SPAMM

The Simple, Poor-man's Allocator/Memory-Manager
Garbage collection extensions for C
Manual version $Revision: 2.7 $
$Date: 1994/11/29 01:59:54 $

Bob Glickstein

Introduction

This document describes the application programmer's interface to SPAMM, a library package for adding automatic garbage collection to C programs.

1 Garbage collection overview

SPAMM is a simple yet powerful package for adding garbage-collection to C programs. Garbage collection frees the programmer from concern about memory management. The programmer simply requests memory when it's needed, then forgets about it. The allocated memory is automatically reclaimed when it's no longer needed.

SPAMM uses a straightforward mark-and-sweep garbage collection algorithm. In this scheme, a garbage collection consists of two phases. In the *mark phase*, a special list of pointers called the *root* is traversed. The referents of these *rooted pointers* are marked as *reachable*. After the mark phase, the *sweep phase* takes place, during which all objects *not* marked reachable are reclaimed (for later reuse). It is the programmer's responsibility to ensure that any pointers to SPAMM-managed objects are properly rooted (that is, that they appear in the root list).

The application programmer asks SPAMM to manage the allocation and reclamation for particular data types. For each different type that is to be managed by SPAMM, the programmer creates a SPAMM *pool*. New objects of a particular type are allocated from the corresponding pool. Managed objects are automatically reclaimed when they are no longer needed; i.e., when there no longer exist any pointers to them.

A garbage collection is invoked automatically when the application requests a new object but the corresponding pool is full. If the garbage collection does not result in freeing up some items in the pool, then the pool size is increased. A garbage collection can be explicitly invoked by the application, or it can be suppressed for the duration of some critical computation.

Each data type has an associated *trace function* and a *reclaim function*. The reclaim function is invoked on an object when a garbage collection (in the sweep phase) is about to reclaim it. The trace function's purpose is to allow the mark phase to find and traverse the pointers inside a SPAMM-managed data object. See Chapter 6 [Rooting and tracing], page 9.

2 Mechanisms

This chapter discusses the technology behind SPAMM, and is intended to motivate and clarify the sections that follow. If you are not interested in the technical details of SPAMM, feel free to skip this chapter.

2.1 Definitions

SPAMM internally maintains two data structures, the *root list* and the *pool list*.

The root list is a list of addresses of pointers to SPAMM-managed objects. In other words, if you've asked SPAMM to manage objects of type `struct` *`foo`*, then the root list will contain values of type `struct` *`foo`* `**`. This is because SPAMM needs to know the *address* of any variable that can point to a `struct` *`foo`*, as explained below.

The pool list is a list of object pools, one pool for each data type you've asked SPAMM to manage. Each pool, in turn, consists of three *page lists*, designated the *empty page list*, the *full page list*, and the *neither page list*. A page list is a list of *pages*.

A page contains a fixed number of allocatable objects; this fixed number is chosen per data type when the type's pool is created. When the application requests a new instance of some object, SPAMM hands back a pointer to one of the objects in one of the already-allocated pages of objects. That object is then flagged (within the page) as *in use*. An *empty page* is one in which none of the constituent objects is in use. A *full page* is one in which all of the constituent objects are in use. A *neither page* is one which is neither empty nor full (i.e., some but not all of its constituent objects are in use).

2.2 Allocation

When the application requests a new object from a given object pool, SPAMM examines the pool to see whether it contains any neither pages. If so, an object is allocated from the first neither page and returned. (If that was the last unused object in the neither page, then the page becomes a full page and is moved from the neither page list to the full page list.)

If the pool contains no neither pages, then the empty pages are consulted. If there is an empty page, an object is allocated from it and returned, and the page is moved to the neither page list (or the full page list if it is a one-element page).

If the pool contains neither neither pages nor empty pages, then a garbage collection is triggered to try and create some empty pages or neither pages. See Section 2.3 [Garbage collection], page 4.

If the garbage collection fails to turn full pages into empty or neither pages, then SPAMM relents and actually allocates a new empty page.

If the application has turned off garbage collection (see Chapter 7 [Controlling SPAMM], page 13), then this sequence happens slightly differently. If no empty or neither page is found, then SPAMM tries to allocate a new page right away. If *that* fails, SPAMM will try a garbage collection (regardless of the request to suspend garbage collection). If *even that* fails to produce new allocatable objects, then a new-page allocation is tried one more time before giving up.

