector finds a free heap page and allocates it in `curr_space`. If there is indeed unfinished garbage collection work, then a protected physical page is selected (randomly) to be scanned every time a current-space allocation is requested. This ensures that scanning will catch up with application allocation, so that it is less likely the heap will run out of space before collection is over.

Another refinement is that the incremental allocation algorithm has to be concerned with *page clusters*. Scanning occurs on the granularity of one page cluster at a time, so the length of a garbage collection pause is directly proportional to the size of the page cluster (the scan region), thus it is important that page clusters are not allowed to grow arbitrarily large, or the purpose of incremental collection will be defeated. Recall in figure 3-2 (c) that a big page cluster spanning several pages is formed when there are objects (larger than one heap page in size) crossing physical page boundaries on consecutive physical pages. To limit the size of page clusters, the current implementation forbids page clusters with more than one object crossing the physical page boundary. By checking the type identifiers of the heap pages at the neighboring physical page boundaries, the allocation procedure verifies that the heap page(s) about to be allocated do not contribute to the creation of a page cluster consisting of more than one object crossing page boundary. It is possible to adjust this restriction to tune performance on different hardware platforms. For instance, if scanning can be carried out faster on a high performance machine, then the restriction on page cluster size could allow more pages and still achieve satisfactory collection pauses.

3.2.4.4 Collector Allocation

Section 3.2.4.2 mentions that forwarded objects are always copied into areas outside of the scan region, so that scanning proceeds in a breath-first manner and pauses can be kept short. In the data structure in figure 2-2 on page 22, objects A, B, C and D are linked together in the same physical page cluster originally. But as the figure 3-6 on page 37 illustrates, by the time scanning is finished the whole object will be spread out over several physical pages, determined by the number of ''levels'' in the application's data structure. In the example, there are only three levels, but conceivably a very large and ''long'' linked data structure can be in the same page cluster, and scanning it will ''fragment'' the composite object onto many physical pages.

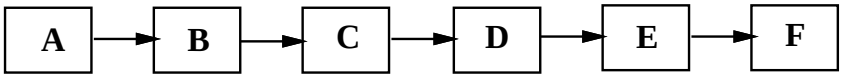
Using the allocation strategy discussed so far, A allocates a forward-space page to copy B and C, into B' and C', respectively[6], and when B' and C' are scanned, the trailing space on their `forw_space` heap page is discarded. For a program traversing down a linked list in order, this means that when the elements are forwarded one by one after successive page faults, each element will be made to ''occupy'' one whole heap page. If each element is much less than one heap page in size, the amount of memory fragmentation is significant.

To avoid fragmentation two `forw_freepage`'s are maintained. The idea is to flip-flop between the two ''forward regions'' referred to by the `forw_freepage`'s, so that instead of ''spreading out'' the composite object onto potentially many physical pages, it is separated into two regions. In addition, the original rule saying that ''the forwarded objects are always allocated outside of the scan region'' becomes *the forwarded objects are always allocated outside of the scan region, unless the current forward-space free heap page has enough space remaining for the next forward object to be copied there.* Perhaps it is most appropriate to illustrate with an example.

Figure 3-8 shows a linked list of six objects. Assume that each object occupies half a heap page, and that they are all residing on different physical pages. For simplicity assume also that all the physical pages are unoccupied except for the objects shown. The head of the list A is protected, so eventually the whole list structure will be forwarded into a more compact area. When a page fault ocurrs on the physical page containing A, A is scanned which causes B to be forwarded. B is subsequently protected, and the application is allowed to continue. Without the augmented rule, by the time B is scanned, the trailing space after B would have been wasted; but with the augmented rule, scanning B causes C to be copied onto the trailing space after B, and C is scanned immediately afterwards. Scanning C causes D to be forwarded, but this time onto a different physical page. Then D is protected, and the application continues. Eventually D is unprotected and scanned, so that E is copied to the trailing space after D. Like before, E is immediately scanned, which causes F to be forwarded. F is then protected and once again the application is allowed to proceed. Finally, when F is unprotected and scanned, there is no more objects to forward. The ultimate result of the scanning, as shown in the figure, is that the linked structure pointed to by A is compacted onto three heap pages on two physical pages. Because forward-space heap page allocation tries to flip-flop between two physical pages, if the linked structure

[6]Assumes that B' and C' are small enough to be on the same heap page.

Before scanning, all objects are on separate physical pages...



After scanning, objects B, ..., F are compacted onto two physical pages.

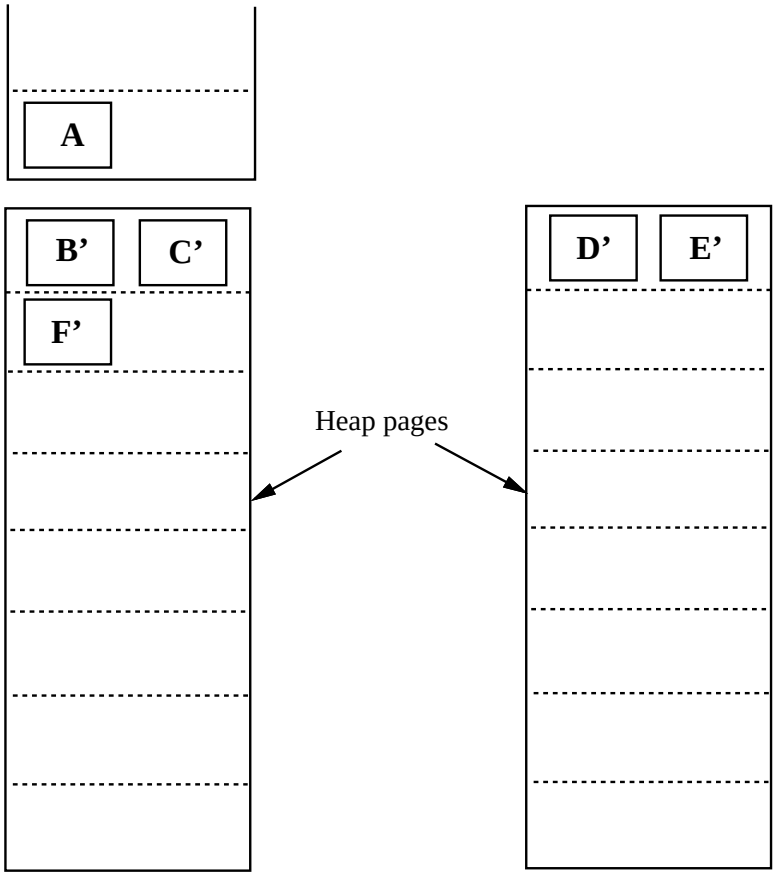


Figure 3-8: Scanning a linked list

were longer, then the remaining elements would also be compacted on these two physical pages.

3.2.5 End GC

Garbage collection is finished when all the protected physical pages have been unprotected and scanned. Each time a current-space heap page allocation is requested, the collector checks whether `protectedpages==0`, and if so, garbage collection is terminated. The trailing words left in `forw_freepage` are discarded; since there will not be any more forward objects to be copied. To mark that collection is over, `forw_space` is set to be equal to `curr_space`. To reclaim storage, `prev_space` is set to `curr_space`, so that all the heap pages which used to be in previous-space are now free for allocation -- a heap page is allocatable if and only if its space identifier is not stable and is not equal to `prev_space` or `curr_space`. The bookkeeping variable `allocatedpages` is corrected, and depending on how much of the heap is occupied (by examining the fraction `allocatedpages/heappages`), a total collection may be started. In that situation, all the stable heap pages are made unstable by assigning them the `curr_space` space identifier, and a new garbage collection is initiated.

3.2.6 Heap Page State Transitions

This section recapitulates the process of incremental, generational mostly-copying collection using state transitions. Figure 3-9 shows a map of all the different possible states for a heap page. The set of possible states is constructed from the cross product of {space identifier: *free*, *current-space*, *previous-space*, *forward-space*, or *stable*}[7], {page protection: protected, or unprotected}, and {collection status: scanned, or not scanned}. As the figure indicates the cross product generates twenty different combinations, but a number of them do not exist. For instance, only forward-space and stable heap pages are ever scanned, so by definition, there is not any scanned heap page (protected or otherwise) in either current, previous, or free space. The collector never allows the application to access unscanned stable and forward-space heap pages, so unprotected and unscanned forward-space and stable heap pages are not possible. Then there are some states that heap pages arrive in only

[7]The difference between forward-space and stable heap pages is that the former has its space identifier set to `forw_space`, while the latter has an even space identifier which is not equal to `forw_space`. In this section, stable heap pages and forward-space heap pages are distinct from each other: the entity ‘‘stable heap page’’ does not include any ‘‘forward-space heap page’’ and vice versa.

by coincidence. Since there is more than one heap page in a physical page, heap pages in current, previous, or free spaces can be found on a *protected* physical page if there happens to be unscanned forward-space or stable heap page on the same physical page. When the protected page is unprotected, the collector scans only the forward-space and stable heap pages. The collector does not operate on the other heap pages. Heap pages arriving in a state by coincidence leaves the state eventually, without active work on the part of the collector.

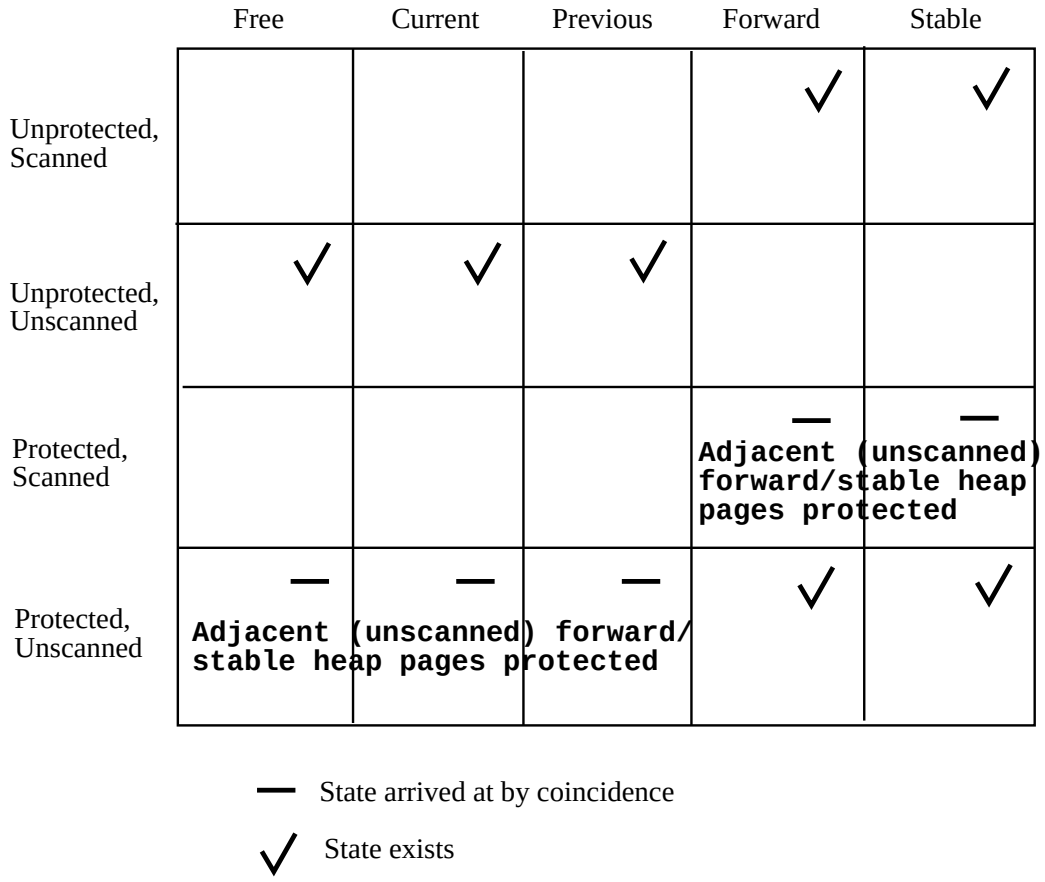


Figure 3-9: Heap page state table

From figure 3-9 one can see that out of twenty states only seven are reachable. Figure 3-10 describes the relationships among these seven states by inserting directed arcs between

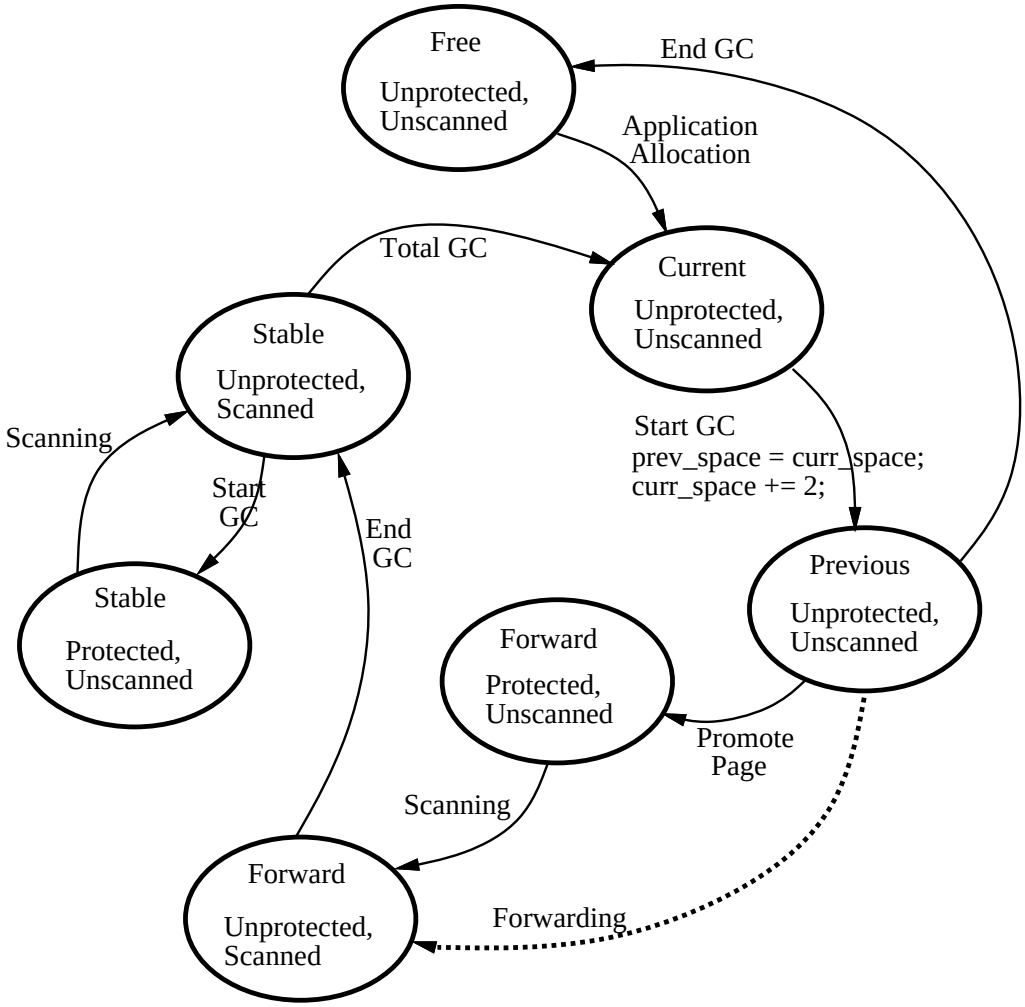


Figure 3-10: Heap page state transitions diagram

states and annotating how one state transitions to the next. Starting at the top of the figure, a free heap page is first allocated by the application program in current-space. When garbage collection is initiated, two possibilities can occur: either the heap page contains an ambiguous root object and is therefore promoted into forward-space, or the heap page does not contain any root object and is therefore demoted into previous-space. If the page is promoted it will have to be protected, but if it is demoted then no protection is necessary. Sometime during the collection some objects in a demoted page maybe copied into another forward-space heap page, if they are found to be accessible from the root objects. This is shown with a dotted arc because the previous-space heap page does not actually change state, only some objects inside it are forwarded. Another possibility is that if the collector discovers an object it needs to forward is larger than or equal to one heap page in size, it will just promote and protect the demoted page containing the big object into forward-space instead. At the end of collection, the previous-space page is reclaimed when it returns to free-space. A promoted page in forward-space may take a longer time to return to free-space. The protected forward-space page is eventually unprotected and scanned, and at the end of the collection it becomes a stable heap page. The stable heap page is retained, and is protected and scanned at each subsequent collection. This goes on until too many heap pages are retained in the heap, at which point the collector decides to do a total collection, and the stable heap page are sent into current-space and collected.

Chapter 4

Experimental Results

An incremental, generational mostly-copying collector has been built which accomplishes the following objectives:

1. *Hardware and compiler independence* -- The mostly-copying collection algorithm does not require any specialized hardware or compiler support to identify root objects. The result is a highly portable garbage collector that can be adapted to uncooperative environments.
2. *Real-time performance* -- The incremental collection strategy distributes the task of scanning objects over time and reduces the length of GC pauses typically suffered by non-incremental collectors drastically. On the DECStation 3100, the maximum GC pause of the incremental collector is well within the 100-millisecond limit imposed by most real-time applications. The average GC pause is merely 4 milliseconds long.
3. *Satisfactory total execution time* -- With generational collection, total execution time is reduced because repeated copying of long-living stable objects can be avoided. Generational collection is important for the incremental collector because incremental collection inherently requires more heap space, while generational collection lessens the contention for heap space.

Elaboration of these results can be found in the following sections. Section 4.1 states the success and limitation of the first objective. Section 4.2 describes the benchmark programs used and presents the benchmark measurements. Sections 4.3 discusses the accomplishment of the second objective, and section 4.4, the third objective.

4.1 Hardware and Compiler Independence

The incremental collector uses standard virtual memory support for page protection. It does not require any specialized hardware support. Because of this, the collector is very portable across different hardware platforms. Any system with support for user controlled page protection is able to use the collector. The collector currently runs under Digital Equipment Corporation's ULTRIX, and is using the operating system's facility, `mprotect`, to control page protection of the heap. The function `mprotect` is also found in SunOS; and many other versions of UNIX provide similar utilities to perform page protection. The

collector was developed on the DECStation 3100 platform (MIPS), and has been ported to the VAX architecture.

The collector also has the advantage of being able to be used with different C++ compilers. The implementation of the incremental collector does not involve any modification to the compiler. This way, the programmer is not tied to a particular compiler because of its garbage collection feature; rather, the programmer is free to choose a compiler that is most suitable to the application. The current implementation (listed in appendix B) is compatible with AT&T C++ Language Releases 1.2 and 2.0. It runs with both the AT&T C++ Translator for version 1.2 and the Glockenspiel C++ compiler for version 2.0.

