mmap: Memory Mapped Files in R

Jeffrey A. Ryan

October 25, 2010

Abstract

The `mmap` package offers a cross-platform interface for R to information that resides on disk. As dataset grow, the finite limits of random access memory constrain the the ability to process larger-than-memory files efficiently. Memory mapped files (*mmap* files) leverage the operating system demand-based paging infrastructure to move data from disk to memory as needed, and do it in a transparent and highly optimized way. This package implements a simple low-level interface to the related system calls, as well as provides a useful set of abstractions to make accessing data on disk consistent with `R` usage patterns. This paper will explore the design and implementation of the `mmap` package, provide a comprehensive look at its usage, and conclude with a look at some performance benchmarks and applications.

1 Background

As datasets of interest grow from megabytes to terabytes to petabytes, the limiting factor for processing is often the availability of memory on a system. Even if memory is sufficient to hold an entire dataset, it is usually only a subset of data that is needed at any given moment. In these instances it is beneficial to be able to only keep the data in memory that is needed at the time of the computation. Traditionally this meant iterating through a large file and reading chunks at a time, or utilizing a database system to manage the process in an external process.

The downside to the above workaround for limited memory is that a deliberate effort by the user must be made to manage the reading and removal of data so as to keep memory usage within the limits of a given system. The system level `mmap` (`MapViewOfFile` on Windows) call is designed to make this process easier and more efficient, from both a coding standpoint as well as an execution one. In fact, most modern database systems rely on a combination of mmap calls to make managing large data on limited memory systems feasible.

To use mmap on large files, it is helpful to understand what is happening internally at the C level. Given a successful initialization call to `mmap`, a pointer is returned to a byte offset of the opened file, typically the start of the file. From this point onward, all references to this pointer result in a series of bytes being read from disk into memory. The read and write operations are hidden from the developer and are highly optimized to minimize seek and copying costs.

The `mmap` package for R provides this level of access by cleanly wrapping the underlying operating system call. This minimal and direct API exposure allows for low-level bytes to be exposed to the R session. As mapped files can be shared among processes, this allows for a simple form of interprocess communication (IPC) to be available between R processes as well as between R and other system processes.

The `mmap` package also makes additional abstractions available to allow simplified data access and manipulation from within R. This includes a direct mapping of standard R data types from binary files, as well as an assortment of virtual types that are not directly supported in R, but still need to be accessed. Examples of these virtual types include single byte integers, four byte floats, and even more complex objects like C structs. This paper will focus on working through basic examples as well as give some comparisons to other solutions available in R that satisfy many of the same objectives.

2 Mapping a file

To create a mapped file, either the `as.mmap` or `mmap` function is used. Files are to be thought of as homogenous fixed-width byte strings on disk, similar to atomic vectors in R. One exception to this is the use of the `struct` type which will be covered later. For now we will begin by mapping atomic vectors.

2.1 as.mmap: memory to disk

To create a file to use, we will first use `as.mmap` to convert in-memory data into a mapped object. Here we create a vector of twenty million random numbers, which takes up about 150 MB of memory in an R session. We'll then convert it into a tempory file and map it back in using the one function `as.mmap`. Note that we reassign to the original variable to free up memory, as it is now persistent on disk.

```
> library(mmap)
> r <- rnorm(2e+07)
> gc()

           used  (Mb) gc trigger  (Mb) max used  (Mb)
Ncells   154279   8.3     350000  18.7   350000  18.7
Vcells 20144469 153.7   22546730 172.1 20146782 153.8

> r <- as.mmap(r)
> r

<mmap:/var/folde...>  (double) num [1:20000000] -0.5604756 -0.2301775 1.558708 ...

> gc()

         used (Mb) gc trigger  (Mb) max used  (Mb)
Ncells 155648  8.4     350000  18.7   350000  18.7
Vcells 144743  1.2   18037384 137.7 20159498 153.9
```

