Garbage Collection

Garbage collection, reference counting, explicit allocation

As with all modern languages, OCaml provides a garbage collector so that you don't need to explicitly allocate and free memory as in C/C++.

As JWZ writes in his "Java sucks" rant [/web/20101122082924/http://www.jwz.org/doc/java.html]:

First the good stuff: Java doesn't have `free()`. I have to admit right off that, after that, all else is gravy. That one point makes me able to forgive just about anything else, no matter howegregious. Given this one point, everything else in this document fades nearly to insignificance. ***But...***

The OCaml garbage collector is a modern hybrid generational/incremental collector which outperforms hand-allocation in most cases. Unlike the Java GC, which gives GCs a bad name, the OCaml GC doesn't allocate huge amounts of memory at start-up, nor does it appear to have arbitrary fixed limits that need to be overridden by hand.

Why would garbage collection be faster than explicit memory allocation as in C? It's often assumed that calling `free` costs nothing. In fact `free` is an expensive operation which involves navigating over the complex data structures used by the memory allocator. If your program calls `free` intermittently, then all of that code and data needs to be loaded into the cache, displacing your program code and data, each time you `free` a single memory allocation. A collection strategy which frees multiple memory areas in one go (such as either a pool allocator or a GC) pays this penalty only once for multiple allocations (thus the cost per allocation is much reduced).

GCs also move memory areas around and compact the heap. This makes allocation easier, hence faster, and a smart GC can be written to interact well with the L1 and L2 caches.

Of course none of this precludes writing a very fast hand-allocator, but it's considerably harder work than most programmers realise.

OCaml's garbage collector has two heaps, the **minor heap** and the **major heap**. This recognises a general principle: Most objects are small and allocated frequently and then immediately freed. These objects go into the minor heap first, which is GCed frequently. Only some objects are long lasting. These objects get promoted from the minor heap to the major heap after some time, and the major heap is only collected infrequently.

The OCaml GC is synchronous. It doesn't run in a separate thread, and it can only get called during an allocation request.

GC vs. reference counting

Perl has a form of garbage collection, but it uses a simple scheme called **reference counting**. Simply put, each Perl object keeps a count of the number of other objects pointing (referencing) itself. When the count falls to zero, nothing is pointing at this object, and so the object can be freed.

Reference counting is not considered as serious garbage collection by computer scientists, yet it has one big practical advantage over full garbage collectors. With reference counting, you can avoid many explicit calls to `close`/`closedir` in code. For example:

```
foreach (@files)
{
  my $io = new IO::File "< $_" or die;
  # read from $io
}
```

This Perl code iterates over a list of `@files`, opening and reading each one. There is no need to close the `$io` file handle at the end of the loop. Because Perl uses reference counting, as soon as we reach the end of the loop, the `$io` variable goes out of scope, so the reference count on the file object goes to zero, and it is immediately freed (ie. closed).

Consider the equivalent OCaml code:

```
let read_file filename =
  let chan = open_in filename in
  (* read from chan *)
in
List.iter read_file files
```

Calls to `read_file` open the file but don't close it. Because OCaml uses a full garbage collector `chan` isn't collected until some time later when the minor heap becomes full. In addition, **OCaml will not close the channel when it collects the handle's memory**. So this program would eventually run out of file descriptors.

You need to be aware of this when writing OCaml code which uses files or directories or any other heavyweight object with complex finalisation.

To be fair to full garbage collection, I should mention the disadvantages of reference counting schemes:

- Each object needs to store a reference count. In other words there's a word overhead for every object. Programs use more memory, and are consequently slower because they are more likely to fill up the cache or spill into swap.
- Reference counting is expensive - every time you manipulate pointers to an object you need to update and check the reference count. Pointer manipulation is frequent, so this slows your program and bloats the code size of compiled code.
- They cannot collect so-called circular, or self-referential structures. I've programmed in many languages in many years and can't recall ever having created one of these.
- Graph algorithms, of course, violate the previous assumption. Don't try to implement the TSP in Perl!

The Gc module

The `Gc` module contains some useful functions for querying and calling the garbage collector from OCaml programs.