However, it should be noted that some optimizing compilers which do not necessarily maintain all accessible pointers (i.e. roots) in the program state may destroy the validity of the root-finding heuristic used by the mostly-copying collection algorithm. For example, instead of having two root pointers referencing two different heap objects, an optimizing compiler may find it more convenient to keep track of just one pointer to one of the objects, and an *integer* offset from that pointer to locate the other object. The mostly-copying collector would then fail to retain the second object because there does not exist a pointer to it in the registers and stack.

4.2 Benchmark Measurements

The two benchmark programs chosen to assess the performance of the incremental collector were written at Digital Equipment Corporation's Western Research Laboratory, and used Bartlett's generational, mostly-copying collector. Since the syntax, semantics and pointer-finding methods of the new collector remain the same as Bartlett's original implementation, it was easy to use the programs with the new collector.

The first program, `words`, courtesy of Joel Bartlett, reads in a text file and builds a binary tree of storage records with information about each word in the file. The records are arranged according to the lexicographical order of the words. (The program listing can be found in appendix A.) The program's behavior in constructing, traversing and mutating the data structure puts heavy demand on efficient allocation and effective memory recycling mechanisms, which represents the class of applications that usually finds garbage collection useful.

The second program, `bipsctrl`, courtesy of Jeremy Dion and Louis Monier, is a part of the CAD system created at WRL for the design of a bipolar integrated processor chip. Unlike `words`, which manipulates only one data type that constitutes the individual elements of the binary tree; `bipsctrl` instantiates part of the chip design by creating instances of many data types to describe the various logical and electrical components. The program then automatically lays out and wires these components.

The programs were executed on a DECStation 3100 to generate the benchmark measurements shown in table 4-I. The left half of the table shows the measurements obtained for the `words` program, while the right half is for `bipsctrl`. For each program, four versions of the collector are used: (from left to right) they are non-incremental generational, incremental generational, non-incremental non-generational, and incremental non-generational. For each version of the collector, each program was run five times consecutively, and the measurements for the best run (chosen to be the one with the shortest total execution time) were used. The time measurements were obtained by adding `getrusage` calls to the collector. It was verified that `getrusage` did not significantly change the running times of the programs.

Reading the table from top to bottom, `Total time` measures the total execution time of the program in seconds. `Initial Heap` is the size of the heap (in Megabytes) when it is first configured, and `Final Heap` is the size of the heap at the end of the program's execution. `Time in GC` measures the cumulative time (measured in seconds) the program spends inside the collector module. `Time in AP` is the cumulative time in the application program itself, and is obtained by subtracting `Time in GC` from `Total time`. The reading of `%time in GC` is the percentage of the total time the program spends in GC mode, and `%time GC overlaps AP` is the percentage of the total time span when GC is technically ''going on'' (i.e. when `curr_space` != `forw_space`); while `%AP during GC` is the percentage of the GC time span (i.e. the total time GC overlaps AP) during which application is executing. The number of times garbage collection is initiated is listed at `#collections`; and the percentage of all the allocated heap pages that are reclaimed is shown in `%collected`. Following `%collected`, `Max Pause` indicates the maximum GC pause recorded during the run. The next three entries show the median, mean and standard deviation of all the non-zero GC pauses.[8] The last

[8]Because of the resolution of `getrusage`, a large number of the pauses are recorded as having zero millisecond duration.

Table 4-I: Comparative Measurements of Benchmark Programs

	words<mantext				bipsctrl			
	GENERATIONAL		NO GENERATION		GENERATIONAL		NO GENERATION	
	NOINC	INC	NOINC	INC	NOINC	INC	NOINC	INC
Total time (sec)	14.65	18.18	15.41	23.86	4.29	4.61	4.14	4.78
Initial Heap (MB)	1	1	1	1	1	1	1	1
Final Heap (MB)	1	2	1	1	2	2	2	2
Time in GC (sec)	1.26	3.58	2.04	9.02	1.02	0.86	0.85	1.01
Time in AP (sec)	13.39	14.6	13.37	14.84	3.27	3.75	3.29	3.77
% time in GC	8.64	19.68	13.26	37.83	23.74	18.57	20.59	21.08
% time GC overlaps AP	-	29.21	-	44.99	-	25.60	-	28.77
% AP during GC	-	32.84	-	16.10	-	28.47	-	27.91
# collections	18	27	17	36	3	3	3	3
% collected	96.14	96.04	57.25	7.01	36.63	79.4	61.56	58.14
Max Pause (msec)	86	59	176	20	605	86	352	43
Median Pause (msec) (of non-zero pauses)	70	4	121	4	207	4	262	4
Mean Pause (msec) (of non-zero pauses)	66.58	4.50	113.5	4.85	254.75	6.03	213.25	6.59
Std. Dev. of Pause (of non-zero pauses)	18.80	2.85	41.38	1.87	250.96	8.13	142.16	5.76
Mean Pause (msec) .(of ALL pauses)	66.58	1.58	113.5	2.74	254.75	3.29	213.25	4.22

entry shows the average GC pause of all the pauses, accounting for both non-zero and zero pauses.

The next two sections describe the real-time and generational aspects of the collector, respectively, and interpret some of the results in table 4-I.

4.3 Real-Time Performance

The major benefit of incremental collection is real-time performance. Real-time applications cannot tolerate the long and potentially unbounded GC pauses imposed by most non-incremental collectors. Figure 4-1 shows the time profile of the `bipsctrl` program running with the non-incremental, generational mostly-copying collector. The dark areas indicate that program control is inside the application itself, and the lighter areas indicate that control is inside the garbage collector. The duration of each of the lighter areas is on the order of hundreds of milliseconds. This is unacceptable for real-time performance. Appel et al. indicate that for a garbage collector to be real-time, the GC pauses must be less than a very small constant time. And for interactive applications, this maximum pause should be less 100 milliseconds [Appel et al. 88].

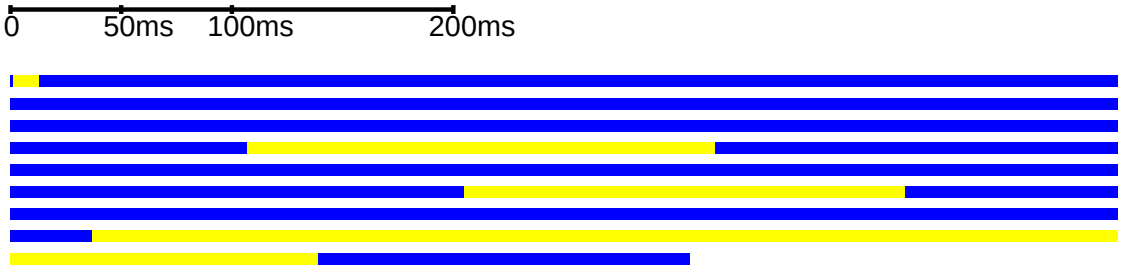


Figure 4-1: Time profile of `bipsctrl` running with the non-incremental collector

With the incremental collector, GC is divided into smaller chunks of work, as illustrated in figure 4-2. Instead of having pauses of hundreds of milliseconds each time the program enters garbage collection mode, GC now pauses much more frequently, but the length of each pause is significantly reduced. Most are less than 10 milliseconds, and the

maximum pause is still well within 100 milliseconds. Comparing figures 4-1 and 4-2, one can also notice that the total execution time for the incremental collector is longer. This is due to the overhead for having real-time collection.

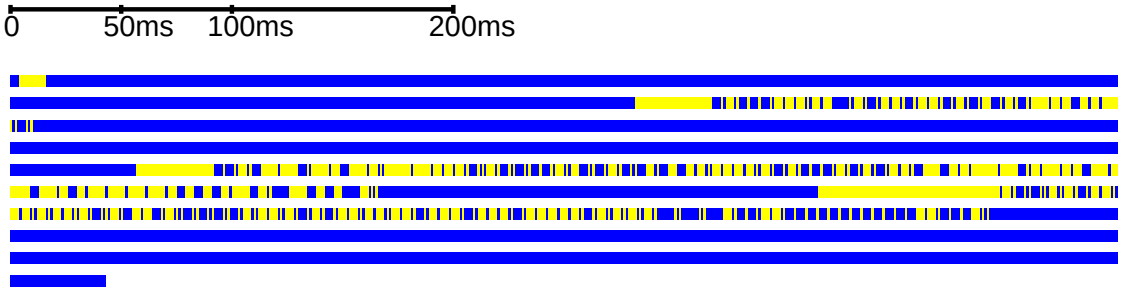


Figure 4-2: Time profile of `bipsctrl` running with the incremental collector

Looking across the `Max Pause` and `Median Pause` rows in table 4-I on page 50, one can see that the incremental collector has much smaller maximum pause and median pause than the non-incremental one for both benchmark programs. However, this is not achieved without any cost. For instance, the total time for `words` running with the incremental generational collector is 24% longer than that running with the non-incremental generational collector ((18.18-14.65)/14.65 = 24%). This is due to the fact that a larger number of garbage collection sessions are initiated in the incremental case than in the non-incremental case (27 times vs. 18 times). The incremental collector inherently requires more memory in order to run as efficiently as the non-incremental collector, because the previous-space heap pages are not reclaimed until the end of the collection, so comparatively less space is available. Therefore if the heap is not large enough, then collection is called frequently, resulting in much wasted scanning.

Along the same line of thinking, the reason why the incremental generational version of `bipsctrl` is only 7% slower than the non-incremental generational version (4.61 sec vs. 4.29 sec) is likely because the behavior of the program is such that equal number of garbage collection sessions (3) are initiated in both cases, and the heap is expanded once equally. It is likely that the amount of scanning is comparable in both cases, and therefore the comparable total execution times.

The costs of incremental collection come not only from scanning and forwarding, but from the overhead of `mprotect` and page fault trap also. Figure 4-II itemizes these costs. On the DECStation 3100, the cost of `mprotect` is 45μsec per call. A page trap takes 200μsec, which includes the time it takes to interrupt the program, enter and then exit the trap handler to return to the main program control. The matrix shows the overhead (in milliseconds) of the number of pointers to scan versus the size of each forwarded object (in number of words). For example, to scan a physical page of 1024 pointers and forward the 1024 64-word long objects that these pointers reference takes 16 milliseconds. The scanning and forwarding operation is essentially free unless the collector has to forward a large number of relatively large objects, e.g. the collector scans an object with 4096 pointers, which occupies 4 physical pages, such that each pointer points at a 100-words long objects to be forwarded. Such an object is somewhat unlikely, but the collector can still handle it in a reasonable amount of time (94 milliseconds). In addition, since the collector does not copy large objects exceeding one heap page (512 bytes = 128 words) in size, the overhead for forwarding these objects -- by simply changing the space identifier(s) -- is very low. This can be observed by the sharp drop in time overhead right past the 128-word mark: forwarding 4096 100-word long objects takes 94 milliseconds, but forwarding the same number of 128-word long objects takes only 20 milliseconds.

Figure 4-II indicates that the costs of each overhead is not overwhelmingly large. But when the cost is incurred over and again a large number of times, the cumulative overhead becomes significant. In the original design of the collector when no attempt was made to minimize the number calls to protect memory, `mprotect` was the dominant overhead. But with the optimization described in section 3.2.4.2, `mprotect` ceases to be the major cost of incremental collection. In fact, the combination of `mprotect` and page fault overhead in the benchmark programs accounts for only 3% of the total execution time. It is, on the other hand, when GC is initiated too often, then the cost of scanning/forwarding in the incremental collector becomes the dominating overhead. The data in table 4-I supports the claim that total execution time is proportional to the number of collections initiated. When garbage collection is called for frequently, it is probably because the heap is not big enough for the application's allocation need. Instead of spending most effort on reclaiming stale objects, the collector is likely to expend a disproportionate amount of time scanning objects that have just been scanned recently.

It is therefore important to tune performance of an application running with the incremental collector to ensure that the heap is large enough, so that excessive scanning and

Table 4-II: Overhead of page fault trap, `mprotect` and scanning

Hardware platform: DECStation 3100

Trap overhead = 200μsec/trap Mprotect overhead = 45μsec/call

Scanning and forwarding overhead:

	# WORDS/OBJECT											
	4	8	16	32	64	100	128	256	512	1024	2048	4096
# 2	-	-	-	-	-	-	-	-	-	4	-	-
P 4	-	-	4	-	-	-	-	-	-	-	-	-
T 8	-	-	-	-	-	-	-	-	-	-	-	-
R 16	-	-	-	-	-	-	-	-	-	-	-	-
S 32	-	-	-	-	-	-	-	-	-	-	-	-
64	-	-	-	-	-	-	-	-	-	-	-	-
128	-	-	-	4	4	4	-	-	-	-	-	-
256	4	-	4	-	4	4	-	-	-	-	-	4
512	4	3	4	12	8	11	3	4	4	4	4	4
1024	4	8	8	12	16	24	4	3	4	4	4	4
2048	19	16	19	19	35	43	12	12	11	11	11	11
4096	24	27	35	43	66	94	20	20	16	20	24	24

(Measurements inside matrix are in milliseconds.)

forwarding as a result of too many GC initiations is prevented. But under most circumstances, figure 4-II suggests that the real-time performance of the incremental collector will still be satisfactory for even very memory intensive codes. The sum of the trap and `mprotect` overheads, and the time for scanning and forwarding objects are likely to be within the 100 millisecond limit.

4.4 Soundness of Generational Collection

Generational collection improves performance whenever the extra cost of bookkeeping is less than the cost of the conducting the additional GC work. The strategy presented in this thesis treats the entire stable set as the remembered set. Since the remembered set is scanned at each collection, the desirability of generational collection decidedly depends upon whether retention of the so-called stable objects is a wise choice. It is beneficial if the stable objects remain alive for a relatively long period of time. Otherwise, if most of the stable objects ‘‘die’’ right after they become stable, then the heap space is not efficiently utilized, and generational collection is not saving much GC work at all -- it would actually be increasing GC work.

From table 4-I it can be seen that with generational collection the benchmarks yield almost uniformly better total execution time. Except for `bipsctrl` running with the non-incremental collector, all the other programs achieve better total time, as they are spending less time in GC. The percentage of GC work saved ranges from about 15%, in the case of the incremental version of `bipsctrl` ((1.01-0.86)/1.01 = 15%), to over 60%, in the case of the incremental version of `words` ((9.02-3.58)/9.02 = 60%). It is, however, worth noticing that the heap space requirement for the incremental generational version of `words` is higher than its counterparts. This is likely due to the fact that incremental generational collection tends to retain more stable objects.

Another point worth noticing is that the incremental generational versions of both benchmarks incur higher maximum GC pauses than the non-incremental versions. Maximum GC pause for the incremental collector always occurs during the call to start GC. The more objects to protect, the longer GC initiation takes. With incremental generational collection, more objects have to be protected during GC initiation because in addition to protecting the ambiguous roots, the collector has to protect the retained objects in the stable set.

Perhaps the longer GC pause can be viewed as a drawback of generational collection for the incremental collector; but when considering other more desirable aspects, generational collection seems to be favorable when used in conjunction with incremental collection. Incremental collection spreads out application allocation and collector scanning/forwarding, so that at any time less memory is available for allocation. Generational collection can decrease the collector's contention on heap space, because the amount of forwarding is reduced as stable objects are never forwarded. Therefore two benefits are accomplished simultaneously: (1) less overhead for copying forwarded objects; and (2) more heap space is available for application allocation. The first benefit depends on whether the stable objects are in fact long-living. The second one helps reduce the number of garbage collections needed.

Chapter 5

Summary and Future Work

5.1 Summary

My thesis in this project is that incremental collection can be done feasibly and efficiently in an architecture and compiler independent manner. To support my thesis, an experiment in building such a collector was carried out, and the following results have been gathered:

- An incremental, generational mostly-copying collector for C++ has been built.
- The collector runs on commercially-available uniprocessors without any special hardware assist. Currently it runs on both the MIPS and the VAX architectures.
- The collector runs in the UNIX platform, using the operating system's support for user controlled page protection (`mprotect`) to synchronize between the mutator and the scanner.
- The implementation of the collector does not require any modification to the C++ compiler. It is compatible with both AT&T C++ Language Releases 1.2 and 2.0.
- On the DECStation 3100, the maximum GC pause of the collector is well within the 100-millisecond limit imposed by most real-time applications. The majority of the GC pauses (i.e. average pause) are only 4 milliseconds each.
- The total execution times of the application programs are not adversely affected. This is due in part to generational collection, which alleviates contention for heap space and helps reduce the number of garbage collections needed.

There are two important lessons obtained from conducting this investigation:

1. An incremental collector requires more heap memory in order to run as efficiently as its non-incremental version. Incremental collection inherently demands more memory because collection is spread out over time and storage is not reclaimed until the end of collection.
2. The costs of trapping, `mprotect` calls, and scanning objects are not overwhelmingly large when viewed individually as single units. But when one or more of these costs are incurred a large number of times, then the cumulative effect can become prohibitively expensive. To enable efficient incremental collection, invocations of all of these costs must be kept at a minimum.

The second lesson is related to the first one in that when the available heap memory is insufficient, garbage collection is initiated frequently, resulting in repeated incurrence of trapping, `mprotect` and scanning costs, which otherwise could have been avoided. It is observed that excessive number of garbage collections can degrade performance reflected in the total execution time significantly.

5.2 Future Work

This incremental, non-concurrent collector can be easily extended to become a concurrent collector. The collector read/write protects the physical pages that the application is not supposed to access. Because of this, in a multiprocessor environment, the collector can scan the protected pages in parallel with the application program's execution. The concurrent collectors described in [Appel et al. 88] and [Detlefs 90] use this strategy of virtual memory page protection to synchronize between the mutator and the scanner. With a concurrent collection scheme, the total execution time for an application can be reduced significantly. Both [Appel et al. 88] and [Detlefs 90] report reductions in GC overhead by about 50%.

Presently, generational collection is conducted by protecting all the stable objects at the start of each collection. When the application accesses a protected stable object, the program traps and the collector enters to unprotect and scan the physical page cluster containing the object. This can be potentially wasteful because stable objects would not need to be scanned if they have not been mutated since the last time they were scanned. Conceivably more experiments can be done with different generational collection strategies in the C++ environment. For instance, if stable objects can be grouped together on the same physical page, and the page is WRITE protected only (so it can be read from but not written to), then the application will be able to read into stable objects without causing page faults. Page faults occur only when application tries to overwrite some part of the physical page, at which point the garbage collection can register that the page has been mutated and must remember to scan it later. Depending on the behavior of the program, this scheme may save some amount of scanning and page trap overhead.