The `as.mmap` call simply writes the raw data using R's *writeBin* to a temporary file on disk. Internally this file is mapped with the appropriate *mode* corresponding to the R storage mode. Keep in mind that the data on disk is only a series of bytes. The OS *mmap* call is indifferent to the formal 'type' offering no facility to convert into a particular C type. By specifying the mode to the R-level `mmap` call though, we can now manipulate this "vector on disk" as if it was in memory and of the type we expect. First we'll extract some elements using standard R semantics, then replace these values. Finally, we will call `munmap` to properly free the resources associated with the mapping.

```
> r[1:10]

 [1] -0.56047565 -0.23017749  1.55870831  0.07050839  0.12928774  1.71506499
 [7]  0.46091621 -1.26506123 -0.68685285 -0.44566197

> r[87643]

[1] -0.2042827

> head(r)

[1] -0.56047565 -0.23017749  1.55870831  0.07050839  0.12928774  1.71506499

> tail(r)

[1]  0.36987133 -1.19688282  0.41855741 -0.17126741  0.42012201 -0.02442838

> length(r)

[1] 20000000

> r[1:10] <- 1:10
> r[87643] <- 3.14159265
> head(r)
```

```
[1] 1 2 3 4 5 6
```

```
> r[87643]
```

```
[1] 3.141593
```

```
> munmap(r)
```

By default, elements are only taken from disk when extracted via a '[' call. This allows for controlled behavior when dealing with objects that are likely to be many times the available memory. Subsetting is *always* required to access the contents of an mapped file. This is similar to the requirement in C of dereferencing the pointer to the data, and is in fact what is happening behind the scenes. To unmap the object and free the system resources, the code must call `munmap`.

Many instances of `mmap` usage will be in a read-only capacity, with data already on disk. These data can come from external processes, or pre-processed by R to be in binary form. To access, a call to `mmap` is required.

2.2 mmap: disk to memory

The basic `mmap` call consists of a file path as the *file* parameter, and specifying the *mode* of the data to be returned. The *mode* argument is unique to the `mmap` wrapper in R, and it is used to specify how the raw bytes are to be mapped into R. There are a myriad of supported types and they all strive to follow the general convention established by R in terms of calling style, namely that provided by the *what* argument of `scan` and `readBin`: `integer()` for integers, `double()` for double/numeric, etc.

The `mmap` package currently supports thirteen fixed-width (byte count) types, including 8, 16, and 24-bit signed and unsigned integers, 32-bit signed integers, floating point numbers with 32 and 64-bits, complex numbers (128-bit), fixed-width character strings (nul terminated, as `writeBin` produces), and single byte char types. Additionally, types may be combined into more complex structures via the `struct` type in mmap. This is analogous to a row-based representation where different types are adjacent on disk. This can be thought of as a `data.frame` or `list` in R.

The C-styled types are offered for compatability with external programs, as well as to minimizing disk usage for values of limited range, though there may be performance penalties for non-stanard byte alignment, so testing is required for maximum performance.

One caveat to the above type availability is that R can only handle a small subset of these on-disk types natively. All conversions to and from C-types to R-types are carried out in package-level C code, and types are automatically promoted so as not to lose precision. More discussion of types will follow in the "Types" section.

To try something a bit more interesting, we'll create some non-standard R data on disk. We'll use a temporary file and the *writeBin* function in base R to alter the size to be 8-bit signed integer values, fitting 10 integers into 10 bytes on disk.

```
> tmp <- tempfile()
> writeBin(1:10L, tmp, size=1)  # write int as 1 byte
> readBin(tmp, integer(), size=1, signed=TRUE, n=10) # read back in to verify
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

```
> file.info(tmp)$size  # only 10 bytes on disk
```

```
[1] 10
```

Now that we have our file, we can map it back into R using the `mmap` function. All the arguments to the function are detailed on the help page, and as this relies heavily on the operating system call, it is advisable to read the related man pages as well for your particular implementation. The key arguments to consider are the first two, *file* and *mode*.

`file` is the path to the binary data on disk. Recall again that this is only the raw bytestring, no meta-data is accounted for or should be included. It is possible that header information could be skipped by utilizing the `len` and `off` arguments, but this is outside of expected usage patterns.

`mode` refers to the binary type on disk. This is used by mmap to perform type conversion to and from R, as well as to correctly manage the atomic length and offset behavior seen in R when subsets of data are requested. Refer to the "Virtual Types" table in the following section for details.