2.3 Garbage collection

In a garbage collection, all objects in all pages in all pools start out marked as *unreachable*. The mark phase then begins and the root list is traversed. Recall that the root list contains pointers to pointers to objects. Each pointer pointed to in the root list is then *traced*.

Tracing a pointer works as follows: If the pointer is null, nothing happens. Otherwise, the pointer is presumed to point to some SPAMM-managed object. That object's page is found (by simple pointer arithmetic). If the object is marked *reachable*, then nothing else is done. Otherwise, the object is marked as reachable, and then it is recursively traced using the pool's trace function. (The pool's trace function is responsible for calling `spamm_Trace` on the pointers inside a SPAMM-managed object.)

After the pointers in the root list are traced, the sweep phase begins, and each in-use object is then processed (by traversing the pool list [and the neither page list and the full page list within each pool]) as follows: The object is tested to see whether it was marked by the mark phase. If so, it is still in use; otherwise, it is garbage. The in-use flag is cleared for that object, reclaiming it for future re-use. The page containing the reclaimed object is moved, if appropriate, from the full page list to the neither page list, or from the neither page list to the empty page list. (The neither page is skipped if the pool's pages are only one element large.)

After the sweep phase, the garbage collection concludes by freeing excess empty pages. Empty pages are considered excessive if the number of in-use objects in a pool is smaller than 25% of the total number of objects in that pool.

`(void (*) ()) spamm_GcStart` [Variable]

> This is a pointer to a void function of no arguments; if you set it to some function, that function will be called when a garbage collection begins.

`(void (*) ()) spamm_GcEnd` [Variable]

> This can point to a function to be called when garbage collection ends. Usually this and `spamm_GcStart` are used to invoke functions that advise the user that a garbage collection is underway.

3 Initializing

Before performing any other SPAMM operations, SPAMM must be initialized.

`void spamm_Initialize ()` [Function]

Initializes SPAMM. Must be called exactly once before using SPAMM functionality.

4 Creating a pool

A SPAMM object pool is of type `struct spamm_ObjectPool`. Create a pool by whatever means are appropriate, then initialize it with `spamm_InitPool`. A separate pool should be created for each data type which you want SPAMM to manage. The type of data object managed by one pool is referred to as the *pool type*.

The `elts` parameter specifies how many objects will be allocated at a time in a page. For large data types which you know will be instantiated infrequently, it makes sense to choose a small value of `elts` to reduce the potential for allocating latent (unused) space. For smaller data types, or ones which are frequently allocated, choose the maximum value for `elts`. It sometimes happens that your application uses a single instance of some data type which, for bookkeeping reasons, needs to be SPAMM-managed[1]. In this case, use a value of 1 for `elts` in that pool.

`void spamm_InitPool` (*struct spamm_ObjectPool* *`*pool`*, *int* `size`, *int* `elts`, *void* (*`*tracefn`)(), *void* (*`*reclaimfn`)()) [Function]

Initialize a pool of objects of a particular type. Arguments are: *pool*, a pointer to an uninitialized object pool structure; *size*, the size (in bytes) of an instance of the desired pool type; *elts*, the number of elements allocated at a time in a page of this pool; *tracefn*, a function responsible for tracing an object of the pool's type; and *reclaimfn*, a function responsible for finalizing a data structure before it becomes inaccessible.

The value `ULBITS` is the number of bits in an `unsigned long` on your architecture. The value of *elts* must lie between 1 and `ULBITS`.

The *tracefn* takes one argument, a pointer to an object of the pool's type. This function will be called by the mark phase of the garbage collection. Chapter 6 [Rooting and tracing], page 9, for more information.

The *reclaimfn* takes one argument, a pointer to an object of the pool's type. This function should free privately-allocated memory within the object and perform other cleanup tasks. This function will be called by the sweep phase of a garbage collection on a reclaimed object. Note that it is not in general possible to know when this will occur, or whether it will occur at all.

This function can raise the `strerror[ENOMEM]` exception.