Another area to work on is the further incrementalization of the collector, specifically, the incrementalization the GC initiation process. Maximum GC pause always occurs during the call to start garbage collection. But since the program faults many times during each

garbage collection session, the average GC pause is the average time it takes to perform GC work at each page fault. For the benchmark programs used in section 4.2, the average GC pause is only 4 milliseconds. It usually takes on the order to 4 milliseconds to unprotect and scan a physical page cluster (refer to figure 4-II). Even for very memory intensive programs, the overhead for unprotecting and scanning seems to be fairly well controlled. However, for the GC initiation process, the overhead is directly proportional to the size of the root set to be protected, and can approach the vicinity of 100 millisecond limit for a very large root set. Therefore, in order to reduce maximum pause, it is necessary that the performance bottleneck found in the GC initiation process be alleviated. It would be desirable to ‘‘spread out’’ the task of initiating GC, much like the concept of incremental collection, where the task of GC itself is ‘‘spread out’’ over time.

Appendix A
Sample C++ program using garbage collection

The following is the listing of the benchmark program **words**.

```
/* Read in words on cin.  Build a binary tree with a word record at each
   node.  Keep a frequency count of a word by updating the 'count' field
   in its word record in the tree.  Then output words in alphabetical
   order on cout.

   Copyright (c) 1989, Digital Equipment Corp.  All rights reserved.
*/

#include <stream.h>
#include <string.h>
#include <ctype.h>

#include  "gcalloc.h"

/* The basic data structure is a binary tree made up of items of the following
   form.
*/

struct  word  {
        word* lesser;
        word* greater;
        int  count;
        char  symbol[ 1 ];
        word( char* chars );
        GCCLASS( word );
};

word::word( char* chars )
{
        GCALLOCV( word, sizeof( word )+strlen( chars )+(1-4) );
        lesser = NULL;
        greater = NULL;
        count = 1;
        strcpy( symbol, chars );
}

void  word::GCPointers( )  {
        gcpointer( lesser );
        gcpointer( greater );
}

/* A word is read from CIN and a word entry is made by the following
   function.  Words are considered to be a string of one more alphabetical
   characters, case is ignored.
*/

int  nextc = ' ';

word*  read_word()
```

```
{
        char  symbol[ 100 ];
        int  cnt;

        while  (isalpha( nextc ) == 0)  {
           if  (nextc == EOF)  return( NULL );
           nextc = getchar();
        }
        cnt = 0;
        while  (isalpha( nextc ))  {
           if  (isupper( nextc ))  nextc = nextc+' ';
           symbol[ cnt++ ] = nextc;
           nextc = getchar();
        }
        symbol[ cnt ] = 0;
        return new word( symbol );
}

/* The table is printed on COUT by walking the tree in alphabetical order */

void  output_table( word* tree )
{
        if  (tree != NULL)  {
           output_table( tree->lesser );
           cout << tree->count << "\t" << tree->symbol << "\n";
           output_table( tree->greater );
        }
}

/* A entry is inserted into the table, or the count is incremented for each
   word by the following function.
*/

word*  count_word( word* tree, word* newword )
{
        int  cmp;

        if  (tree != NULL)  {
           cmp = strcmp( &newword->symbol[0], &tree->symbol[0] );
           if  (cmp < 0)
              tree->lesser = count_word( tree->lesser, newword );
           else  if  (cmp > 0)
              tree->greater = count_word( tree->greater, newword );
           else
              tree->count = tree->count+1;
           return( tree );
        }
        return( newword );
}

main( int argc, char* argv[] )
{
        word  *wp, *tree = NULL;

        while  ((wp = read_word()) != NULL)  tree = count_word( tree, wp );
        output_table( tree );
}
```

Appendix B
Source code for the Incremental Garbage Collector

B.1 Header file for C++ version 1.2

The following is the listing of **gcalloc-1.2.h**, *the header file for C++ version 1.2.*

```
/* This module implements garbage collected storage for C++ (version 1.2)
   programs using an incremental  version of the generational
   mostly-copying garbage collection algorithm.

   Copyright (c) 1991, 1989, Digital Equipment Corp.  All rights reserved.
*/

#ifndef GCALLOCH
#define GCALLOCH 1

/* Defining garbage collected classes
   ----------------------------------
   Classes allocated in the garbage collected heap must have constructor method
   and a "pointer locator" method name GCPointer.  For example, a class that
   holds a variable length string, a reference count, and pointers to strings
   that are greater or lesser than it can be defined as follows:

        struct  word  {
                word* lesser;
                word* greater;
                int  count;
                char  symbol[ 4 ];
                word( char* chars );
                GCCLASS( word );
        };

        word::word( char* chars )
        {
                GCALLOCV( word, sizeof( word )+strlen( chars )-3 );
                lesser = NULL;
                greater = NULL;
                count = 1;
                strcpy( symbol, chars );
        }

        void  word::GCPointers( )  {
                gcpointer( lesser );
                gcpointer( greater );
        }

   The declaration GCCLASS informs the garbage collector that the type word
   is garbage collected and has a callback method word::GCPointers.

   The constructor, word::word, allocates space in the heap and then
   initializes it.  Space allocation is done by the define GCALLOCV that takes
   the type and the size in bytes as its arguments.  When the size of the type
```

is known at compile time, storage can be allocated in the constructor method by using GCALLOC which takes the object type as it's argument. For example, for a type t, GCALLOCV(t, sizeof(t)) can be replaced by GCALLOC(t).

The "pointer locator" method, word::GCPointers, is used by the garbage collector to identify all pointers in the object. This is done by having the method call gcpointer with each pointer in the object that could point to a garbage collected object. In order to correctly invoke pointer location methods for superclasses or class objects contained in the class, the following rules must be followed:

1) for class C:P {}, C::GCPointers must contain P::GCPointers()
2) for class C { X x; }, C::GCPointers must contain x.GCPointers()
3) for class C { X* x; }, C::GCPointers must contain gcpointer(x)

Sometimes a class will be a subclass of a garbage collected object, yet add no pointers to the class. In this case, it need not specify GCCLASS in the class declaration, but it must include a constructor method that allocates garbage collected storage. Often this method can be expressed as:

```
subclass::subclass {GCALLOC(subclass)}
```

If the subclass does not provide this constructor, then instances of the subclass will be allocated from the non-garbage-collected heap. The base class may assure that this never happens by including the following test in its constructor method:

```
if  (gcobject( this ) == 0)  abort();
```

Once the object has been defined, storage is allocated using the normal C++ mechanism:

```
sp = new word( "dictionary" );
```

Definitions

pointer to an object - a pointer that points to the start of an object.
garbage collected object - an object whose storage is allocated by gcalloc.

Caveats

When the garbage collector is invoked, it searches the processor's registers, the stack, and the program's static area for "hints" as to what storage is still accessible. These hints are used to identify objects that are the "roots" and are to be left in place. Objects that the roots point to will be moved to compact the heap. Because of this:

Objects allocated in the garbage collected heap MAY MOVE.

Pointers to garbage collected objects MAY BE passed as arguments or stored in static storage.

Pointers to garbage collected objects MAY NOT be stored in dynamically allocated objects that are not garbage collected, UNLESS one has specified the GCHEAPROOTS flag in a gcheap declaration.

Pointers to garbage collected objects contained in garbage collected objects MUST always point outside the garbage collected heap or to a garbage collected object.

Sizing the heap

In order to make heap allocated storage as painless as possible, the user does not have to do anything to configure the heap. This default is an initial heap of 1 megabyte that is expanded in 1 megabyte increments whenever the heap is more than 25% full after a total garbage collection. Total garbage collections are done when the heap is more than 35% full.

However, if this is not the desired behavior, then it is possible to "tune" the collector by including one or more global gcheap declarations in the program. In order to understand the parameters supplied in a gcheap declaration, one needs an overview of the storage allocation and garbage collection algorithm.

Storage is allocated from the heap until 50% of the heap has been allocated. All accessible objects allocated since the last collection are retained and made a part of the stable set. If less than <collect all percent> of the heap is allocated, then the collection process is finished. Otherwise, the entire heap (including the stable set) is garbage collected. If the amount allocated following the total collection is greater than <increment heap percent>, then an attempt is made to expand the heap.

```
gcheap  <CC-identifier>( <initial heap size>,
                         <maximum heap size>,
                         <increment size>,
                         <collect all percent>,
                         <increment heap percent>,
                         <increment stable percent>,
                         <heap log> )
```

The arguments are defined as follows:

<CC-identifier>	a legal C++ identifier.
<initial heap size>	initial size of the heap in bytes. DEFAULT: 1048576.
<maximum heap size>	maximum heap size in bytes. DEFAULT: 2147483647.
<increment size>	# of bytes to add to each heap on each expansion. DEFAULT: 1048576.
<collect all percent>	number between 0 and 50 that is the percent allocated after a partial collection that will force a total collection. A value of 0 will disable generational collection. DEFAULT: 35.
<increment heap percent>	number between 0 and 50 that is the percent of newly allocated after a collection that will force heap expansion. DEFAULT: 25.
<increment stable percent>	number between 0 and 50 that is the percent stable after a collection is initiated that will force heap expansion. DEFAULT: 20.
<flags>	controls logging on stderr, error checking, and root finding: & GCSTATS = log collection statistics & GCMEM = log memory usage statistics & GCROOTLOG = log roots found in the stack, registers, and static area & GCHEAPROOTS = treat non-GC heap as roots & GCHEAPLOG = log possible roots in non-GC heap & GCTSTOBJ = perform object consistency tests

```
                                    & GCGUESSPTRS = guess number of pointers in
                                                    objects
                                    & GCZERO = zero free memory after GC
                                    & GCDEBUGLOG = log events internal to the
                                                   garbage collector
                                    & GCHEAPMAP = maintain memory allocation
                                                  map
                                  DEFAULT: 0.

   When multiple gcheap declarations occur, the one that specifies the largest
   <maximum heap size> value will control all factors except flags which is
   the inclusive-or of all <flags> values.

   Configured values may be overridden by values supplied from environment
   variables.  The user must set these variables in a consistent manner.  The
   variables and the values they set are:

        GCMINBYTES         <initial heap size>
        GCMAXBYTES         <maximum heap size>
        GCINCBYTES         <increment size>
        GCALLPERCENT       <collect all percent>
        GCINCPERCENT       <increment heap percent>
        GCSTABLEPERCENT    <increment stable percent>
        GCFLAGS            <flags>

   If any of these variables are supplied, then the actual values used to
   configure the garbage collector are logged on stderr.

   Limits
   ------
   No more than 2046 garbage collected classes.
   Individual objects no larger than 4,193,788 bytes.
*/

       /*****************************************************
        * C++ Garbage Collected Storage Interface Definitions *
        *****************************************************/

/* Declarations for objects not directly used by the user of the interface. */

typedef int  *GCP;              /* Pointer to a garbage collected object. */

extern  GCP gcmove( GCP ptr );  /* Objects are moved by this function. */

extern void  gccollect();       /* Invokes the collector. */

typedef void (*GCCALLBACKPROC)( GCP );
                                /* Callback procedure type */

extern int  gcregistercallback( GCCALLBACKPROC pointers, char* type );
                                /* Registers a callback method */

/* The following defines are used to compute the number of words needed for
   an object.  The count includes 1 word for the header.  These defines are
   used inside GCALLOC and GCALLOCV.
*/

#define GCBYTEStoWORDS( x ) (((x)+7)>>2)        /* Word align */

extern  GCP gcalloc( int words, int callback ); /* Actual space allocator */
```

```
/* User defined objects use the following definition to define the class
   as garbage collected.  It defines the pointer method, a static variable
   to hold the index into the garbage collector's table of methods, and a
   method to return the type name.
*/

#define GCCLASS( type )  void  GCPointers();                                  \
                         static int  __GCPointers;                            \
                         char*  __GCType(){ return "type"; }

/* Storage for fixed size objects is allocated by:  */

#define GCALLOC( type )                                                       \
   this = ((this==0) ?                                                        \
           (((__GCPointers==0) ?                                              \
             __GCPointers=                                                    \
                    gcregistercallback( (GCCALLBACKPROC)&type::GCPointers,    \
                                        type::__GCType() ) :                  \
             0),                                                              \
            (type*)gcalloc( GCBYTEStoWORDS( sizeof(type) ), __GCPointers )) : \
           this )

/* Storage for variable size objects is allocated by:  */

#define GCALLOCV( type, bytes )                                               \
   this = ((this==0) ?                                                        \
           (((__GCPointers==0) ?                                              \
             __GCPointers=                                                    \
                    gcregistercallback( (GCCALLBACKPROC)&type::GCPointers,    \
                                        type::__GCType() ) :                  \
             0),                                                              \
            (type*)gcalloc( GCBYTEStoWORDS( bytes ), __GCPointers )) :        \
           this )

/* The procedure gcpointer is called by the callback procedure with each
   pointer to a garbage collected object in the object.
*/

inline void  gcpointer( void*& ptr ) { ptr = (void*)gcmove( (GCP)ptr ); }

/* The following predicate returns 1 if the object is allocated where it will
   be scanned by the garbage collector, otherwise it returns 0.
*/

extern int  gcobject( void* ptr );

/* The class gcheap is used to configure the heap as earlier described.  */

class  gcheap {
   public:
        gcheap( int minheapbytes,
                int maxheapbytes,
                int incheapbytes,
                int allpercent,
                int incpercent,
                int stablepercent,
                int flags );
};

const   GCSTATS = 1,            /* Log garbage collector info */
```

```
        GCMEM = 2,              /* Log memory usage information */
        GCROOTLOG = 4,          /* Log roots found in registers, stack and
                                   static area */
        GCHEAPROOTS = 8,        /* Treat non-GC heap as roots */
        GCHEAPLOG = 16,         /* Log possible non-GC heap roots */
        GCTSTOBJ = 32,          /* Extensively test objects */
        GCGUESSPTRS = 64,       /* Guess pointers in objects */
        GCZERO = 128,           /* Zero free memory after GC */
        GCDEBUGLOG = 256,       /* Log events internal to collector */
        GCHEAPMAP = 512,        /* X-window display showing allocation */
        GCNOINC = 1024;         /* Force non-incremental collection */

#endif
```

B.2 Header file for C++ version 2.0

The following is the listing of **gcalloc-2.0.h**, *the header file for C++ version 2.0.*

```
/* This module implements garbage collected storage for C++ (version 2.0)
   programs using an incremental  version of the generational
   mostly-copying garbage collection algorithm.

   Copyright (c) 1991, 1989, Digital Equipment Corp.  All rights reserved.
*/

#ifndef GCALLOCH
#define GCALLOCH 1

/* Defining garbage collected classes
   ----------------------------------
   Classes allocated in the garbage collected heap are denoted by the GCCLASS
   statement in their declaration.  For example, a class that holds a fixed
   length string, a reference count, and pointers to strings that are greater
   or lesser than it can be defined as follows:

        struct  word  {
                word* lesser;
                word* greater;
                int  count;
                char  symbol[ 4 ];
                word( char* chars );
                GCCLASS( word );
        };

        word::word( char* chars )
        {
                lesser = NULL;
                greater = NULL;
                count = 1;
                strcpy( symbol, chars );
        }

        void  word::GCPointers( )  {
                gcpointer( lesser );
                gcpointer( greater );
```

```
    }
```

The declaration GCCLASS informs the garbage collector that the type word is garbage collected. Besides overloading new and delete for this type of object, it declares the user defined "pointer locator" method.

The "pointer locator" method, word::GCPointers, is used by the garbage collector to identify all pointers in the object. This is done by having the method call gcpointer with each pointer in the object that could point to a garbage collected object. In order to correctly invoke pointer location methods for superclasses or class objects contained in the class, the following rules must be followed:

1) for class C:P {}, C::GCPointers must contain P::GCPointers()
2) for class C { X x; }, C::GCPointers must contain x.GCPointers()
3) for class C { X* x; }, C::GCPointers must contain gcpointer(x)

Once the object has been defined, storage is allocated using the normal C++ mechanism:

```
    sp = new word( "dictionary" );
```

Caveats

When the garbage collector is invoked, it searches the processor's registers, the stack, and the program's static area for "hints" as to what storage is still accessible. These hints are used to identify objects that are the "roots" and are to be left in place. Objects that the roots point to will be moved to compact the heap. Because of this:

Objects allocated in the garbage collected heap MAY MOVE.

Pointers to garbage collected objects MAY BE passed as arguments or stored in static storage.

Pointers to garbage collected objects MAY NOT be stored in dynamically allocated objects that are not garbage collected, UNLESS one has specified the GCHEAPROOTS flag in a gcheap declaration.

Pointers to garbage collected objects contained in garbage collected objects MUST always point outside the garbage collected heap or to a garbage collected object.

Garbage collected arrays are not supported as arrays are always allocated by the global ::operator new() (section 5.3.3, AT&T C++ Language System Release 2.0).

Variable size objects

Garbage collected objects who's size is computed at runtime have their storage allocated by having the class' constructor method have a call to GCALLOV as its first statement. GCALLOV takes two arguments, the name of the class and the number of bytes needed. For example, a variable size word might use the constructor:

```
    word::word( char* chars )
    {
            GCALLOCV( word, <SOME FIXED SIZE>+strlen( chars ) );
            lesser = NULL;
```

```
            greater = NULL;
            count = 1;
            strcpy( symbol, chars );
    }
```

N.B. As GCALLOCV relies on the "assignment to this" anachronism, it is subject to change in future releases of the compiler.

Sizing the heap

In order to make heap allocated storage as painless as possible, the user does not have to do anything to configure the heap. This default is an initial heap of 1 megabyte that is expanded in 1 megabyte increments whenever the heap is more than 25% full after a total garbage collection. Total garbage collections are done when the heap is more than 35% full.

However, if this is not the desired behavior, then it is possible to "tune" the collector by including one or more global gcheap declarations in the program. In order to understand the parameters supplied in a gcheap declaration, one needs an overview of the storage allocation and garbage collection algorithm.