```
> m <- mmap(file = tmp, mode = int8())
> m[]
 [1]  1  2  3  4  5  6  7  8  9 10
> nbytes(m)
[1] 10
> munmap(m)
```

3 Data Types

By design, R makes use of a limited subset of data types internally. These include signed integers (32-bit), floating point doubles (64-bit), and complex numbers (128-bit) for numerical computations, as well as native support for character and raw byte values. There is also a compound type available with `list`, which may contain any of the above. This relatively limited selection is quite sufficient for use in R, but it is sometimes necessary to work with data that may originate as different types or precision. `mmap`'s *mode* argument allows for transparent conversion of most common types into the supported R subset through the use of a virtual class paradigm. The following table describes the currently supported virtual type support in `mmap`.

Virtual Types

mmap	*R*	*C*	*bytes*
`raw()`	raw	unsigned char	1
`char()`	raw	char	1
`uchar()`	raw	unsigned char	1
`int8()`	integer	signed char	1
`uint8()`	integer	unsigned char	1
`int16()`	integer	signed short	2
`uint16()`	integer	unsigned short	2
`int24()`	integer	three byte int	3
`uint24()`	integer	unsigned three byte int	3
`int32()`	integer	int	4
`integer()`	integer	int	4
`real32()`	double	single precision float	4
`real64()`	double	double precision float	8
`double()`	double	double precision float	8
`cplx()`	complex	complex	16
`complex()`	complex	complex	16
`char(n)`	character	fixed-width ascii	n + 1
`character(n)`	character	fixed-width ascii	n + 1
`struct(...)`	list	struct of above types	variable

The leftmost column of the table is the constructor function used in `mmap` to create and describe this extended collection of types. The first sixteen functions are called *without parameters* and passed as the `mode` argument to the `mmap` constructor. Fixed width character vectors are mapped with a mode `char(n)`, where *n* must specify the number of characters in each element of the character mapping. A `nul` byte will be automatically assumed to increase the length of each string by one. The `struct` function takes any number of *other* valid types from above, and creates a object of class *struct*. This allows for collections of disparate types to be organized together in row-major relations.

Coercion from one type to another internally will move from least precision to most precision for extraction, but replacement functions will truncate values without warning. It is up to the user to determine the minimal precision required, and assure that the values assigned are within this range. A table of legal value ranges by type is available at the end of this document. A few examples will illustrate some basic usage.

```
> # write out a vector of upper case letters as a char * array
> writeBin(LETTERS, tmp)
> let <- mmap(tmp, char(1))
> let

<mmap:/var/folde...>  (char) chr [1:26] A B C D E F ...

> let[]

 [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"
[20] "T" "U" "V" "W" "X" "Y" "Z"

> munmap(let)
> #
> # view the data as a series of bytes instead, using raw()
> let <- mmap(tmp, raw())
> let[]

 [1] 41 00 42 00 43 00 44 00 45 00 46 00 47 00 48 00 49 00 4a 00 4b 00 4c 00 4d
[26] 00 4e 00 4f 00 50 00 51 00 52 00 53 00 54 00 55 00 56 00 57 00 58 00 59 00
[51] 5a 00

> munmap(let)
> #
> # view the data as a series of short integers
> let <- mmap(tmp, int16())
> let[]

 [1] 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89
[26] 90

> munmap(let)
```

As you can see, the data on disk is simply an array of bytes. This provides maximum flexibility as there is no associated metadata to keep track of. Byte arrays are architecture dependent but allow for very simple interprocess communication and extraction.

To make use of data other than a homogenous collection of byte types, we can map a C-style struct from disk into R's multi-type container, the *list*. We do this by means of a `struct(...)` call. For this example we'll start with an array of struct's on disk that are each composed of a 2-byte integer, a 4-byte integer, and an 8-byte floating point double. First we'll need to define our *struct*, as well as make sure it has the size we are expecting.

```
> # 2-byte (int16)
> # 4-byte (int32 or integer)
> # 8-byte float (real64 or double)
> record.type <- struct(short=int16(),int=int32(),double=real64())
> record.type

struct:
  (short) integer(0)
  (int) integer(0)
  (double) double(0)

> nbytes(record.type) # 14 bytes in total

[1] 14
```

Now we can extract individual elements of the array of structs.

```
> m[1]

$short
[1] 1

$int
[1] 366214

$double
[1] -1.382365

> m[1:3]

$short
[1] 1 2 3

$int
[1] 366214 164342 223787

$double
[1] -1.3823652  1.0580897 -0.3492971

> m[1:3, "short"]

$short
[1] 1 2 3

> length(m)

[1] 100
```

As mentioned previously, the result is a mapping to a list. It is also consistent with R that the object could also be a data.frame. `mmap` supports a set of hook functions with `extractFUN` and `replaceFUN` to allow for automatic class coercion of mapped objects upon extraction and replacement. This can be defined at the point of mapping, or added later. We'll try this here by converting our list result into a data.frame instead.

```
> extractFUN(m) <- function(X) do.call(data.frame, X)
> extractFUN(m)

function (X)
do.call(data.frame, X)
```

As you can see the object now has an extraction hook to enable on-the-fly coercion. This allows the use of raw bytes on disk (useful for application independent data sharing), while at the same type exploiting the feature rich language of R. The examples in the package also show how this can be used for other classes as well, such as Date and POSIXct time. See `example(mmap)`.