Suppose you have a data type defined as follows:

```
struct foo {
    char          *name;
    struct bar    *b;
    struct xyzzy *x;
    int            i;
};
```

You wish SPAMM to manage allocation and reclamation of instances of `struct foo`, so you create a pool to hold `struct foo` objects:

---

[1] This will be the case, for instance, if the data type in question can be pointed to by a pointer which might also point to another SPAMM-managed object.

```
...
struct spamm_ObjectPool fooPool;

spamm_InitPool(&fooPool, sizeof (struct foo), ULBITS,
               fooTrace, fooReclaim);
...
```

Suppose further that the `b` and `x` fields of a `struct foo` are pointers to other objects that are managed by SPAMM. Then `fooTrace` could be defined like this:

```
static void
fooTrace(f)
    struct foo *f;
{
    spamm_Trace(f->b);
    spamm_Trace(f->x);
}
```

This allows SPAMM, in the mark phase of a garbage collection, to follow the pointers inside a `struct foo` and recursively mark the found objects.

Now suppose that the `name` field of a `struct foo` is a privately-allocated string. Then `fooReclaim` probably needs to look like this:

```
static void
fooReclaim(f)
    struct foo *f;
{
    if (f->name)
        free(f->name);
}
```

`int spamm_PoolStats` (*struct spamm_ObjectPool *`op`*, *int *`empty`*, *int *`neither`*, *int *`full`*) [Function]

Reports the number of pages allocated in *op*. The `int` pointed to *empty* is set to the number of empty pages; *neither* will point to the number of neither pages; and *full* will point to the number of full pages. The return value is the sum of all pages allocated in *op*.

Any of *empty*, *neither*, or *full* may be `NULL`, but this does not affect the return value.

5 Allocating an object

Once you have created an object pool and initialized it with `spamm_InitPool`, you can request objects of the pool's type using `spamm_Allocate`.

`void * spamm_Allocate` (*struct spamm_ObjectPool *op*) [Function]

Return a new object from *op*, an object pool initialized by `spamm_InitPool`.

This function may raise the `strerror[ENOMEM]` exception.

6 Rooting and tracing

All pointers to SPAMM-managed objects must be rooted (inserted by reference into SPAMM's private root list), or must be accessible from rooted pointers. This allows the mark phase of a garbage collection to find objects that are being used and prevent them from being reclaimed in the sweep phase.

SPAMM's private root list is a doubly-linked list of pointers to pointers to objects. When the programmer creates a variable which can point to a SPAMM-managed object, it should be rooted immediately using one of the mechanisms described below. The variable should be unrooted before its lifetime expires (i.e., before the end of the scope in which it was defined).

Not all in-use objects are necessarily pointed to by pointers in the root list. Objects pointed to from the root list might themselves point to other SPAMM-managed objects which are not rooted. Such objects are said to be *accessible* from the rooted objects. Those objects themselves might point to others which are also considered accessible. The transitive closure of the objects accessible from the root list constitutes the set of objects-in-use. Other objects that have been allocated but which are not accessible are garbage which gets reclaimed during the sweep phase of a garbage collection.

Because SPAMM does not know the internal layout of a data object it is managing, it has no way of knowing which parts of the object might be pointing to other SPAMM-managed objects. It therefore relies on a programmer-defined trace function, one per data type being managed, to traverse the pointers within an object. The trace function is associated with a data type in a call to `spamm_InitPool` (see Chapter 4 [Creating a pool], page 6) and need only call the function `spamm_Trace` on each of the appropriate pointers contained within an object of the pool's type.

`void spamm_Trace` (*void* *`*ptr`*) [Function]

> Cause the appropriate trace function to be invoked on *ptr*, a pointer to a SPAMM-managed object. Typically called from the trace function of one pool on the components of its pool type which are other SPAMM-managed types. `spamm_Trace` ensures that the trace function is only called once for any particular object during a garbage collection.

6.1 Rooting mechanisms

SPAMM provides three functions to control rooting and unrooting, and it also provides a new syntactic construct. The syntactic construct is preferred over the lower-level functions to control rooting ad unrooting (except for pointers whose *extent* is not the same as their *scope*; see below). The syntactic construct consists of two macros, `SPAMM_ROOT` and `SPAMM_ENDROOT`, and is used like this:

```
void somefunction()
{
    struct foo *f = (struct foo *) 0;
    struct bar *b = (struct bar *) 0;

    SPAMM_ROOT((&f, &b, 0)) {
        ...Code that uses the variables f and b...
    } SPAMM_ENDROOT;
}
```

In this example, the pointers `f` and `b` are rooted on entry to the `SPAMM_ROOT` block, and are automatically unrooted on exit from that block. Note the following:

- The arguments to `SPAMM_ROOT` are enclosed in *double* parentheses.
- The arguments to `SPAMM_ROOT` are the *addresses* of pointers.
- The argument list is terminated with a zero.
- The pointers whose addresses are given to `SPAMM_ROOT` are initialized to zero.