Storage is allocated from the heap until 50% of the heap has been allocated. All accessible objects allocated since the last collection are retained and made a part of the stable set. If less than <collect all percent> of the heap is allocated, then the collection process is finished. Otherwise, the entire heap (including the stable set) is garbage collected. If the amount allocated following the total collection is greater than <increment heap percent>, then an attempt is made to expand the heap.

```
gcheap  <CC-identifier>( <initial heap size>,
                         <maximum heap size>,
                         <increment size>,
                         <collect all percent>,
                         <increment heap percent>,
                         <increment stable percent>,
                         <heap log> )
```

The arguments are defined as follows:

<CC-identifier>	a legal C++ identifier.
<initial heap size>	initial size of the heap in bytes. DEFAULT: 1048576.
<maximum heap size>	maximum heap size in bytes. DEFAULT: 2147483647.
<increment size>	# of bytes to add to each heap on each expansion. DEFAULT: 1048576.
<collect all percent>	number between 0 and 50 that is the percent allocated after a partial collection that will force a total collection. A value of 0 will disable generational collection. DEFAULT: 35.
<increment heap percent>	number between 0 and 50 that is the percent of newly allocated after a collection that will force heap expansion. DEFAULT: 25.
<increment stable percent>	number between 0 and 50 that is the percent stable after a collection is initiated that will force heap expansion. DEFAULT: 20.
<flags>	controls logging on stderr, error checking, and root finding: & GCSTATS = log collection statistics

```
                                    & GCMEM =  log memory usage statistics
                                    & GCROOTLOG = log roots found in the stack,
                                                  registers, and static area
                                    & GCHEAPROOTS = treat non-GC heap as roots
                                    & GCHEAPLOG = log possible roots in non-GC
                                                  heap
                                    & GCTSTOBJ = perform object consistency
                                                 tests
                                    & GCGUESSPTRS = guess number of pointers in
                                                    objects
                                    & GCZERO = zero free memory after GC
                                    & GCDEBUGLOG = log events internal to the
                                                   garbage collector
                                    & GCHEAPMAP = maintain memory allocation
                                                  map
                                  DEFAULT: 0.

   When multiple gcheap declarations occur, the one that specifies the largest
   <maximum heap size> value will control all factors except flags which is
   the inclusive-or of all <flags> values.

   Configured values may be overridden by values supplied from environment
   variables.  The user must set these variables in a consistent manner.  The
   variables and the values they set are:

        GCMINBYTES         <initial heap size>
        GCMAXBYTES         <maximum heap size>
        GCINCBYTES         <increment size>
        GCALLPERCENT       <collect all percent>
        GCINCPERCENT       <increment heap percent>
        GCSTABLEPERCENT    <increment stable percent>
        GCFLAGS            <flags>

   If any of these variables are supplied, then the actual values used to
   configure the garbage collector are logged on stderr.

   Limits
   ------
   No more than 2046 user defined garbage collected classes.
   Individual objects no larger than 4,193,788 bytes.
*/

       /*****************************************************
        * C++ Garbage Collected Storage Interface Definitions *
        *****************************************************/

/* Declarations for objects not directly used by the user of the interface. */

typedef int  *GCP;              /* Pointer to a garbage collected object. */

extern  GCP gcmove( GCP ptr );  /* Objects are moved by this function. */

extern void  gccollect();       /* Invokes the collector. */

typedef void (*GCCALLBACKPROC)( GCP );
                                /* Callback procedure type */

extern int  gcregistercallback( GCCALLBACKPROC pointers, char* type );
                                /* Registers a callback method */
```

```
/* The following define is used to compute the number of words needed for
   an object.  The count includes 1 word for the header.  The defines are
   used inside GCCLASS and GCALLOCV.
*/

#define GCBYTEStoWORDS( x ) (((x)+7)>>2)        /* Word align */

extern  GCP gcalloc( int words, int callback ); /* Actual space allocator */

/* User defined objects use the following definition to define the class
   as garbage collected.  It defines the pointer method, a static variable
   to hold the index into the garbage collector's table of methods, and storage
   allocation methods.
*/

#define GCCLASS( type )                                                    \
        void* operator new( unsigned int bytes )  {                        \
           if  (__GCPointers == 0)                                         \
              __GCPointers =                                               \
                    gcregistercallback( (GCCALLBACKPROC)&type::GCPointers,  \
                                        #type );                           \
           return (void*)gcalloc( GCBYTEStoWORDS( bytes ), __GCPointers );  \
        }                                                                  \
        void operator delete( void* ) {}                                   \
        void  GCPointers();                                                \
        static int  __GCPointers

/* GCPointer methods move pointers by calling the procedure gcpointer.  It is
   actually a define enclosing an inline procedure as C++ 2.0 does not
   correctly compile void*& arguments.
*/

#define  gcpointer( x ) _gcpointer( &x )

inline void  _gcpointer( void* ptr ){ *((GCP*)ptr) = gcmove( *((GCP*)ptr) ); }

/* Storage for variable size objects is allocated by the following mechanism
   that depends upon ASSIGNMENT TO THIS, i.e. this feature might not work in
   the future.
*/

#define GCALLOCV( type, bytes )                                            \
   this = ((this==0) ?                                                     \
           (((__GCPointers==0) ?                                           \
             __GCPointers=                                                 \
                    gcregistercallback( (GCCALLBACKPROC)&type::GCPointers,  \
                                        #type ) :                          \
             0),                                                           \
            (type*)gcalloc( GCBYTEStoWORDS( bytes ), __GCPointers )) :     \
           this )

/* The class gcheap is used to configure the heap as earlier described.  */

class  gcheap {
   public:
        gcheap( int minheapbytes,
                int maxheapbytes,
                int incheapbytes,
                int allpercent,
                int incpercent,
```

```
                int stablepercent,
                int flags );
};

const   GCSTATS = 1,            /* Log garbage collector info */
        GCMEM = 2,              /* Log memory usage information */
        GCROOTLOG = 4,          /* Log roots found in registers, stack and
                                   static area */
        GCHEAPROOTS = 8,        /* Treat non-GC heap as roots */
        GCHEAPLOG = 16,         /* Log possible non-GC heap roots */
        GCTSTOBJ = 32,          /* Extensively test objects */
        GCGUESSPTRS = 64,       /* Guess pointers in objects */
        GCZERO = 128,           /* Zero free memory after GC */
        GCDEBUGLOG = 256,       /* Log events internal to collector */
        GCHEAPMAP = 512,        /* X-window display showing allocation */
        GCNOINC = 1024;         /* Force non-incremental collection */
#endif
```