```
> m[1]

  short    int    double
1     1 366214 -1.382365

> m[2:5]

  short    int      double
1     2 164342  1.05808972
2     3 223787 -0.34929713
3     4 960135  0.06192997
4     5 571006 -1.25329358

> m[2:5, "double"] # note that subset is on mmap, returning a new data.frame
```

```
       double
1  1.05808972
2 -0.34929713
3  0.06192997
4 -1.25329358
```

```
> m[2:5, 2]
```

```
     int
1 164342
2 223787
3 960135
4 571006
```

```
> m[1:9][,"double"]  # second brackets act on d.f., as the first is on the mmap
```

```
[1] -1.38236525  1.05808972 -0.34929713  0.06192997 -1.25329358 -1.36476174
[7]  1.13131072 -1.80529501 -1.68255758
```

4 Performance

While there is a certain novelty to being able to use mapped files within R, the real value comes from performance gains. This can be seen in three distinct areas: (1) simplified interface to on-disk data, (2) reduction of memory footprint, and (3) increased throughput. Any combination of the three can be seen as a benefit and makes `mmap` an important tool for high-performance programming.

4.1 Interface Simplicity

Handling large data on disk has always been possible in R using the built-in functions to read chunks of files. This is simple in strategy, albeit highly susceptible to errors. Keeping track of offsets, as well as freeing memory explicitely in R isn't likely the most optimal use of a developer or analyst's time. mmap allows for direct access to subsets of data on disk, using standard R subsetting semantics. This allows for R code to be cleaner, as well as safer.

4.2 Reduced Memory Requirements

The primary motivation to using mmap comes from removing the need to keep an entire data object in-core at all times. The mmap package allows for direct access to subsets of data on disk, all while removing the need to have per-process memory allocated to the entire file.

On small data, this is likely to not be an issue, but as data demands grow beyond available memory the benefits to minimizing a memory footprint grow too. Even when data can fit into memory, it isn't the data that is needed per se, it is the analytical computations on that data. This puts an upper bound on data size well short of available memory.

Another facet to mapped files is in the inherent ability to share data across disparate processes. By mapping a file into memory, multiple processes can make use of the same data without requiring additional resources. Caching, reads, and writes are all managed at the system-level, and as such are highly optimized. Parallel computations on multicore architectures are simplified through the use of shared data.

4.3 Increased Throughput

For random access to large data on disk, the underlying mmap system call is as optimal a solution as modern operating systems offer. Minimizing the memory footprint in R also reduces the need for expensive allocation and garbage collections, further increasing performance. mmap also provides for automatic caching of data, as directed by the OS mechanisms. This typically incurs a small penalty upon a new chunk of data being read, but can result in faster than in-core performance on recently accessed data chunks.

An additional built in benefit from mmap objects comes from some simple Ops behavior. As mmap objects are typically larger than desired for in-memory storage, logical operations will make use of memory and time reducing techniques to return only matches to queries. The behavior is consistent with the R code `which(x==0)` to find data that matches some criteria, though operates via the standard Ops based equility test, namely `x==0`. This tends to be substantially faster though, as large logical vectors are not created, reducing both processing time as well as memory use.

```
> one.to.onemil <- 1:1000000L
> writeBin(1:1000000L, tmp)
> m <- mmap(tmp, int32())
> str(m < 100)

 int [1:99] 1 2 3 4 5 6 7 8 9 10 ...

> str(which(one.to.onemil < 100))

 int [1:99] 1 2 3 4 5 6 7 8 9 10 ...

> system.time(m < 100)

   user  system elapsed
  0.007   0.000   0.007

> system.time(which(one.to.onemil < 100))

   user  system elapsed
  0.011   0.003   0.013
```

5 Summary

The `mmap` package attempts to provide two levels of access to the POSIX system mmap call. One level offers direct byte access, as well as user specified mappings of arguments from R to the system. The second interface, albeit using the same functions, offers a more R-like level of interaction with data on disk, providing direct byte to R-type extraction and replacement. Whether used for speed, memory reduction, or simplification of code, the `mmap` package provides R with one more tool to make programming with data easier and more robust.

Table 1: Typical Valid Ranges By Type

type	*minimum*	*maximum*
int8	-128	127
uint8	0	255
int16	-32768	32767
uint16	0	65534
int24	-8388608	8388607
uint24	0	16777215
int32	-2147483648	2147483647