Here is a program which runs and then prints out GC statistics just before quitting:

```
Random.self_init ();;
Graphics.open_graph " 640x480";;

let rec iterate r x_init i =
        if i == 1 then x_init
        else
                let x = iterate r x_init (i-1) in
                r *. x *. (1.0 -. x);;

for x = 0 to 640 do
        let r = 4.0 *. (float_of_int x) /. 640.0 in
        for i = 0 to 39 do
                let x_init = Random.float 1.0 in
                let x_final = iterate r x_init 500 in
                let y = int_of_float (x_final *. 480.) in
                Graphics.plot x y
        done
done;;

read_line ();;

Gc.print_stat stdout;;
```

Here is what it printed out for me:

```
minor_words: 115926165          # Total number of words allocated
promoted_words: 31217           # Promoted from minor -> major
```

```
major_words: 31902              # Large objects allocated in major directly
minor_collections: 3538         # Number of minor heap collections
major_collections: 39           # Number of major heap collections
heap_words: 63488               # Size of the heap, in words = approx. 256K
heap_chunks: 1
top_heap_words: 63488
live_words: 2694
live_blocks: 733
free_words: 60794
free_blocks: 4
largest_free: 31586
fragments: 0
compactions: 0
```

We can see that minor heap collections are approximately 100 times more frequent than major heap collections (in this example, not necessarily in general). Over the lifetime of the program, an astonishing 440 MB of memory was allocated, although of course most of that would have been immediately freed in a minor collection. Only about 128K was promoted to long-term storage on the major heap, and about another 128K consisted of large objects which would have been allocated directly onto the major heap.

We can instruct the GC to print out debugging messages when one of several events happen (eg. on every major collection). Try adding the following code to the example above near the beginning:

```
Gc.set { (Gc.get ()) with Gc.verbose = 0x01 };;
```

(We haven't seen the `{ expression with field = value }` form before, but it should be mostly obvious what it does). The above code anyway causes the GC to print a message at the start of every major collection.

Finalisation and the Weak module

We can write a function called a **finaliser** which is called when an object is about to be freed by the GC.

The `Weak` module lets us create so-called weak pointers. A **weak pointer** is best defined by comparing it to a "normal pointer". When we have an ordinary OCaml object, we reference that object through a name (eg. `let name = ... in`) or through another object. The garbage collector sees that we have a reference to that object and won't collect it. That's what you might call a "normal pointer". If, however, you hold a weak pointer or weak reference to an object, then you hint to the garbage collector that it may collect the object at any time. (Not necessarily that it *will* collect the object). Some time later, when you come to examine the object, you can either turn your weak

pointer into a normal pointer, or else you can be informed that the GC did actually collect the object.

Finalisation and weak pointers can be used together to implement an in-memory object database cache. Let's imagine that we have a very large number of large user records in a file on disk. This is far too much data to be loaded into memory all at once, and anyway other programs might access the data on the disk, so we need to lock individual records when we hold copies of them in memory.

The *public* interface to our "in-memory object database cache" is going to be just two functions:

```
type record = { mutable name : string; mutable address : string }

val get_record : int -> record
val sync_records : unit -> unit
```

The `get_record` call is the only call that most programs will need to make. It gets the n^{th} record either out of the cache or from disk and returns it. The program can then read and/or update the `record.name` and `record.address` fields. The program then just literally forgets about the record! Behind the scenes, finalisation is going to write the record back out to disk at some later point in time.

The `sync_records` function can also be called by user programs. This function synchronises the disk copy and in-memory copies of all records.

OCaml doesn't currently run finalisers at exit. However you can easily force it to by adding the following command to your code. This command causes a full major GC cycle on exit:

```
at_exit Gc.full_major;;
```

Our code is also going to implement a cache of recently accessed records using the `Weak` module. The advantage of using the `Weak` module rather than hand-rolling our own code is two-fold: Firstly the garbage collector has a global view of memory requirements for the whole program, and so is in a better position to decide when to shrink the cache. Secondly our code will be much simpler.