B.3 Program file for the incremental collector

The following is the listing of the incremental, generational mostly-copying garbage collector for C++. It is compatible with C++ versions 1.2 and 2.0.

```
/* This module implements garbage collected storage for C++ programs using
   an incremental version of the generational "mostly-copying" garbage
   collection algorithm.  The implementation is compatible with AT&T C++
   Language System Releases 1.2 and 2.0.

   Copyright (c) 1991, 1989, Digital Equipment Corp.  All rights reserved.

   For a discussion of the interface, see gcalloc-1.2.h (for C++ version
   1.2) or gcalloc-2.0.h file (for C++ version 2.0).

   For a discussion of the mostly-copying garbage collection
   algorithm, see

   Joel Bartlett,
   "Compacting Garbage Collection with Ambiguous Roots",
   WRL Research Report 88/2, February 1988.

   Joel Bartlett,
   "Mostly-Copying Garbage Collection Picks Up Generations and C++",
   WRL Technical Note TN-12, October 1989.

   For a discussion of the incremental, generational mostly-copying
   collection algorithm, see

   G. May Yip,
   "Incremental, Generational Copying Garbage Collection in
   Uncooperative Environments",
   MIT SM Thesis, June 1991.
*/

/* Default is C++ Version 1.2 */
```

```
#if (!COMPILER_VERSION_1_2 & !COMPILER_VERSION_2_0)
#define COMPILER_VERSION_1_2  1
#endif

/* External definitions */

#include <stdio.h>      /* Streams are not used as they might not be
                           initialized when needed. */
#include <sys/ioctl.h>
#include <sys/time.h>
#include <machine/param.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <signal.h>

#ifdef COMPILER_VERSION_2_0

#include <libc.h>
#include <osfcn.h>
#include <stdlib.h>

#endif

#ifdef COMPILER_VERSION_1_2
extern char*  getenv( char* name );

extern unsigned*  sbrk( int size );

extern void  bzero( char* string, int length );

extern  free( char* obj );

extern  pipe( int filedes[ 2 ] );

extern  fork();

extern  close( int d );

extern  dup( int d );

extern  select( int nfds, int *readfds, int *writefds, int *execptfds,
                struct timeval *timeout);

extern  execlp(char *file, char *arg0 ... );

extern int  getpagesize();

extern int  mprotect( void* addr, int numbytes, int protection);
#endif

#ifdef COMPILER_VERSION_2_0
extern "C"  void  bzero( char* string, int length );

extern "C"  select( int nfds, int *readfds, int *writefds, int *execptfds,
                    struct timeval *timeout);

extern "C"  getpagesize();

extern "C"  mprotect( void* addr, int numbytes, int protection);
#endif
```

```
       /**********************************
        * Processor Dependent Definitions *
        **********************************/

/* MIPS */

#ifdef mips

#include <machine/vmparam.h>

/* Assume stack alignment on 32-bit words. */

#define  STACKINC  4

/* Need to save and examine registers s0-s8. */

#define  REGISTER_COUNT 9
#ifdef COMPILER_VERSION_1_2
extern unsigned*  gcregisters( unsigned* registers );
#endif
#ifdef COMPILER_VERSION_2_0
extern "C" unsigned*  gcregisters( unsigned* registers );
#endif

/* Static area bounds */

extern unsigned  end;
#define STATIC_0 ((unsigned*)USRDATA)
#define STATIC_1 (&end)

/* Objects must be double aligned */

#ifndef MISALIGN
#define DOUBLE_ALIGN 1
#endif

/* Physical page size: phys_shift = log[base 2]( size of a physical page ) */

#define phys_shift 12

#endif  /* MIPS */


/* VAX */

#ifdef vax

#include <machine/vmparam.h>

/* Assume stack alignment on 32-bit words. */

#define  STACKINC  4

/* Need to save and examine registers 2-10. */

#define  REGISTER_COUNT 10
#ifdef COMPILER_VERSION_1_2
extern unsigned*  gcregisters( unsigned* registers );
#endif
#ifdef COMPILER_VERSION_2_0
```

```
extern "C" unsigned*  gcregisters( unsigned* registers );
#endif

/* Static area bounds */

extern unsigned  etext, end;
#define STATIC_0 ((unsigned*)(((((int)&etext)+NBPG-1)/NBPG)*NBPG))
#define STATIC_1 (&end)

/* Physical page size: phys_shift = log[base 2]( size of a physical page ) */

#define phys_shift 10

#endif  /* VAX */


/* Bottom of stack is computed by the constructor for the global variable
   sb.
*/

static struct stackbase {
        unsigned  address;
        stackbase( int i ) { address = (((unsigned)&i+NBPG-1)/NBPG)*NBPG; }
};

static  stackbase  sb( 0 );

#define  STACKBASE (sb.address)


       /************************************
        * Garbage Collected Heap Definitions *
        ************************************/

/* The heap consists of a discontiguous set of memory blocks, called
   heap pages, and each heap page is PAGEBYTES long.  There are two
   restriction on the size of a heap page: (i) PAGEBYTES must be
   *smaller* than the virtual memory page size, and (ii) the virtual
   memory page size (in bytes) must be a multiple of PAGEBYTES.

   Terminology: for the purpose of documentation, "page" and "heap page"
   both all refer to a PAGEBYTES-long memory in the heap, while
   "physical page" refers to a virtual memory page.  Furthermore, whenever
   the word "page" is encountered, it is assumed to mean a page in the heap,
   unless when modified explicitly, as in "a page in the heap must be smaller
   than a physical page."
*/

static  int firstheappage,     /* Page # of first heap page */
            lastheappage,      /* Page # of last heap page */
            heappages,         /* # of pages in the heap */
            heapspanpages,     /* # of pages that span the heap */
            physspanpages,     /* # of physical pages that span the heap */
            curr_freewords,    /* # words left on the current page */
            *curr_freep,       /* Ptr to the first free word on the current
                                  page */
            forw_freewords,    /* # words left on the forward page */
            *forw_freep,       /* Ptr to the first free word on the forward
                                  page */
            firstword_to_scan, /* Address of first word to be scanned */
```

```
            lastword_to_scan,  /* Address of last word to be scanned */
            *scanp,            /* Pointer to the object being scanned */
            allocatedpages,    /* # total number of pages allocated */
            currentpages,      /* # of pages allocated for current space */
            forwardedpages,    /* # of pages in the stable set (forwarded) */
            protectedpages,    /* # of physical pages being protected */
            curr_freepage,     /* First free page in current space */
            forw_freepage,     /* First free page in forward space */
            old_forw_freepage, /* Previous value of forw_freepage */
            *space,            /* Space number for each page */
            *plink,            /* Page link for each page */
            *type,             /* Type of object allocated on the page */
            *protect,          /* protect map for physical pages */
            *firstword,        /* Bitmap of 1st words of user objects */
            queue_head,        /* Head of list of stable set of pages */
            queue_tail,        /* Tail of list of stable set of pages */
            curr_space,        /* Current space number */
            prev_space,        /* Previous space number */
            forw_space;        /* Forward space number */


/* During incremental collection, heap pages can be in one of the
   following states, depending on their page generation (i.e. space)
   number:

   (*) NEWLY ALLOCATED - for recently allocated pages;
                         space[page] == curr_space.
   (*) PREVIOUSLY ALLOCATED - for pages that were allocated before
                              incremental collection was initiated;
                              space[ page ] == prev_space.
   (*) FORWARDED - for pages containing forwarded objects;
                   space[ page ] == forw_space.
   (*) FREE - free pages that could be allocated, either as newly allocated
              pages or forwarded pages;
              space[ page ] != curr_space && space[ page ] != prev_space &&
              space[ page ] != forw_space.
*/

/* Page types */

#define OBJECT 0
#define CONTINUED 1

/* PAGEBYTES controls the number of bytes/heap page */

#define PAGEBYTES 512
#define PAGEWORDS (PAGEBYTES/sizeof(int))
#define WORDBYTES (sizeof(int))
#define HEAPPERCENT( x ) (((x)*100)/heappages)

/* The following variable holds the capacity of a physical page,
   i.e. # of heap pages/physical page.
 */
static int  phys_cap;

/* PHYS_PAGEBYTESAES controls the nubmer of bytes/physical page */

#define PHYS_PAGEBYTES (phys_cap*PAGEBYTES)
#define PHYS_PAGEWORDS (PHYS_PAGEBYTES/sizeof(int))
```

```
/* Number of pages reserved for forwarded objects in the "forward region" */
#define RESERVEDPAGES  (heappages/5)

/* The physical page mask can be bitwise ANDed with an address to obtain
   the corresponding starting address of the physical page.
*/
static int  phys_pagemask;

/* Similarly, the heap page mask can be bitwise ANDed with an address to
   obtain the corresponding starting address of the heap page.
*/
static int  heap_pagemask;

/* Page number <--> pointer conversion is done by the following defines */

#define PAGE_to_GCP( p ) ((GCP)((p)*PAGEBYTES))
#define GCP_to_PAGE( p ) (((int)(p))/PAGEBYTES)

/* GC Pointer, ADDRess, PAGE number, and Physical Page ADDRess conversion */

#define GCP_to_PPADDR( gcp )     (int(gcp) & phys_pagemask)
#define ADDR_to_PAGE( addr )     (((addr) & heap_pagemask)/PAGEBYTES)
#define PAGE_to_ADDR( page )     ((page)*PAGEBYTES)
#define PAGE_to_PPADDR( page )   ((page)*PAGEBYTES & phys_pagemask)

/* Space values */

#define UNALLOCATEDPAGE -2
#define FREEPAGE 1;
#define STABLE( x )     ((~space[ (x) ]) & 1)
#define UNSTABLE( x )   (space[ (x) ] & 1)


/* Objects that are allocated in the heap have a one word header.  The form
   of the header is:

         31           21 20             1 0
        +---------------+----------------+-+
        | callback indx | # words in obj |1|
        +---------------+----------------+-+
        |          user data               |  <-- user data starts here.
                       .
                       .
                       .
        |                                  |
        +----------------------------------+

   The number of words in the object count INCLUDES one word for the header.

   When an object is forwarded, the header is replaced by the pointer to
   the new object that will have bit 0 equal to 0.
*/

#define MAKE_CALLBACK( index ) ((index)<<21 | 1)
#define MAKE_HEADER( words, callback ) ((words)<<1 | (callback))
#define FORWARDED( header ) (((header) & 1) == 0)
#define HEADER_CALLBACK( header ) ((header)>>21 & 0x7FF)
#define HEADER_WORDS( header ) ((header)>>1 &0xFFFFF )
#define HEADER_BYTES( header ) (((header)>>1 & 0xFFFFF)*WORDBYTES)
#ifdef DOUBLE_ALIGN
```

```
#define ONEPAGEOBJ_WORDS (PAGEWORDS-1)
#define HEADER_PAGES( header ) ((HEADER_WORDS( header )+PAGEWORDS)/PAGEWORDS)
#else
#define ONEPAGEOBJ_WORDS PAGEWORDS
#define HEADER_PAGES( header ) ((HEADER_WORDS( header )+PAGEWORDS-1)/PAGEWORDS)
#endif
#define MAX_HEADER_PAGES (0xFFFFF/PAGEWORDS)    /* 8191 =  4,193,792 bytes */
#define MAX_HEADER_CALLBACK 0x7FF               /* 2047 */

/* The first word of user objects is noted in the firstword bit map.  This
   allows gcmove to rapidly detect a derived pointer and convert it into an
   object and an offset.
*/

#define BIT_BYTES (PAGEWORDS/8)
#define BIT_WORDS (PAGEWORDS/32)
#define ISA_FIRSTWORD( p ) (firstword[ ((int)p)/(PAGEBYTES/BIT_WORDS) ] &   \
                            1<<( ((int)p)>>2 & 0x1F ))
#define SET_FIRSTWORD( p ) (firstword[ ((int)p)/(PAGEBYTES/BIT_WORDS) ] |=  \
                            1<<( ((int)p)>>2 & 0x1F ))

/* There is an option to draw the heap map using the EZX program.
   Objects are drawn on 'display'.
*/
static FILE*  display;


       /*********************************
        * Exported Interface Definitions *
        *********************************/

#ifdef COMPILER_VERSION_1_2
#include "gcalloc-1.2.h"
#endif
#ifdef COMPILER_VERSION_2_0
#include "gcalloc-2.0.h"
#endif

/* An instance of the type gcheap is created to configure the size of the
   initial heap, the expansion increment, the maximum size of the heap, the
   allocation percentage to force a total collection, the allocation
   percentage to force heap expansion, and garbage collection options.
*/

/* Default heap configuration */

const int  GCMINBYTES = 1048576,   /* # of bytes of initial heap */
           GCMAXBYTES = 2147483647,/* # of bytes of the final heap */
           GCINCBYTES = 1048576,   /* # of bytes of each increment */
           GCALLPERCENT = 35,      /* % allocated to force total
                                      collection */
           GCINCPERCENT  = 25,     /* % NEWLY allocated to force expansion */
           GCSTABLEPERCENT = 20,   /* % stable to force expansion */
           GCFLAGS = 0;            /* option flags */

/* Actual heap configuration */

static int  gcminbytes = GCMINBYTES,    /* # of bytes of initial heap */
            gcmaxbytes = GCMAXBYTES,    /* # of bytes of the final heap */
            gcincbytes = GCINCBYTES,    /* # of bytes of each increment */
```

```
            gcallpercent = GCALLPERCENT,/* % allocated to force total
                                           collection */
            gcincpercent = GCINCPERCENT,/* % NEWLY allocated to force
                                           expansion */
            gcstablepercent=GCSTABLEPERCENT,/* %stable to force expansion */
            gcflags = GCFLAGS,          /* option flags */
            gcdefaults = 1,             /* default setting in force */
            gcheapcreated = 0;          /* boolean indicating heap created */

gcheap::gcheap( int minheapbytes,
                int maxheapbytes,
                int incheapbytes,
                int allpercent,
                int incpercent,
                int stablepercent,
                int flags )  {

        if  (gcheapcreated == 0  &&  minheapbytes > 0  &&
             (gcdefaults || maxheapbytes >= gcmaxbytes))  {
           gcdefaults = 0;
           gcminbytes = minheapbytes;
           gcmaxbytes = maxheapbytes;
           gcincbytes = incheapbytes;
           gcallpercent = allpercent;
           gcincpercent = incpercent;
           gcstablepercent = stablepercent;
           if  (gcminbytes < 4*PAGEBYTES)  gcminbytes = 4*PAGEBYTES;
           if  (gcmaxbytes < gcminbytes)  gcmaxbytes = gcminbytes;
           if  (gcallpercent < 0  ||  gcallpercent > 50)
              gcallpercent = GCALLPERCENT;
           if  (gcincpercent < 0  ||  gcincpercent > 50)
              gcincpercent = GCINCPERCENT;
           if  (gcstablepercent < 0  ||  gcstablepercent > 50)
              gcstablepercent = GCSTABLEPERCENT;
        }
        gcflags = gcflags | flags;
}


/* The following structure contains the callback procedures registered with
   the garbage collector.  It is allocated from the non-garbage collected
   heap.
*/

static int  callbacks_count = 0;
static int  callbacks_size = 0;
static const int  callbacks_inc = 100;

static struct  callback_struct  {
        GCCALLBACKPROC  proc;   /* GCPointers method */
        char*  type;            /* Type name */
        int  number;            /* Number of the type in heap */
        int  bytes;             /* Number of bytes of the type in heap */
}  *callbacks;

/* Freespace objects have a null callback that stored in callbacks[ 0 ].  Pad
   objects for double alignment have a null callback in callbacks[ 1 ].  The
   header for a one-word double alignment pad is kept in doublepad.
*/
```

```
static int  freespace_callback = MAKE_CALLBACK( 0 );

static  void freespace_pointers( GCP dummy ) {};

#ifdef DOUBLE_ALIGN
static int  doublepad;
#endif

/* Callback procedures are "registered" with the garbage collector by the
   following procedure.  Calls to it are hidden inside GCALLOC and GCALLOCV.
*/

int  gcregistercallback( GCCALLBACKPROC  proc, char* type )  {
        if  (callbacks_count > MAX_HEADER_CALLBACK)  {
           fprintf( stderr, "\n***** gcalloc  %d classes already defined\n",
                    MAX_HEADER_CALLBACK-1 );
           abort();
        }
        if  (callbacks_count == callbacks_size)  {
           callback_struct*  np = new callback_struct[ callbacks_size+
                                                       callbacks_inc ];
           for  (int i=0; i < callbacks_count; i++)  np[ i ] = callbacks[ i ];
           delete  callbacks;
           callbacks = np;
           callbacks_size = callbacks_size+callbacks_inc;
           if  (callbacks_count == 0)  {
              callbacks[ 0 ].proc = freespace_pointers;
              callbacks[ 0 ].type = "GCFreeSpace";
              callbacks[ 1 ].proc = freespace_pointers;
              callbacks[ 1 ].type = "GCDoublePad";
              callbacks_count = 2;
           }
        }
        callbacks[ callbacks_count ].proc = proc;
        callbacks[ callbacks_count ].type = type;
        return  MAKE_CALLBACK( callbacks_count++ );
}


       /***************************
        * Mostly Copying Collector *
        ***************************/

/* Get heap configuration information from the environment.  Return true if
   the value is provided.
*/

static int  environment_value( char* name, int& value )
{
        char* valuestring = getenv( name );

        if (valuestring != NULL)  {
           value = atoi( valuestring );
           return  1;
        }
        return  0;
}

/* When run with GCHEAPMAP flag set (see gcalloc-??.h), a graphical
   display will appear on the screen to monitor the allocation/
```

```
   deallocation activities in the heap.

   The graphical display is set up by means of a pipe to an EZX process.
   Garbage collection can be stepped using the mouse buttons:

        button 1 - advance to the next step.
        button 2 - enable/disable stepping.
        button 3 - Postscript for display to gcheap.PSF.
*/

/* Colors to use for free storage. */

static  char* freecolors[16] =
        { 0, "red", 0, "turquoise", 0, "green", 0, "yellow",
          0, "red", 0, "turquoise", 0, "green", 0, "yellow" };

/* The following procedure writes the header on the heap display. */

static void  display_headers( char* phase )
{
        fputs( "(object header", display );
        fprintf( display, "(fill-rectangle 10 0 10 10 %s)",
                 freecolors[ (curr_space % 8)+2 ] );
        fprintf( display, "(fill-rectangle 22 0 10 10 %s)",
                 freecolors[ (curr_space % 8)+4 ] );
        fprintf( display, "(fill-rectangle 34 0 10 10 %s)",
                 freecolors[ (curr_space % 8)+6 ] );
        fputs( "(text 50 10 \"Free space\" \"8x13\")", display );
        fprintf( display, "(fill-rectangle 140 0 10 10 %s)",
                 freecolors[ (curr_space % 8) ] );
        fputs( "(text 156 10 \"Recently Allocated\" \"8x13\")", display );
        fputs( "(fill-rectangle 310 0 10 10 black)", display );
        fputs( "(text 326 10 \"Stable set\" \"8x13\")", display );
        fprintf( display, "(text 420 10 \"%s\" \"8x13\")", phase );
        fputs( ")", display );
        fputs( "(step #t)", display );
        fflush( display );
}

/* Each page is represented by a square of the following size.  */

#define  PAGE_PIXELS  5

/* The following procedure is called to create the heap display. */

static void  displayinit()
{
        int  toezx[ 2 ];

        if  ((gcflags & GCHEAPMAP) == 0)  return;
        /* Spawn off an ezx process */
        pipe( toezx );
        if  (fork() == 0)  {
           close( 0 );
           dup( toezx[ 0 ] );
           close( toezx[ 0 ] );
           close( toezx[ 1 ] );
           execlp( "ezx", "ezx", 0 );
           exit( 1 );
        }
```

```
        display = fdopen( toezx[ 1 ], "w" );
        /* Initialize the display */
        fprintf( display,
                "(window heap 0 0 %d 220 \"C++ Garbage Collected Heap\")\n",
                PAGE_PIXELS*128 );
        fputs( "(click 1 0 0 0 0 (next-step))", display );
        fputs( "(click 2 0 0 0 0 (set! *stepper* (not *stepper*)))", display );
        fputs( "(click 3 0 0 0 0 (ezx-command '(postscript \"gcppheap.PSF\")))",
               display );
        display_headers( "Application Allocation" );
}

/* A page is colored on the heap map by the following function. */

static void  page_map( int page )
{
        char*  color;

        if  (STABLE( page ))
           color = "BLACK";
        else
           color = freecolors[ space[ page ] % 8 ];
        page = page-firstheappage;
        fprintf( display, "(object p%x (fill-rectangle %d %d %d %d %s))",
                 page, (page & 127)*PAGE_PIXELS,
                 PAGE_PIXELS*(page/2048)+(page/128)*PAGE_PIXELS+20,
                 PAGE_PIXELS, PAGE_PIXELS, color );
        fflush( display );
}

/* The heap is allocated and the appropriate data structures are initialized
   by the following function.  It is called the first time any storage is
   allocated from the heap.
*/

static void  gcinit( )
{
        char  *heap;
        int  i;

        /* Log actual heap parameters if from environment or logging */
        if  ((environment_value( "GCMINBYTES", gcminbytes ) |
              environment_value( "GCMAXBYTES", gcmaxbytes ) |
              environment_value( "GCINCBYTES", gcincbytes ) |
              environment_value( "GCALLPERCENT", gcallpercent ) |
              environment_value( "GCINCPERCENT", gcincpercent ) |
              environment_value( "GCSTABLEPERCENT", gcstablepercent ) |
              environment_value( "GCFLAGS", gcflags ))  ||
              gcflags & GCSTATS)  {
           fprintf( stderr,
                    "***** YIP gcalloc gcheap( %d, %d, %d, %d, %d, %d, %d )\n",
                    gcminbytes, gcmaxbytes, gcincbytes, gcallpercent,
                    gcincpercent, gcstablepercent, gcflags );
        }

        void page_fault_handler( int sig, int code, struct sigcontext *scp );
#ifdef mips
        signal( SIGSEGV, (SIG_PF)page_fault_handler );
#endif
#ifdef vax
```

```
        signal( SIGBUS, (SIG_PF)page_fault_handler );
#endif

        /* Record system parameters and assign page mask values */
        int physical_pagebytes = getpagesize();
        phys_cap = physical_pagebytes/PAGEBYTES;
        phys_pagemask = ~(PHYS_PAGEBYTES-1);
        heap_pagemask = ~(PAGEBYTES-1);

        /* heap size rounded up to the nearest physical page size */
        physspanpages = (gcminbytes+PHYS_PAGEBYTES-1)/PHYS_PAGEBYTES;
        heapspanpages = heappages = physspanpages*phys_cap;

        /* Allocate heap and side tables.  Exit on allocation failure. */
        if  ((heap=new char[ heappages*PAGEBYTES+PHYS_PAGEBYTES-1 ]) == NULL)
           goto fail;
        if ((unsigned)heap & (PHYS_PAGEBYTES-1))
           heap = heap+(PHYS_PAGEBYTES-((unsigned)heap&(PHYS_PAGEBYTES-1)));
        firstheappage = GCP_to_PAGE( heap );
        lastheappage = firstheappage+heapspanpages-1;
        if  ((space = new int[ heapspanpages ]) == NULL    ||
             (plink = new int[ heapspanpages]) == NULL     ||
             (type = new int[ heapspanpages+1 ]) == NULL   ||
             (protect = new int[ physspanpages ]) == NULL  ||
             (firstword = new int[ (heapspanpages+1)*BIT_WORDS ]) == NULL) {
fail:      fprintf( stderr,
                    "\n****** gcalloc  Unable to allocate %d byte heap\n",
                    gcminbytes );
           abort();
        }
        space = space-firstheappage;
        plink = plink-firstheappage;
        type = type-firstheappage;
        type[lastheappage+1] = OBJECT;
        firstword = firstword-firstheappage*BIT_WORDS;
        SET_FIRSTWORD( ((int*)((lastheappage+1)*PAGEBYTES)) + 2 );

        /* Initialize tables */
        for (i = firstheappage ; i <= lastheappage ; i++)
           space[ i ] = FREEPAGE;
        bzero( (char*)protect, physspanpages*sizeof(int) );
        protect = protect - (PAGE_to_ADDR(firstheappage)>>phys_shift);

        static void setup_endangered_rec();
        setup_endangered_rec();

        curr_space = 3;
        forw_space = 3;
        prev_space = 3;
        curr_freepage = firstheappage;
        curr_freewords = 0;
        allocatedpages = 0;
        forwardedpages = 0;
        currentpages = 0;
        protectedpages = 0;
        queue_head = 0;
        gcheapcreated = 1;
#ifdef DOUBLE_ALIGN
        doublepad = MAKE_HEADER( 1, MAKE_CALLBACK( 1 ) );
#endif
```

```
        displayinit();
}

/* Once the heap has been allocated, it is expanded after garbage
   collection whenever it is appropriate until the maximum size is
   reached.  If space cannot be allocated to expand the heap, then the
   heap will be left at its current size and no further expansions will
   be attempted.  SHOULDEXPANDHEAP is a boolean that returns true when
   the heap should be expanded.  EXPANDHEAP is called to expand the
   heap.  It returns true when the heap could be expanded.
*/

static int  shouldexpandheap()
{
        if  (HEAPPERCENT( allocatedpages ) < gcincpercent  ||
             heappages >= gcmaxbytes/PAGEBYTES  ||  gcincbytes == 0)
           return  0;
        else
           return  1;
}

static  expandfailed = 0;

static int  expandheap()  {