This last point is worth some explanation. All pointers in SPAMM's root list are assumed either to be zero, or to point to a valid SPAMM-managed object. These pointers will never be examined except during the mark phase of a garbage collection. Because it is not in general possible to predict when a garbage collection will occur, it is safest to simply ensure that all rooted pointers are properly initialized *before* they become rooted.

Important warning: Beware of functions that return SPAMM-managed objects but which are called without rooting their values! In a construct such as:

```
a(b(), c());
```

`b` may execute and produce a result, but the result is left, *unrooted*, on the stack while `c` executes, which could result in the return value of `b` being garbage-collected before `a` gets control. This workaround will solve the problem:

```
x = NULL;
y = NULL;
SPAMM_ROOT((&x, &y, 0)) {
    a(x = b(), y = c());
} SPAMM_ENDROOT;
```

The rooted variables `x` and `y` will protect the return values of `b` and `c`.

The syntactic constructs `SPAMM_ROOT` and `SPAMM_ENDROOT` are only appropriate when the *extent* of the pointers being rooted equals their *scope*. The scope of a variable refers to the region of code in which it is legal to refer to the variable; thus, a variable local to a function may only be referred to within that function, and the function is said to be that variable's scope. The extent of a variable refers to the variable's lifetime, or the period during which the variable is valid. For a variable local to a function, the extent is the same as the scope. However, consider this:

```
int foo()
{
    static struct bar *b = 0;

    SPAMM_ROOT((&b, 0)) {
        ...
    } SPAMM_ENDROOT;
}
```

The extent of `b` persists beyond the scope of the function; that is, its value remains intact between invocations of `foo`. It is therefore wrong to unroot it at the end of `foo`'s scope. The low-level functions `spamm_Root` and `spamm_Unroot` should be used instead:

```
int foo()
{
    static struct bar *b = 0;
    static int rootindex;

    if (...first time through...) {
        rootindex = spamm_Root(&b);
    }
    ...
    if (...last time through...) {
        spamm_Unroot(rootindex);
    }
}
```

6.1.1 Low-level rooting functions

It is easier and less error-prone to use the `SPAMM_ROOT`/`SPAMM_ENDROOT` pair of macros to keep your pointers properly rooted. However, if you require finer control over the process, there are three functions that you can use.

`int spamm_Root` (*void **ptr*) [Function]

Inserts *ptr* (a pointer to a pointer to a SPAMM-managed object) into the root list and returns the new *root list index* used (see below).

`int spamm_RootList` (*void **p1, void **p2, . . ., 0*) [Function]

Each argument is a pointer to a pointer to a SPAMM-managed object. Calls `spamm_Root` on each of its arguments. A zero terminates the argument list. The return value is the number of pointers that were rooted.

`void spamm_Unroot` (*int n*) [Function]

Removes from the root list the entry whose index is *n* (which is an index such as the one returned by `spamm_Root`).

The root list is implemented as a doubly-linked list using the Dlist package (q.v.). Items in a dlist are referenced by their integer indices. The value returned by `spamm_Root` is the dlist index of the entry used for rooting its argument.

When a new pointer is rooted, it is always placed at the head of the dlist. The convenience macros `SPAMM_ROOT` and `SPAMM_ENDROOT` rely on this property for keeping track of which pointers need to be removed from the root list. `SPAMM_ROOT` records the position p in the root list of the latest element added (using `spamm_RootList`) as well as the number of elements n added. At the end of a `SPAMM_ROOT` block, `SPAMM_ENDROOT` finds location p in the root list and unroots n pointers from that spot. The correctness of this shortcut depends on that segment of the root list remaining intact through the `SPAMM_ROOT` block. In other words, if the programmer adds or removes pointers in the middle of a portion of the root list which `SPAMM_ROOT` created, then `SPAMM_ENDROOT` will wind up removing the wrong pointers from the root list. So don't do that.

7 Controlling SPAMM

SPAMM gives the programmer some control over how and when it operates. In particular, it is possible to request that a garbage collection take place immediately; it is also possible to suspend garbage collection for a period.