For our example, we're going to use a very simple format for the file of users' records. The file is just a list of user records, each user record having a fixed size of 256 bytes. Each user record has just two fields (padded with spaces if necessary), the name field (64 bytes) and the address field (192 bytes). Before a record can be loaded into memory, the program must acquire an exclusive lock on the record. After the in-memory copy is written back out to the file, the program must release the lock. Here is some code to define the on-disk format

and some low-level functions to read, write, lock and unlock records:

```
open Unix

(* On-disk format. *)
let record_size = 256
let name_size = 64
let addr_size = 192

(* Low-level load/save records to file. *)
let seek_record n fd =
  lseek fd (n * record_size) SEEK_SET;
  ()

let write_record record n fd =
  seek_record n fd;
  write fd record.name 0 name_size;
  write fd record.address 0 addr_size;
  ()

let read_record record n fd =
  seek_record n fd;
  read fd record.name 0 name_size;
  read fd record.address 0 addr_size;
  ()

(* Lock/unlock the nth record in a file. *)
let lock_record n fd =
  seek_record n fd;
  lockf fd F_LOCK record_size;
  ()

let unlock_record n fd =
  seek_record n fd;
  lockf fd F_ULOCK record_size;
  ()
```

We also need a function to create new, empty in-memory `record` objects:

```
(* Create a new, empty record. *)
let new_record () =
  { name = (String.make name_size ' ');
    address = (String.make addr_size ' ') }
```

Because this is a really simple program, we're going to fix the number of records in advance:

```
(* Total number of records. *)
```

```
let nr_records = 10000

(* On-disk file. *)
let diskfile = openfile "users.bin" [ O_RDWR ] 0
```

Download users.bin.gz and decompress it before running the program.

Our cache of records is very simple:

```
(* Cache of records. *)
let cache = Weak.create nr_records
```

The `get_record` function is very short and basically composed of two halves. We grab the record from the cache. If the cache gives us `None`, then that either means that we haven't loaded this record from the cache yet, or else it has been written out to disk (finalised) and dropped from the cache. If the cache gives us `Some record` then we just return `record` (this promotes the weak pointer to the record to a normal pointer).

```
open Printf

(* The finaliser function. *)
let finaliser n record =
  printf "*** objcache: finalising record %d\n" n;
  write_record record n diskfile;
  unlock_record n diskfile

(* Get a record from the cache or off disk. *)
let get_record n =
  match Weak.get cache n with
    Some record ->
      printf "*** objcache: fetching record %d from memory cache\n" n;
      record
  | None ->
      printf "*** objcache: loading record %d from disk\n" n;
      let record = new_record () in
      Gc.finalise (finaliser n) record;
      lock_record n diskfile;
      read_record record n diskfile;
      Weak.set cache n (Some record);
      record
```

The `sync_records` function is even easier. First of all it empties the cache by replacing all the weak pointers with `None`. This now means that the garbage collector *can* collect and finalise all of those records. But it doesn't necessarily mean that the GC *will* collect the records straightaway (in fact it's not likely that it will), so to force the GC to

collect the records immediately, we also invoke a major cycle:

```
(* Synchronise all records. *)
let sync_records () =
  Weak.fill cache 0 nr_records None;
  Gc.full_major ()
```

Finally we have some test code. I won't reproduce the test code, but you can download the complete program and test code objcache.ml, and compile it with:

```
ocamlc -w s unix.cma objcache.ml -o objcache
```

Exercises

Here are some ways to extend the example above, in approximately increasing order of difficulty:

1. Implement the record as an **object**, and allow it to transparently pad/unpad strings. You will need to provide methods to set and get the name and address fields (four public methods in all). Hide as much of the implementation (file access, locking) code in the class as possible.
2. Extend the program so that it acquires a **read lock** on getting the record, but upgrades this to a **write lock** just before the user updates any field.
3. Support a **variable number of records** and add a function to create a new record (in the file). [Tip: OCaml has support for weak hashtables.]
4. Add support for **variable-length records**.
5. Make the underlying file representation a **DBM-style hash**.
6. Provide a general-purpose cache fronting a "users" table in your choice of **relational database** (with locking).