        int  incphyspages = (gcincbytes+PHYS_PAGEBYTES-1)/PHYS_PAGEBYTES,
             incheappages = incphyspages*phys_cap,
             new_firstheappage = firstheappage,
             inc_firstheappage,
             new_lastheappage = lastheappage,
             inc_lastheappage,
             new_heappages,
             new_heapspanpages,
             new_physspanpages,
             *new_space = NULL,
             *new_plink = NULL,
             *new_type = NULL,
             *new_protect = NULL,
             *new_scanned = NULL,
             *new_firstword = NULL,
             i;
        char*  heap;

        /* Check for previous expansion failure */
        if  (expandfailed)  return  0;

        /* Allocate additional heap and determine page span */
        heap = new char[ incheappages*PAGEBYTES+PHYS_PAGEBYTES-1 ];
        if  (heap == NULL)  goto fail;
        if ((unsigned)heap & (PHYS_PAGEBYTES-1))
           heap = heap+(PHYS_PAGEBYTES-((unsigned)heap&(PHYS_PAGEBYTES-1)));
        inc_firstheappage = GCP_to_PAGE( heap );
        inc_lastheappage = inc_firstheappage+incheappages-1;
        if  (inc_firstheappage < firstheappage)
           new_firstheappage = inc_firstheappage;
        if  (inc_lastheappage > lastheappage)
           new_lastheappage = inc_lastheappage;
        new_heappages = heappages+incheappages;
        new_heapspanpages = new_lastheappage-new_firstheappage+1;
        new_physspanpages = new_heapspanpages/phys_cap;
```

```
        /* Allocate contiguous space for each side table, recover gracefully
           from allocation failure.  */
        if  ((new_space = new int[ new_heapspanpages ]) == NULL   ||
             (new_plink = new int[ new_heapspanpages ]) == NULL   ||
             (new_type = new int[ new_heapspanpages+1 ]) == NULL  ||
             (new_protect = new int[ new_physspanpages ]) == NULL ||
             (new_scanned = new int[ new_heapspanpages ]) == NULL ||
             (new_firstword = new int[ (new_heapspanpages+1)*BIT_WORDS ])
                == NULL) {
fail:      if  (heap)  delete heap;
           if  (new_space)  delete new_space;
           if  (new_plink)  delete new_plink;
           if  (new_type)  delete new_type;
           if  (new_protect) delete new_protect;
           if  (new_scanned) delete new_scanned;
           if  (new_firstword)  delete new_firstword;
           expandfailed = 1;
           if  (gcflags & GCSTATS)
              fprintf( stderr, "\n***** gcalloc  Heap expansion failed\n" );
           return  0;
        }
        new_space = new_space-new_firstheappage;
        new_plink = new_plink-new_firstheappage;
        new_type = new_type-new_firstheappage;
        new_firstword = new_firstword-new_firstheappage*BIT_WORDS;

        /* Initialize new side tables, delete the old ones */

        bzero( (char*)new_scanned, new_heapspanpages*sizeof(int) );
        new_scanned = new_scanned-new_firstheappage;

        for (i = new_firstheappage ; i < firstheappage ; i++)
           new_space[ i ] = UNALLOCATEDPAGE;
        for (i = firstheappage ; i <= lastheappage ; i++) {
           new_space[ i ] = space[ i ];
        }
        for (i = lastheappage+1 ; i < new_lastheappage ; i++)
           new_space[ i ] = UNALLOCATEDPAGE;
        for (i = inc_firstheappage ; i <= inc_lastheappage ; i++)
           new_space[ i ] = FREEPAGE;
        delete (space+firstheappage);
        space = new_space;

        for  (i = firstheappage; i <= lastheappage; i++)  {
           new_plink[ i ] = plink[ i ];
           new_type[ i ] = type[ i ];
        }
        delete (plink+firstheappage);
        plink = new_plink;
        delete (type+firstheappage);
        type = new_type;

        bzero( (char*)new_protect, new_physspanpages*sizeof(int) );
        new_protect =
             new_protect - (PAGE_to_PPADDR(new_firstheappage)>>phys_shift);
        for (int addr_physpage = PAGE_to_ADDR(firstheappage);
             addr_physpage  <  PAGE_to_ADDR(lastheappage);
             addr_physpage += PHYS_PAGEBYTES )
           *(new_protect+(addr_physpage>>phys_shift)) =
                           is_physpage_protected( addr_physpage );
```

```
        delete (protect+(PAGE_to_ADDR(firstheappage)>>phys_shift));
        protect = new_protect;

        for  (i = firstheappage*BIT_WORDS;
              i <= lastheappage*BIT_WORDS+BIT_WORDS-1; i++)
           new_firstword[ i ] = firstword[ i ];
        delete (firstword+firstheappage*BIT_WORDS);
        firstword = new_firstword;

        /* To facilitate easy implementation of NEXT_OBJECT, SMALL_CLUSTER,
           and DETERMINE_PHYSPAGE_CLUSTER, the following (phony) bookkeeping
           is made about the non-existing "page" just beyond the heap */
        new_type[ new_lastheappage+1 ] = OBJECT;
        SET_FIRSTWORD( ((int*)((new_lastheappage+1)*PAGEBYTES)) + 2 );

        firstheappage = new_firstheappage;
        lastheappage = new_lastheappage;
        heappages = new_heappages;
        heapspanpages = new_heapspanpages;
        physspanpages = new_physspanpages;

        if  (gcflags & GCSTATS)
           fprintf( stderr, "\n***** gcalloc  Heap expanded to %d bytes\n",
                    heappages*PAGEBYTES );

        return  1;
}

/* A pointer pointing to the header of an object is stepped to the next
   header by the following function.
*/

static GCP  next_object( GCP xp )
{
        xp++;     /* xp now points at body of object */
        while  (xp++)
           if  (ISA_FIRSTWORD( xp ))
              return (xp-1);
}

/* A pointer can be verified to point to an object in the heap by the
   following function.  An invalid pointer will be logged and the program
   will abort.
*/

static void  verify_object( GCP cp, int old )
{
        int  page = GCP_to_PAGE( cp );
        GCP  xp = PAGE_to_GCP( page ); /* Ptr to start of page */
        int  error = 0;

        if  (page < firstheappage)  goto fail;
        error = 1;
        if  (page > lastheappage)  goto fail;
        error = 2;
        if  (space[ page ] == UNALLOCATEDPAGE)  goto fail;
        error = 3;
        if  (old  &&  UNSTABLE( page )  &&  space[ page ] != prev_space)
           goto fail;
        error = 4;
```

```
        if  (old == 0  &&  space[ page ] != forw_space)  goto fail;
        error = 5;
        if  (callbacks[ HEADER_CALLBACK( cp[-1] ) ].proc == freespace_pointers)
           return;
        while  (cp > xp+1)  xp = next_object( xp );
        if  (cp == xp+1)  return;
fail:
        fprintf( stderr, "\n***** gcalloc  invalid pointer " );
        fprintf( stderr,
                "error: %d  pointer: 0x%x nextcp 0x%x xp 0x%x\n",
                 error, cp, cp+HEADER_WORDS(*cp), xp );
        abort();
}

/* An object's header is verified by the following function.  An invalid
   header will be logged and the program will abort.
*/

static void  verify_header( GCP cp )
{
        int  size = HEADER_WORDS( cp[ -1 ] ),
             page = GCP_to_PAGE( cp ),
             error = 0;

        if  FORWARDED( cp[ -1 ] )  goto fail;
        error = 1;
        if  ((unsigned)HEADER_CALLBACK( cp[ -1 ] ) >=
             (unsigned)callbacks_count)  goto fail;
        if  (size <= ONEPAGEOBJ_WORDS)  {
           error = 2;
           if  (cp-1+size > PAGE_to_GCP( page+1 ))  goto fail;
        } else  {
           error = 3;
           int  pages = HEADER_PAGES( cp[ -1 ] ),
                pagex = page;
           while  (--pages)  {
              pagex++;
              if  (pagex > lastheappage  ||
                   type[ pagex ] != CONTINUED  ||
                   space[ pagex ] != space[ page ])
                 goto fail;
           }
        }
        return;
fail:   fprintf( stderr,"\n***** gcalloc  invalid header  ");
        fprintf( stderr,
                "error: %d  object&: 0x%x  header: 0x%x\n",
                 error, cp, cp[ -1 ] );
        abort();
}

/* The consistency of the heap is checked by the following function. */

static void  verify_heap()
{
        GCP cp, lastcp;
        for  (int page = firstheappage; page < lastheappage; page++) {
            cp = PAGE_to_GCP( page );
            if  ( type[page] == OBJECT &&
                 ( space[page] == curr_space ||
```

```
                (STABLE(page) && space[ page ] != UNALLOCATEDPAGE) ) ) {
            lastcp = PAGE_to_GCP(page+1) - 1;
            while  (cp < lastcp &&
                    (curr_freewords == 0 || cp != curr_freep)) {
                verify_object( cp+1, 1);
                verify_header( cp+1 );
                cp = cp+HEADER_WORDS( *cp );
            }
         }
      }
}

/* The following variable holds the number of pointers guessed in an object
   by guess_pointer.  Note that cp points to the object header.
*/

static int  guess_pointer_count;

static int  guess_pointers( GCP cp )
{
      guess_pointer_count = 0;
      if  (HEADER_CALLBACK( *cp ) != 0)  {
         for  (int i=1; i < HEADER_WORDS( *cp ); i++)  {
            int  page = GCP_to_PAGE( (GCP)cp[ i ] );
            if  (page >= firstheappage  &&  page <= lastheappage  &&
                 space[ page ] != UNALLOCATEDPAGE)
               guess_pointer_count++;
         }
      }
      return  guess_pointer_count;
}

/* The stable set is moved into the current_generation by the following
   function.  A total collection is performed by calling this before calling
   gccollect.  When generational collection is not desired, this is called
   after collection to empty the stable set.
*/

static void  makecurrentstableset()
{
      int pagecount, i;
      while  (queue_head)  {
#ifdef DOUBLE_ALIGN
         pagecount = HEADER_PAGES( *(PAGE_to_GCP( queue_head )+1) );
#else
         pagecount = HEADER_PAGES( *PAGE_to_GCP( queue_head ) );
#endif
         i = queue_head;
         while  (pagecount--)  {
            space[ i++ ] = curr_space;
            if  (gcflags & GCHEAPMAP)  page_map( i-1 );
         }
         queue_head = plink[ queue_head ];
      }

      currentpages = allocatedpages;
      forwardedpages = 0;
}

/* A page index is advanced by the following function */
```

```
static inline int  next_page( int page )
{
        return  (page == lastheappage) ? firstheappage : page+1;
}

/* A page is added to the stable set page queue by the following function. */

static void  queue( int page )
{
        if  (queue_head != 0)
           plink[ queue_tail ] = page;
        else
           queue_head = page;
        plink[ page ] = 0;
        queue_tail = page;
}

/* Pages that have might have references in the stack or the registers are
   promoted to the forward space (which is stable) by the following function.
*/

static void  promote_page( int page )
{
        if  (page >= firstheappage  &&  page <= lastheappage  &&
             space[ page ] == prev_space)  {
           if  (type[ page ] == CONTINUED)  {
              while  (type[ --page ] == CONTINUED);
           }
#ifdef DOUBLE_ALIGN
           int  pagecount = HEADER_PAGES( *(PAGE_to_GCP( page )+1) );
#else
           int  pagecount = HEADER_PAGES( *PAGE_to_GCP( page ) );
#endif
           if  (gcflags & GCDEBUGLOG)
              fprintf( stderr, "promoted 0x%x\n", PAGE_to_GCP( page ) );
           queue( page );
           forwardedpages += pagecount;
           if  (gcflags & GCHEAPMAP)  {
              for  (int i = 0; i < pagecount; i++)  {
                 space[ page+i ] = forw_space;
                 page_map( page+i );
              }
           }
           else  do  {
                    space[ page++ ] = forw_space;
                 }  while (--pagecount);
        }
}

        /****************************************
         * Protecting/unprotecting physical pages *
         ****************************************/

/* Test whether a physical page is protected. */

#define is_physpage_protected( addr )   (*(protect+(addr>>phys_shift)))

/* The following routine registers that a physical page is protected. */

inline static void  physpage_set_protected( int addr_physpage )
```

```
{
        *(protect+(addr_physpage>>phys_shift)) = 1;
        protectedpages++;
}

/* The following routine registers that a physical page is unprotected. */

inline static void  physpage_set_unprotected( int addr_physpage )
{
        *(protect+(addr_physpage>>phys_shift)) = 0;
        protectedpages--;
}

#define NO_ACCESS 0   /* Page access code (argument to mprotect) */

/* Inline wrapper for call to mprotect. */

static inline void call_mprotect( void* addr, int numbytes, int protection )
{
           mprotect( addr, numbytes, protection );
}

/* An endangered physical page is one that has to be protected before
   the garbage collector returns control to the application.  The
   following routine marks a physical page as endangered.
*/
extern void add_endangered_physpage( int addr_physpage );

/* Boolean to indicate whether scanning is going on */
extern int scanning_physpage;

/* The following routine protects a physical page and registers it as
   protected.
*/
inline static void  protect_physpage( int addr_physpage )
{
        if  ( !is_physpage_protected( addr_physpage ) ) {
           /* if scanning is going on, then delay protecting of
              the physical page.  Mark it as endangered. */
           if  ( scanning_physpage )
              add_endangered_physpage( addr_physpage );
           else {
              call_mprotect( (void*)addr_physpage, PHYS_PAGEBYTES, NO_ACCESS );
              physpage_set_protected( addr_physpage );
           }
        }
}

/* The following routine unprotects a physical page and registers it as
   unprotected.
*/
inline static void  unprotect_physpage( int addr_physpage )
{
        if  ( is_physpage_protected( addr_physpage ) ) {
           call_mprotect( (void*)addr_physpage, PHYS_PAGEBYTES, PROT_WRITE );
           physpage_set_unprotected( addr_physpage );
        }
}

/* A physical page cluster is the smallest set of contigous pages in the heap
```

```
   which needs to be scanned.  The following three functions determine,
   protect and unprotect, respectivenly, the physical page cluster containing
   the specified physical page address.
*/

static void  determine_physpage_cluster( int& addr_physpage,
                                         int& num_physpages )
{
        num_physpages = 1;
        while  (type[ ADDR_to_PAGE(addr_physpage) ] != OBJECT) {
           addr_physpage -= PHYS_PAGEBYTES;
           num_physpages++;
        }
        while  (type[ADDR_to_PAGE(addr_physpage+PHYS_PAGEBYTES*num_physpages)]
                !=OBJECT)
           num_physpages++;
}

static void  protect_physpage_cluster( int addr_physpage, int num_physpages=0 )
{
        if  (!num_physpages)
           determine_physpage_cluster( addr_physpage, num_physpages );

        if  (num_physpages == 1) {
           if  ( !is_physpage_protected( addr_physpage ) ) {
              call_mprotect( (void*)addr_physpage, PHYS_PAGEBYTES, NO_ACCESS );
              physpage_set_protected( addr_physpage );
           }
        }
        else {
           for  (; num_physpages > 0;
                   num_physpages--, addr_physpage += PHYS_PAGEBYTES) {
               if  ( !is_physpage_protected( addr_physpage ) ) {
                  call_mprotect( (void*)addr_physpage,
                                 num_physpages*PHYS_PAGEBYTES, NO_ACCESS );
                  break;
               }
           }
           while  (num_physpages--) {
              if  (!is_physpage_protected( addr_physpage ))
                 physpage_set_protected( addr_physpage );
              addr_physpage += PHYS_PAGEBYTES;
           }
        }
}

static void  unprotect_physpage_cluster(int addr_physpage,int num_physpages=0)
{
        if  (num_physpages)
           call_mprotect( (void*)addr_physpage, num_physpages*PHYS_PAGEBYTES,
                          PROT_WRITE);
        else {
           determine_physpage_cluster( addr_physpage, num_physpages );
           call_mprotect( (void*)addr_physpage, num_physpages*PHYS_PAGEBYTES,
                          PROT_WRITE );
        }
        while  (num_physpages--) {
           if  ( is_physpage_protected( addr_physpage ) )
              physpage_set_unprotected( addr_physpage );
           addr_physpage += PHYS_PAGEBYTES;
```

```
        }
}


/* Test whether two addresses are residing on the same physical page. */

static inline int  same_physpage( int addr_a, int addr_b ) {
        return ((addr_a & phys_pagemask) == (addr_b & phys_pagemask));
}

/* Is the address inside the region being scanned? */

static inline int  inside_scan_region( int addr ) {
        return ( firstword_to_scan <= addr && addr <= lastword_to_scan );
}

/* The following routine performs collector allocation to allocate a
   free heap page in forward-space.  If space is not available then
   the heap is expanded.

   The argument to allocate_forwpage, badptr, is necessary so that any heap
   page on the physical page that badptr is on will not be allocated.
*/

static void  allocate_forwpage( GCP badptr )
{
        int allpages = heapspanpages;

        while  (allpages--)  {
           if  (space[ forw_freepage ] < prev_space  &&
                UNSTABLE( forw_freepage ) &&
                !same_physpage( (int)badptr, PAGE_to_ADDR(forw_freepage) ) &&
                !inside_scan_region( PAGE_to_ADDR(forw_freepage) )) {
              forw_freewords = PAGEWORDS;
              allocatedpages++;
              forwardedpages++;
              space[ forw_freepage ] = forw_space;
              type[ forw_freepage ] = OBJECT;
              bzero( (char*)&firstword[forw_freepage*BIT_WORDS], BIT_BYTES );
              forw_freep = PAGE_to_GCP( forw_freepage );
              if  (gcflags & GCHEAPMAP)  {
                 page_map( forw_freepage );
              }
              queue( forw_freepage );
              /* SWAP old_forw_freepage and forw_freepage
                 to improve locality of stable pages      */
              int temp = old_forw_freepage;
              old_forw_freepage = next_page( forw_freepage );
              forw_freepage = temp;
              return;
           }
           else forw_freepage = next_page( forw_freepage );
        }
        /* Failed to allocate space, keep trying iff heap can expand. */
        if  (expandheap())  {
           allocate_forwpage( badptr );
           return;
        }
        /* Can't do it */
        fprintf( stderr, "\n***** allocate_forwpage  " );
```

```
        fprintf( stderr,
                "Unable to allocate %d bytes in a %d byte heap\n",
                 PAGEBYTES, heappages*PAGEBYTES );
        abort();
}

/* REGISTER_GCMOVE_PROMOTED_PAGES remembers the heap pages that GCMOVE
   promotes inside the scan region.  REGISTER_GCMOVE_PROMOTED_PAGES
   first determines whether the scanning pointer has swept past the newly-
   promoted pages already.  If so, the pages are registered so that they
   will be scanned later; otherwise, these pages are guaranteed to have been
   scanned later on anyway.
*/

/* An object is scanned iff it is in the scan region and its address is
   less than the address of the object being scanned (i.e. scanp).
*/
#define SCANNED( addr )  ( addr < int(scanp) )

#define MAX_RESCAN_CHUNKS 8          /* Max number of chunks to rescan */
static  int rescan_all  = 0;         /* Flag to rescan entire scan region */
static  int rescanchunks = 0;        /* Chunk count */
static  int rescan1[ MAX_RESCAN_CHUNKS ];  /* Record chunks to rescan */
static  int rescan2[ MAX_RESCAN_CHUNKS ];  /* Alternate with rescan1 */
static  int *rescan = rescan1;             /* Ptr to current rescan record */

static void  register_gcmove_promoted_pages( int promoted_page_head )
{
        if  (rescan_all) return;

        if  (SCANNED( PAGE_to_ADDR( promoted_page_head ) ))  {
           if  (rescanchunks < MAX_RESCAN_CHUNKS)
              rescan[ rescanchunks++ ] = promoted_page_head;
           else rescan_all = 1;
        }
}

/* An object is forwarded by the following function.  The forwarded
   object must be protected because it is yet to be scanned.
*/

GCP  gcmove( GCP cp )
{
        int  page = GCP_to_PAGE( cp ),                    /* Page number */
             header,                                     /* Object header */
             freep_ppaddr = GCP_to_PPADDR(forw_freep);  /* freep's PPADDR */
        GCP  np;                              /* Pointer to the new object */
        int  addr_physpage = 0;         /* Object's Physical Page ADDRess */

        /* If out of heap then ok */
        if  (page < firstheappage  ||  page > lastheappage  ||
             space[ page ] == UNALLOCATEDPAGE)
           return( cp );

        /*  If object in stable storage or in current space then ok */
        if  (STABLE( page ) || space[page] == curr_space)
                return( cp );

        if  ( space[page] != prev_space ) {
           fprintf(stderr,
```

```
            "gcmove: space[%d] = %d, cp = 0x%x, prev_space %d\n",
            page, space[page], cp, prev_space);
   abort();
}

/* Check for a derived pointer */
if  (((int)cp) & 3  ||  ISA_FIRSTWORD( cp ) == 0)  {
   while  (type[ page ] == CONTINUED)  page--;
   GCP  p1, p2 = PAGE_to_GCP( page );
   if  (gcflags & GCTSTOBJ)  verify_object( p2, 1 );
   while  (p2 < cp)  {
      p1 = p2;
      p2 = next_object( p2 );
   }
   return (GCP)((char*)gcmove( p1+1 )+((char*)cp-(char*)(p1+1)));
}

/* Object maybe protected */
if  (!inside_scan_region((int)cp) && is_physpage_protected((int)cp)) {
   addr_physpage = GCP_to_PPADDR( cp );
   unprotect_physpage( addr_physpage );
   /* Addr_physpage must be reprotected on exit from gcmove */
}

/* Verify that the object is a valid pointer and decrement ptr cnt */
if  (gcflags)  {
   if  (gcflags & GCTSTOBJ)  verify_object( cp, 1 );
   if  (gcflags & GCGUESSPTRS)  guess_pointer_count--;
}

/* If cell is already forwarded, return forwarding pointer */
header = cp[-1];
if  (FORWARDED( header ))  {
   if  (gcflags & GCTSTOBJ)  {
      verify_object( (GCP)header, 0 );
      verify_header( (GCP)header );
   }
   if  (addr_physpage) {
      protect_physpage( addr_physpage );
   }
   return( (GCP)header );
}

/* Move the object */
if  (gcflags & GCTSTOBJ)  {
   verify_header( cp );
}
int freep_outside_scan_region=!inside_scan_region((int)forw_freep);

/* Forward or promote object */
int  words = HEADER_WORDS( header );
if  (words >= forw_freewords) {

   /* Objects >= a page are promoted to the stable set */
   if  (words >= ONEPAGEOBJ_WORDS)  {
      int pagecount = HEADER_PAGES( *(cp-1) );
      forwardedpages += pagecount;
      queue( page );
      for  (int i = 0; i < pagecount; i++)
          space[ page+i ] = forw_space;
```

```
            if  ( inside_scan_region( PAGE_to_ADDR(page) ) )
               register_gcmove_promoted_pages( page );
            else {
                    protect_physpage_cluster( PAGE_to_PPADDR( page ) );
            }
            if  (gcflags & GCHEAPMAP) {
               for  (int i = 0; i < pagecount; i++)
                   page_map( page+i );
            }
            if  (addr_physpage) {
               protect_physpage( addr_physpage );
            }
            return( cp );
         }

         /* Discard any partial page and allocate a new one */
         if  (forw_freewords != 0)  {
            if  ( freep_outside_scan_region &&
                  is_physpage_protected( freep_ppaddr ) &&
                 !same_physpage( (int)cp, freep_ppaddr ) ) {
               unprotect_physpage( freep_ppaddr );
               *forw_freep = MAKE_HEADER(forw_freewords, freespace_callback);
               forw_freewords = 0;
               protect_physpage( freep_ppaddr );
            }
            else {  /* No need to unprotect/protect forw_freep */
               *forw_freep = MAKE_HEADER(forw_freewords, freespace_callback);
               forw_freewords = 0;
            }
         }
         allocate_forwpage( cp );
         freep_outside_scan_region=!inside_scan_region((int)forw_freep);
         freep_ppaddr = GCP_to_PPADDR(forw_freep);

#ifdef DOUBLE_ALIGN
         if  ( freep_outside_scan_region &&
               is_physpage_protected( freep_ppaddr )) {
            unprotect_physpage( freep_ppaddr );
         }
         *forw_freep = doublepad;
         forw_freewords--;
         forw_freep++;
         goto forward_object;
#endif

      }  /* end if (words >= forw_freewords) */

      /* Forward object, leave forwarding pointer in old object header */

      if  ( freep_outside_scan_region &&
            is_physpage_protected( freep_ppaddr ) )  {
         unprotect_physpage( freep_ppaddr );
      }
forward_object:
      *forw_freep++ = header;
      np = forw_freep;
      SET_FIRSTWORD( forw_freep );
      cp[-1] = (int)np;
      forw_freewords -= words;
      while  (--words)  *forw_freep++ = *cp++;
```

```
#ifdef DOUBLE_ALIGN
        if  ((forw_freewords & 1) == 0  &&  forw_freewords)  {
           *forw_freep = doublepad;
           forw_freewords--;
           forw_freep++;
        }
#endif
        if  ( freep_outside_scan_region )  {
           protect_physpage( freep_ppaddr );
        }
        if  (addr_physpage) {
           protect_physpage( addr_physpage );
        }
        return( np );
}

/* Pages that have been remembered with REGISTER_GCMOVE_PROMOTED_PAGES
   are re-scanned by the following function.
*/