`void spamm_CollectGarbage ()` [Function]

Performs a garbage collection. The global variable `spamm_GcStart` is a pointer to a void function of no arguments; if you set it to some function, that function will be called when a garbage collection begins. Similarly, the variable `spamm_GcEnd` can point to a function to be called when garbage collection ends. Usually these function pointers are used to invoke functions that advise the user that a garbage collection is under way.

To suspend garbage collection for a period, use the `SPAMM_GCSUSPEND` and `SPAMM_ENDGCSUSPEND` pair of macros like so:

```
... code that might be interrupted by a garbage collection ...
SPAMM_GCSUSPEND {
    ... some critical code you don't want interrupted ...
} SPAMM_ENDGCSUSPEND;
... code that might be interrupted by a garbage collection ...
```

The macros `SPAMM_GCSUSPEND` and `SPAMM_ENDGCSUSPEND` are a convenient and less error-prone interface to the low-level functions that perform the actual suspension, namely `spamm_GcSuspend` and `spamm_GcUnsuspend`.

`void spamm_GcSuspend ()` [Function]

Suspend garbage collection until the next `spamm_GcUnsuspend`.

`void spamm_GcUnsuspend ()` [Function]

Remove the current garbage collection suspension.

Actually, `spamm_GcSuspend` and `spamm_GcUnsuspend` don't necessarily start and stop garbage collection suspension. They merely increment and decrement (respectively) an internal counter which starts at zero. Only while the counter *is* zero can a garbage collection occur. Thus it is possible to write code like this:

```
spamm_GcSuspend();
foo();
bar();
spamm_GcUnsuspend();
```

even if the definition of `foo` looks like this:

```
void
foo()
{
    spamm_GcSuspend();
    ... some code ...
    spamm_GcUnsuspend();
}
```

If garbage collection suspension did not use the counter scheme, then the call to `foo` would re-enable garbage collection when it called `spamm_GcUnsuspend`, unbeknownst to the caller, who thinks it's still suspended when it calls `bar` after `foo`.

In fact, garbage collection suspension is even less straightforward than that. It is still possible for a garbage collection to occur even if a suspension is pending. However, as described in Section 2.2 [Allocation], page 3, this will only happen as a last resort if the operating system cannot deliver more free memory when required.

8 SPAMM and exceptions

SPAMM makes use of the Except package (q.v.). Memory requested from the operating system is obtained via `emalloc`, and so can raise the `strerror[ENOMEM]` exception. Also, the `SPAMM_ROOT`, `SPAMM_ENDROOT`, `SPAMM_GCSUSPEND`, and `SPAMM_ENDGCSUSPEND` are defined in terms of Except primitives. A block of code written as

```
SPAMM_ROOT((&a, &b, &c, 0)) {
    ... your code ...
} SPAMM_ENDROOT;
```

expands as

```
do {
    int _Spamm_Root_Count_ = spamm_Rootlist(&a, &b, &c, 0);
    int _Spamm_Root_First_Index = dlist_Head(&spamm_ObjectRoot);

    TRY {
        ... your code ...
    } FINALLY {
        int i, j, index = _Spamm_Root_First_Index_;

        for (i = 0; i < _Spamm_Root_Count_; ++i) {
            j = index;
            index = dlist_Next(&spamm_ObjectRoot, j);
            spamm_Unroot(j);
        }
    } ENDTRY;
} while (0);
```

(Surrounding the expansion in a `do ... while (0)` is an idiomatic programming construct for encapsulating multi-statement macros.) Placing the unrooting portion of the expansion inside of a `FINALLY` block allows the pointers to be properly unrooted even if your code raises an exception that throws control outside of the `SPAMM_ROOT` block.

Similarly, a block of code written as

```
SPAMM_GCSUSPEND {
    ... your code ...
} SPAMM_ENDGCSUSPEND;
```

expands as

```
do {
    spamm_GcSuspend();
    TRY {
        ...your code...
    } FINALLY {
        spamm_GcUnsuspend();
    } ENDTRY;
} while (0);
```

The caveats and constraints that apply to programming with the Except package also apply to SPAMM. In particular, since a `SPAMM_ROOT` or a `SPAMM_GCSUSPEND` block is also a

`TRY` block, you must use Except's special mechanisms for performing non-local exits from those blocks (see Section "Non-local exits" in *Except*).

Index

Table of Contents