static void rescan_gcmove_promoted_pages()
{
        int chunks = rescanchunks;
        rescanchunks = 0;
        int *rescan_record = rescan;

        /* Reset RESCAN to point to unused array */
        rescan = ( int(rescan) == int(rescan2) ) ? rescan1 : rescan2;

        GCP cp, nextcp;
        for  (int i=0; i < chunks; i++) {
            cp = PAGE_to_GCP( rescan_record[ i ] );
            nextcp = cp+PAGEWORDS;
#ifdef DOUBLE_ALIGN
            cp++;      // skip over doublepad word
#endif
            while  ( cp < nextcp &&
                    (cp != forw_freep  || forw_freewords == 0)) {
               (*callbacks[ HEADER_CALLBACK( *cp ) ].proc)( cp+1 );
               cp = cp+HEADER_WORDS( *cp );
            }
        }
}

/* To optimize performance mprotect calls are to be avoided as much as
   possible.  The following routines implements efficient synchronization
   between SCAN_PHYSPAGE_CLUSTER and GCMOVE, such that physical pages
   unprotected in GCMOVE are not reprotected until program control is
   about to exit SCAN_PHYSPAGE_CLUSTER.
*/

/* Number of physical pages that are endangered */

static int endangered_physpages=0;

/* The data structure to hold the addresses of those physical pages that
   need to be reprotected before scanning ends.
*/
const  NELEMS = 10;
```

```
static struct  array_elem {
        struct array_elem *next_array_elem;
        int array[ NELEMS ];
} *endangered_head, *endangered;

/* The following routine creates the endangered data structure. */

static void setup_endangered_rec()
{
        endangered_head = endangered = new array_elem();
}

/* The following function enters a physical page into the set of
   endangered physical pages.
*/

static void add_endangered_physpage( int ppaddr )
{
        if  ( endangered->array[(endangered_physpages-1)%NELEMS] == ppaddr ||
             (endangered_physpages%NELEMS > 1 ?
              endangered->array[(endangered_physpages-2)%NELEMS] : 0)
             == ppaddr )
           return;

        endangered->array[ endangered_physpages%NELEMS ] = ppaddr;

        if  ( ++endangered_physpages%NELEMS == 0 ) {
           endangered->next_array_elem = new array_elem();
           endangered = endangered->next_array_elem;
           endangered->next_array_elem = NULL;
        }
}

/* The following function reprotects all the endangered physical pages.
   It is called right before scanning exits and the application is about
   to resume execution.
*/

static void reprotect_endangered_physpages()
{
        for  ( endangered = endangered_head;
               endangered != NULL;
               endangered = endangered->next_array_elem ) {

            for  (int i=0; i < NELEMS && endangered_physpages-- != 0; i++)
                protect_physpage(endangered->array[ i ]);
        }

        if  (endangered_physpages != -1) abort();
        endangered_physpages = 0;
        endangered = endangered_head;
        endangered->next_array_elem = NULL;
}

/* The following is a boolean variable indicating whether program
   control inside SCAN_PHYSPAGE_CLUSTER.
*/
int scanning_physpage = 0;

/* The following function scans one physical page cluster. */
```

```
static int  scan_physpage_cluster( int addr_physpage )
{
        scanning_physpage = 1;

        /* Find and unprotect the physical page cluster region */
        int num_physpages;
        determine_physpage_cluster( addr_physpage, num_physpages );
        firstword_to_scan=addr_physpage;
        lastword_to_scan=addr_physpage+num_physpages*PHYS_PAGEBYTES-BIT_WORDS;
        unprotect_physpage_cluster( firstword_to_scan, num_physpages );

        /* Scan the cluster */
        GCP nextscanp;
rescan_physpage_cluster:
        for  (int page =  ADDR_to_PAGE(firstword_to_scan);
                  page <= ADDR_to_PAGE( lastword_to_scan);
                  page++ ) {
            if  (STABLE( page ) && type[ page ] == OBJECT) {
               scanp = PAGE_to_GCP( page );
               nextscanp = scanp+PAGEWORDS;
#ifdef DOUBLE_ALIGN
               scanp++;      // skip over doublepad word
#endif
               while  ( scanp < nextscanp &&
                        (scanp != forw_freep  || forw_freewords == 0)) {
                  (*callbacks[ HEADER_CALLBACK( *scanp ) ].proc)( scanp+1 );
                  scanp = scanp+HEADER_WORDS( *scanp );
               }
            }
        }

        /* Check if rescanning is necessary */
        while  (rescanchunks) {
           if  (rescan_all)  {
              rescan_all = 0;
              goto rescan_physpage_cluster;
           }
           else rescan_gcmove_promoted_pages();
        }
        firstword_to_scan = lastword_to_scan = NULL;

        scanning_physpage = 0;
        reprotect_endangered_physpages();

        return num_physpages;
}

/* When incremental collection is going on, the following function
   is called to scan "target" number of physical pages.
*/

int  scan_few_physpages( int target )
{
        int addr_physpage,   // Address of physical page being scanned
            num_physpages,   // Number of physical pages scanned so far
            sum = 0;         // Total number of physical pages scanned

        while  (target > 0 && protectedpages) {
            for  (addr_physpage = PAGE_to_ADDR( firstheappage );
                  addr_physpage < PAGE_to_ADDR( lastheappage ) && target > 0;
```

```
                addr_physpage += PHYS_PAGEBYTES) {
              if  ( is_physpage_protected( addr_physpage ) )  {
                 if (space[ADDR_to_PAGE(addr_physpage)] == UNALLOCATEDPAGE)
                    abort();
                 num_physpages = scan_physpage_cluster( addr_physpage );
                 sum += num_physpages;
                 target -= num_physpages;
              }
           }
        }
        firstword_to_scan = lastword_to_scan = NULL;
        return sum;
}

/* During incremental collection, physical pages where unscanned objects
   are found need to be protected.  Whenever user application accesses one of
   these pages, a page fault occurs.   The following function is the trap
   handler set up to handle these page faults.  The faulted physical page
   cluster is unprotected and scanned before becoming accessible to the
    user again.
*/

static void  page_fault_handler( int sig, int code, struct sigcontext *scp )
{
#ifdef mips
        int faulted_addr = scp->sc_badvaddr;
#endif
#ifdef vax
        int faulted_addr = code;
#endif
        int faulted_physpage = GCP_to_PPADDR( faulted_addr ),
            faulted_page = ADDR_to_PAGE( faulted_addr ),
            error = 0;

        /* Non-incremental mode should NOT cause page fault */
        if  (gcflags & GCNOINC) goto fail;
        error = 1;
        /* Error if not garbage collecting */
        if  (curr_space == forw_space) goto fail;
        error = 2;
        /* Error if faulted address is outside of heap */
        if  ( faulted_page < firstheappage || faulted_page > lastheappage)
           goto fail;
        error = 3;

        /* Finally, go ahead and scan the physical page cluster */
        scan_physpage_cluster( faulted_physpage );
        return;

fail:
        fprintf( stderr,
                "**** Page fault  error: %d  badaddr 0x%x\n",
                error, faulted_addr );
        abort();
}

/* Output a newline to stderr if logging is enabled. */

void  newline_if_logging()
{
```

```
        if  (gcflags & (GCDEBUGLOG | GCROOTLOG | GCHEAPLOG | GCGUESSPTRS))
           fprintf( stderr, "\n" );
}

/* A root is logged to stderr by the following function. */

static void  log_root( unsigned* fp )
{
        int  page = GCP_to_PAGE( fp );
        if  (page < firstheappage  ||  page > lastheappage  ||
             space[ page ] == UNALLOCATEDPAGE  ||
             (UNSTABLE( page )  &&  space[ page ] != curr_space))
           return;
        while  (type[ page ] == CONTINUED)  page--;
        GCP  p1, p2 = PAGE_to_GCP( page );
        while  (p2 < (GCP)fp)  {
           p1 = p2;
           p2 = next_object( p2 );
        }
        fprintf( stderr, "***** gcalloc  root&: 0x%x  object&: 0x%x  %s\n",
                 fp, p1, callbacks[ HEADER_CALLBACK( *(p1) ) ].type );
}

/* Log the memory use statistics on stderr. */

static void  memory_stats()
{
        GCP  cp;
        int  page = firstheappage-1;

        for  (int i = 0; i < callbacks_count; i++)  {
           callbacks[ i ].number = 0;
           callbacks[ i ].bytes = 0;
        }
        while  (++page <= lastheappage)  {
           if  ((space[ page ] == curr_space  ||
                 (STABLE( page )  &&  space[ page ] != UNALLOCATEDPAGE))
                &&  type[ page ] == OBJECT)  {
              cp = PAGE_to_GCP( page );
              while  (GCP_to_PAGE( cp ) == page  &&
                      (cp != curr_freep  || curr_freewords == 0))  {
                 int  x = HEADER_CALLBACK( *cp ),
                      words = HEADER_WORDS( *cp );
                 callbacks[ x ].number++;
                 callbacks[ x ].bytes += HEADER_BYTES( *cp );
                 if  (words > ONEPAGEOBJ_WORDS)  {
                    int  free = HEADER_PAGES( *cp )*PAGEBYTES-
                                HEADER_BYTES( *cp );

                    if  (free)  {
                       callbacks[ 0 ].number++;
                       callbacks[ 0 ].bytes += free;
                    }
                 }
                 cp = cp+words;
              }
           }
        }
        fprintf( stderr, "***** gcalloc  number     bytes  type\n" );
        for  (i = 0; i < callbacks_count; i++)  {
```

```
            fprintf( stderr, "               %6d  %8d  %s\n",
                     callbacks[ i ].number, callbacks[ i ].bytes,
                     callbacks[ i ].type );
        }
}

/* When incremental collection is over, the following function is called
   to perform miscellaneous accounting and bookkeeping tasks.
*/
extern void  gcwrapup();

/* Garbage collection is initiated by the following procedure.  It is
   typically called when one-third of the pages in the heap have been
   allocated.  It may also be directly called.
*/

static unsigned  registers[ REGISTER_COUNT ];   /* Ambiguous registers */

void  gccollect()
{
        unsigned  *fp;       /* Pointer for checking the stack */
        int page,
            freepages=0;     /* # of free heap pages */

        /* Check for heap not yet allocated */
        if  (gcheapcreated == 0)  {
           gcinit();
           return;
        }
        /* Log entry to the collector */
        if  (gcflags & GCSTATS)  {
           fprintf( stderr, "***** gcalloc  Collecting - %d%% allocated -> \n",
                    HEAPPERCENT( allocatedpages ) );
           newline_if_logging();
        }

        /* Scan a few pages and return if already collecting */
        if  (curr_space != forw_space) {
           scan_few_physpages( 1 );
           return;
        }

        /* Discard any remaining portion of current page */
        if  (curr_freewords != 0)  {
           *curr_freep = MAKE_HEADER( curr_freewords, freespace_callback );
           curr_freewords = 0;
        }

        /* Verify that heap is consistent */
        if (gcflags & GCDEBUGLOG) verify_heap();

        /* Partition regions for forwarded and newly-allocated objects */
        forw_freepage = old_forw_freepage = curr_freepage;
        forw_freep = curr_freep = PAGE_to_GCP( curr_freepage );

        /* Advance curr_freep so that _if possible_, there will be
           "RESERVEDPAGES" number of free heap pages in "forward" region */
        page = forw_freepage;
        freepages = 0;
        for  ( int i=0; freepages<=RESERVEDPAGES && i<physspanpages; i++ ) {
```

```
      if  ( UNSTABLE(page) && space[page] != curr_space ) ++freepages;
      page = next_page( page );
}
curr_freep = (GCP)PAGE_to_PPADDR( page );
curr_freepage = GCP_to_PAGE( curr_freep );

/* Change headers on heap display */
if  (gcflags & GCHEAPMAP)
   display_headers( "Starting Garbage Collection" );

/* Advance space */
currentpages = 0;
forw_space = curr_space+1;
curr_space = curr_space+2;

/* Examine stack, registers, static area and possibly the non-garbage
   collected heap for possible pointers */
if  (gcflags & GCROOTLOG)  fprintf( stderr, "stack roots:\n" );
for (fp = gcregisters( registers ) ;
     fp < (unsigned*)STACKBASE ;
     fp = (unsigned*)(((char*)fp)+STACKINC) )  {
   if  (gcflags & GCROOTLOG)  log_root( fp );
   promote_page( GCP_to_PAGE( *fp ) );
}
if  (gcflags & GCROOTLOG)
   fprintf( stderr, "static and register roots:\n" );
for  (fp = STATIC_0 ; fp < STATIC_1 ; fp++)  {
   if  (gcflags & GCROOTLOG)  log_root( fp );
   promote_page( GCP_to_PAGE( *fp ) );
}
if  (gcflags & GCHEAPROOTS  ||  gcflags & GCHEAPLOG)  {
   if  (gcflags & GCHEAPLOG)
      fprintf( stderr, "non-GC heap roots:\n" );
   unsigned*  heapend = (unsigned*)sbrk( 0 );
   unsigned*  firstheapp = (unsigned*)PAGE_to_GCP( firstheappage );
   unsigned*  lastheapp = (unsigned*)PAGE_to_GCP( lastheappage );
   while  (fp < heapend)  {
      if  (fp < firstheapp  ||  fp > lastheapp  ||
           space[ GCP_to_PAGE( fp ) ] == UNALLOCATEDPAGE)  {
         if  (gcflags & GCHEAPLOG)  log_root( fp );
         if  (gcflags & GCHEAPROOTS)
            promote_page( GCP_to_PAGE( *fp ) );
         fp++;
      }
      else  {
         fp = fp+PAGEWORDS;
      }
   }
}

/* Ambiguous roots have been promoted, now protect all the
   promoted (stable) pages */
page = queue_head;
while  (page)  {
   protect_physpage_cluster( PAGE_to_PPADDR(page) );
   page = plink[ page ];
}

if  (gcflags & GCSTATS)
   fprintf(stderr, "ambiguous roots: %d pages, %d%% of heap\n",
```

```
                forwardedpages, HEAPPERCENT( forwardedpages ));

        if  (gcflags & GCNOINC) {
           if  (gcflags & GCHEAPMAP)
              display_headers( "Copying Retained Storage" );
           do {
             for  (int addr_physpage = PAGE_to_ADDR( firstheappage );
                   addr_physpage < PAGE_to_ADDR( lastheappage );
                   addr_physpage += PHYS_PAGEBYTES)  {
                if  (is_physpage_protected( addr_physpage )) {
                   scan_physpage_cluster( addr_physpage );
                   if  ( protectedpages == 0 )
                      gcwrapup();
                }
             }
           } while ( protectedpages );
        }
        else if  (HEAPPERCENT( forwardedpages ) > gcstablepercent)
                expandheap();
}

/* The following procedure is called on the completion of garbage collection
   before free memory is zeroed (memory is zeroed only when that special
   option is set).  It provides a handy place to put a breakpoint.
*/

static void  gcdone() {};


/* The following procedure is called to end garbage collection.  It
   resets various bookkeeping parameters, and determines whether
   total collection and heap expansion are necessary.
*/

static void  gcwrapup()
{
        forw_space = curr_space;  /* Register that GC is over */
        prev_space = curr_space;  /* Reclaim storage */

        /* Discard rest of current forward freepage */
        if  (forw_freewords != 0) {
           *forw_freep = MAKE_HEADER( forw_freewords, freespace_callback);
           forw_freewords = 0;
        }
        forw_freep = NULL;

        if (protectedpages != 0) abort();

        if  (gcflags & GCDEBUGLOG) verify_heap();

        /* Print memory use statistics if required */
        if  (gcflags & GCMEM)  memory_stats();

        /* Change headers on heap display */
        if  (gcflags & GCHEAPMAP) display_headers( "Application Allocation" );

        /* Reset bookkeeping parameters */
        allocatedpages = currentpages+forwardedpages;
        currentpages=0;
        gcdone();
```

```
        /* Zero free memory if required */
        if  (gcflags & GCZERO)  {
           int  page = firstheappage-1;
           while  (++page <= lastheappage)  {
              if  (space[ page ] != curr_space && UNSTABLE( page ))
                 bzero( (char*)PAGE_to_GCP( page ), PAGEBYTES );
           }
        }

        /* Check for generational collection and heap expansion */
        if  (gcallpercent) {
           /* Performing generational collection */
           if  (HEAPPERCENT(allocatedpages)>=gcallpercent) {
              /* Perform a total collection */
              makecurrentstableset();
              int save_gcallpercent = gcallpercent;
              gcallpercent = 100;
              gccollect();
              gcallpercent = save_gcallpercent;
              if  (shouldexpandheap()) expandheap();
           }
        }
        else  {
           /* Not performing generational collection */
           if  (shouldexpandheap())  expandheap();
           makecurrentstableset();
        }
}

/* The following function tests whether a set of contiguous heap pages are
   all unprotected
*/

static int  all_unprotected( int page, int num_pages )
{
        int begin_physpage = PAGE_to_PPADDR( page ),
            end_physpage   = PAGE_to_PPADDR( page+num_pages-1 );
        do
            if  (is_physpage_protected( begin_physpage )) return 0;
        while ( (begin_physpage += PHYS_PAGEBYTES) <= end_physpage );
        return 1;
}

/* The following predicate returns 1 if the specified set of heap pages is
   not "linked" to other pages to form a physical page cluster that's larger
   than the minimum necessary cluster size.
*/

static inline int  SMALL_CLUSTER( int firstpage, int lastpage ) {
        return ( same_physpage(PAGE_to_ADDR(firstpage),
                               PAGE_to_ADDR(lastpage)) ||
                (type[ firstpage - firstpage%phys_cap ] == OBJECT &&
                 type[ lastpage - lastpage%phys_cap + phys_cap ] == OBJECT) );
}

/* When gcalloc is unable to allocate storage, it calls this routine to
   allocate one or more pages.  If space is not available then the garbage
   collector is called and/or the heap is expanded.
*/
```

```
static void  allocatepage( int pages )
{
        int  free,              /* # contiguous free pages */
             firstpage,         /* Page # of first free page (unprotected) */
             first_prot = -1,   /* Page # of first free page (protected) */
             allpages;          /* # of pages in the heap */

        if  (pages > MAX_HEADER_PAGES)  {
           fprintf( stderr,
            "\n***** gcalloc  Unable to allocate objects larger than %d bytes\n",
#ifdef DOUBLE_ALIGN
                    MAX_HEADER_PAGES*PAGEBYTES-8 );
#else
                    MAX_HEADER_PAGES*PAGEBYTES-4 );
#endif
           abort();
        }

        /* Scan a few pages iff incremental collection is going on */
        if  (curr_space != forw_space)  {
           scan_few_physpages( 1 );
           /* Check if collection is over */
           if  ( protectedpages == 0 )
              gcwrapup();
        }

        if  (curr_space == forw_space)  {
           /* Start incremental collection if more than 1/N of the
              space will be allocated, such that
                 N = 3 if doing incremental collection,
                 N = 2 otherwise.
           */
           if  (!(gcflags & GCNOINC)) {
                    /* incremental collection */
              if  (allocatedpages+pages >= heappages/3)
                 gccollect();
           } else {
              if  (allocatedpages+pages >= heappages/2)
                 gccollect();
           }
        }

        /* Try to allocate space */
        free = 0;
        allpages = heapspanpages;
        while  (allpages--)  {
           if  (space[ curr_freepage ] < prev_space  &&
                UNSTABLE( curr_freepage )) {
              /* make sure that curr_freepage is a "cluster" by itself */
              if  (free++ == 0)  {   /* Potential first page */
                 if  ( SMALL_CLUSTER(curr_freepage, curr_freepage+pages-1) )
                    firstpage = curr_freepage;
                 else {
                    curr_freepage =
                        next_page(curr_freepage - curr_freepage%phys_cap
                                  + phys_cap-1 );
                    free = 0;
                    continue;  // loop "while (allpages--)" again
                 }
              }
```

```
            if  (free == pages) {  /* Page(s) found */
               curr_freep = PAGE_to_GCP( firstpage );
               curr_freewords = pages*PAGEWORDS;
               allocatedpages = allocatedpages+pages;
               currentpages   = currentpages+pages;
               curr_freepage  = next_page( firstpage+pages-1 );
               space[ firstpage ] = curr_space;
               type[ firstpage ]  = OBJECT;
               bzero( (char*)&firstword[ firstpage*BIT_WORDS ],
                      BIT_BYTES*pages );
               for  (int i = 1; i < pages; i++) {
                   space[ firstpage+i ] = curr_space;
                   type[ firstpage+i ]  = CONTINUED;
               }
               if  (gcflags & GCHEAPMAP)
                  for  (i = 0; i < pages; i++)
                      page_map( firstpage+i );
               if  (!all_unprotected(firstpage, pages))
                  scan_physpage_cluster( PAGE_to_PPADDR(firstpage) );
               return;
            }
            else {  /*  free < pages  */
               curr_freepage = next_page( curr_freepage );
               if  (curr_freepage == firstheappage)  free = 0;
            }
         }
         else {  /* curr_freepage is not free */
            free = 0;
            curr_freepage = next_page( curr_freepage );
         }
      }

      /* Failed to allocate space, keep trying iff heap can expand.  Assure
         that minimum increment size is at least the size of this object.
      */
      if  (gcincbytes/PAGEBYTES < pages) gcincbytes = pages*PAGEBYTES;
      if  (expandheap())  {
         allocatepage( pages );
         return;
      }

      /* Can't do it */
      fprintf( stderr,
          "\n***** gcalloc  Unable to allocate %d bytes in a %d byte heap\n",
              pages*PAGEBYTES, heappages*PAGEBYTES );
      abort();
}

/* Storage is allocated by the following function.  It returns a pointer
   to the object.  It is up to the specific constructor procedure to assure
   that all pointer slots are correctly initialized.  The word count includes
   one word for the header.
*/

GCP  gcalloc( int words, int callback )
{
      GCP  object;    /* Pointer to the object */

      /* Try to allocate from current page */
      if  (words <= curr_freewords)  {
```

```
            *curr_freep = MAKE_HEADER( words, callback );
            object = curr_freep+1;
            curr_freewords = curr_freewords-words;
            curr_freep = curr_freep+words;
#ifdef DOUBLE_ALIGN
            if  ((curr_freewords & 1) == 0  &&  curr_freewords)  {
              *curr_freep = doublepad;
              curr_freewords = curr_freewords-1;
              curr_freep = curr_freep+1;
            }
#endif
            SET_FIRSTWORD( object );
            return( object );
         }
         /* Discard any remaining portion of current page */
         if  (curr_freewords != 0)  {
            *curr_freep = MAKE_HEADER( curr_freewords, freespace_callback );
            curr_freewords = 0;
         }
         /* Object is less than 1 page in size */
         if  (words < ONEPAGEOBJ_WORDS)  {
            allocatepage( 1 );
#ifdef DOUBLE_ALIGN
            *curr_freep = doublepad;
            curr_freewords = curr_freewords-1;
            curr_freep = curr_freep+1;
#endif
            *curr_freep = MAKE_HEADER( words, callback );
            object = curr_freep+1;
            curr_freewords = curr_freewords-words;
            curr_freep = curr_freep+words;
#ifdef DOUBLE_ALIGN
            if  ((curr_freewords & 1) == 0)  {
               *curr_freep = doublepad;
               curr_freewords = curr_freewords-1;
               curr_freep = curr_freep+1;
            }
#endif
            SET_FIRSTWORD( object );
            return( object );
         }
         /* Object >= 1 page in size */

#ifdef DOUBLE_ALIGN
         allocatepage( (words+PAGEWORDS)/PAGEWORDS );
         *curr_freep = doublepad;
         curr_freewords = curr_freewords-1;
         curr_freep = curr_freep+1;
#else
         allocatepage( (words+PAGEWORDS-1)/PAGEWORDS );
#endif

         *curr_freep = MAKE_HEADER( words, callback );
         object = curr_freep+1;
         curr_freewords = 0;
         curr_freep = NULL;
         SET_FIRSTWORD( object );
         return( object );
}
```

```
/* The following predicate returns 1 if the object is checked by the garbage
   collector, otherwise 0.
*/

#ifdef COMPILER_VERSION_1_2
int  gcobject( void* obj )
{
        if  (obj >= STATIC_1  &&  obj < &obj)  {
           int  page = GCP_to_PAGE( obj );
           if  (page < firstheappage  ||  page > lastheappage  ||
                space[ page ] == UNALLOCATEDPAGE)
              return 0;
        }
        return  1;
}
#endif


/* The delete operator is redefined so that the user program may call
   delete on garbage collected objects.
*/

#ifdef COMPILER_VERSION_1_2
extern void operator  delete( void* p )  {
        if  (gcheapcreated)  {
           int  page = GCP_to_PAGE( p );
           if  (page >= firstheappage  &&  page <= lastheappage  &&
                space[ page ] != UNALLOCATEDPAGE)
              return;
        }
        if  (p)  free( (char*)p );
}
#endif
```

References

[Appel 89] Andrew W. Appel.
Simple Generational Collection and Fast Allocation.
Software-Practice and Experience , February, 1989.

[Appel et al. 88] Andrew W. Appel, John R. Ellis, and Kai Li.
Real-Time Concurrent Collection on Stock Multiprocessors.
ACM SIGPLAN Notices , June, 1988.

[Bartlett 88] Joel F. Bartlett.
Compacting Garbage Collection with Ambiguous Roots.
Technical Report 88/2, DEC Western Research Laboratory, February, 1988.

[Bartlett 89] Joel F. Bartlett.
Mostly-Copying Collection Picks Up Generations and C++.
Technical Report TN-12, DEC Western Research Laboratory, October, 1989.

[Detlefs 90] David Detlefs.
Concurrent Garbage Collection for C++.
Technical Report CMU-CS-90-119, Carnegie Mellon University, May, 1990.

[Fenichel&Yochelson 69]
Robert R. Fenichel, and Jerome C. Yochelson.
A LISP garbage-collector for virtual-memory.
Communciations of the ACM 12, (11), 611-612 , 1969.

[Li 90] Kai Li.
Real-time Concurrent Collection in User Mode.
Technical Report CS-TR-291-90, Princeton University, October, 1990.
In Workshop on Garbage Collection in Objected Oriented Systems, October 1990, Ottawa.

[Moon 84] David Moon.
Garbage Collection in a large LISP system.
ACM , 1984.
In ACM Symposium on LISP and Functional Programming.

[Shaw 88] Robert Shaw.
Empirical Analysis of a Lisp System.
Technical Report CSL-TR-88-351, Stanford University, February, 1988.

[Stroustrup 86] Bjarne Stroustrup.
The C++ Programming Language.
Addison-Wesley, 1986.

[Thacker&Stewart 87]
Charles P. Thacker, and Lawrence C. Stewart.
Firefly: A Multiprocessor Workstation.
ACM , 1987.
In Proceedings of the Second International Conference on Achitectural Support for Programming Languages and Operating Systems.

[Weiser 89] Mark Weiser.
The Portable Common Runtime Approach to Interoperability.
ACM SIGPLAN Notices , December, 1989.

WRL Research Reports

‘‘Titan System Manual.’’
Michael J. K. Nielsen.
WRL Research Report 86/1, September 1986.

‘‘Global Register Allocation at Link Time.’’
David W. Wall.
WRL Research Report 86/3, October 1986.

‘‘Optimal Finned Heat Sinks.’’
William R. Hamburgen.
WRL Research Report 86/4, October 1986.

‘‘The Mahler Experience: Using an Intermediate Language as the Machine Description.’’
David W. Wall and Michael L. Powell.
WRL Research Report 87/1, August 1987.

‘‘The Packet Filter: An Efficient Mechanism for User-level Network Code.’’
Jeffrey C. Mogul, Richard F. Rashid, Michael J. Accetta.
WRL Research Report 87/2, November 1987.

‘‘Fragmentation Considered Harmful.’’
Christopher A. Kent, Jeffrey C. Mogul.
WRL Research Report 87/3, December 1987.

‘‘Cache Coherence in Distributed Systems.’’
Christopher A. Kent.
WRL Research Report 87/4, December 1987.

‘‘Register Windows vs. Register Allocation.’’
David W. Wall.
WRL Research Report 87/5, December 1987.

‘‘Editing Graphical Objects Using Procedural Representations.’’
Paul J. Asente.
WRL Research Report 87/6, November 1987.

‘‘The USENET Cookbook: an Experiment in Electronic Publication.’’
Brian K. Reid.
WRL Research Report 87/7, December 1987.

‘‘MultiTitan: Four Architecture Papers.’’
Norman P. Jouppi, Jeremy Dion, David Boggs, Michael J. K. Nielsen.
WRL Research Report 87/8, April 1988.

‘‘Fast Printed Circuit Board Routing.’’
Jeremy Dion.
WRL Research Report 88/1, March 1988.

‘‘Compacting Garbage Collection with Ambiguous Roots.’’
Joel F. Bartlett.
WRL Research Report 88/2, February 1988.

‘‘The Experimental Literature of The Internet: An Annotated Bibliography.’’
Jeffrey C. Mogul.
WRL Research Report 88/3, August 1988.

‘‘Measured Capacity of an Ethernet: Myths and Reality.’’
David R. Boggs, Jeffrey C. Mogul, Christopher A. Kent.
WRL Research Report 88/4, September 1988.

‘‘Visa Protocols for Controlling Inter-Organizational Datagram Flow: Extended Description.’’
Deborah Estrin, Jeffrey C. Mogul, Gene Tsudik, Kamaljit Anand.
WRL Research Report 88/5, December 1988.

‘‘SCHEME->C A Portable Scheme-to-C Compiler.’’
Joel F. Bartlett.
WRL Research Report 89/1, January 1989.

‘‘Optimal Group Distribution in Carry-Skip Adders.’’
Silvio Turrini.
WRL Research Report 89/2, February 1989.

‘‘Precise Robotic Paste Dot Dispensing.’’
William R. Hamburgen.
WRL Research Report 89/3, February 1989.

''Simple and Flexible Datagram Access Controls for Unix-based Gateways.''
Jeffrey C. Mogul.
WRL Research Report 89/4, March 1989.

''Spritely NFS: Implementation and Performance of Cache-Consistency Protocols.''
V. Srinivasan and Jeffrey C. Mogul.
WRL Research Report 89/5, May 1989.

''Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines.''
Norman P. Jouppi and David W. Wall.
WRL Research Report 89/7, July 1989.

''A Unified Vector/Scalar Floating-Point Architecture.''
Norman P. Jouppi, Jonathan Bertoni, and David W. Wall.
WRL Research Report 89/8, July 1989.

''Architectural and Organizational Tradeoffs in the Design of the MultiTitan CPU.''
Norman P. Jouppi.
WRL Research Report 89/9, July 1989.

''Integration and Packaging Plateaus of Processor Performance.''
Norman P. Jouppi.
WRL Research Report 89/10, July 1989.

''A 20-MIPS Sustained 32-bit CMOS Microprocessor with High Ratio of Sustained to Peak Performance.''
Norman P. Jouppi and Jeffrey Y. F. Tang.
WRL Research Report 89/11, July 1989.

''The Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance.''
Norman P. Jouppi.
WRL Research Report 89/13, July 1989.

''Long Address Traces from RISC Machines: Generation and Analysis.''
Anita Borg, R.E.Kessler, Georgia Lazana, and David W. Wall.
WRL Research Report 89/14, September 1989.

''Link-Time Code Modification.''
David W. Wall.
WRL Research Report 89/17, September 1989.

''Noise Issues in the ECL Circuit Family.''
Jeffrey Y.F. Tang and J. Leon Yang.
WRL Research Report 90/1, January 1990.

''Efficient Generation of Test Patterns Using Boolean Satisfiablilty.''
Tracy Larrabee.
WRL Research Report 90/2, February 1990.

''Two Papers on Test Pattern Generation.''
Tracy Larrabee.
WRL Research Report 90/3, March 1990.

''Virtual Memory vs. The File System.''
Michael N. Nelson.
WRL Research Report 90/4, March 1990.

''Efficient Use of Workstations for Passive Monitoring of Local Area Networks.''
Jeffrey C. Mogul.
WRL Research Report 90/5, July 1990.

''A One-Dimensional Thermal Model for the VAX 9000 Multi Chip Units.''
John S. Fitch.
WRL Research Report 90/6, July 1990.

''1990 DECWRL/Livermore Magic Release.''
Robert N. Mayo, Michael H. Arnold, Walter S. Scott, Don Stark, Gordon T. Hamachi.
WRL Research Report 90/7, September 1990.

''Pool Boiling Enhancement Techniques for Water at Low Pressure.''
Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.
WRL Research Report 90/9, December 1990.

''Writing Fast X Servers for Dumb Color Frame Buffers.''
Joel McCormack.
WRL Research Report 91/1, February 1991.

‘‘Analysis of Power Supply Networks in VLSI Circuits.’’
Don Stark.
WRL Research Report 91/3, April 1991.

‘‘Procedure Merging with Instruction Caches.’’
Scott McFarling.
WRL Research Report 91/5, March 1991.

‘‘Don’t Fidget with Widgets, Draw!.’’
Joel Bartlett.
WRL Research Report 91/6, May 1991.

‘‘Pool Boiling on Small Heat Dissipating Elements in Water at Subatmospheric Pressure.’’
Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.
WRL Research Report 91/7, June 1991.

‘‘Incremental, Generational Mostly-Copying Garbage Collection in Uncooperative Environments.’’
G. May Yip.
WRL Research Report 91/8, June 1991.

WRL Technical Notes

‘‘TCP/IP PrintServer: Print Server Protocol.’’
Brian K. Reid and Christopher A. Kent.
WRL Technical Note TN-4, September 1988.

‘‘TCP/IP PrintServer: Server Architecture and Implementation.’’
Christopher A. Kent.
WRL Technical Note TN-7, November 1988.

‘‘Smart Code, Stupid Memory: A Fast X Server for a Dumb Color Frame Buffer.’’
Joel McCormack.
WRL Technical Note TN-9, September 1989.

‘‘Why Aren’t Operating Systems Getting Faster As Fast As Hardware?’’
John Ousterhout.
WRL Technical Note TN-11, October 1989.

‘‘Mostly-Copying Garbage Collection Picks Up Generations and C++.’’
Joel F. Bartlett.
WRL Technical Note TN-12, October 1989.

‘‘Limits of Instruction-Level Parallelism.’’
David W. Wall.
WRL Technical Note TN-15, December 1990.

‘‘The Effect of Context Switches on Cache Performance.’’
Jeffrey C. Mogul and Anita Borg.
WRL Technical Note TN-16, December 1990.

‘‘MTOOL: A Method For Detecting Memory Bottlenecks.’’
Aaron Goldberg and John Hennessy.
WRL Technical Note TN-17, December 1990.

‘‘Predicting Program Behavior Using Real or Estimated Profiles.’’
David W. Wall.
WRL Technical Note TN-18, December 1990.
‘‘Systems for Late Code Modification.’’
David W. Wall.
WRL Technical Note TN-19, June 1991.