mmap: Memory Mapped Files

Jeffrey A. Ryan

Contents

1 Introduction

The ability to access low-level operating system calls from high-level functions is a mainstay of most contemporary software languages. As *R* straddles the line between outright language and data analysis environment, some of the tools typically available in other interpretted languages do not exist in the base distribution.

One such low-level tool is the mmap functionality, the ability to use memory mapped files without explicitely managing the process of reading and writing data to disk. This offers many advantages, including reduced cpu and memory resources, as well as the ability to transparently manage data across processes.

The new package *mmap* by Ryan, allows for *R* like syntax to be with native operating system mmap calls. The package aims to provide a thin layer in *R* around the underlying operating system's mmap functionality. As such, it relies heavily on the system calls for functionality. This increases the performance of the package while minimizing the learning curve for developers. At the same time the *mmap* package provides *R* based abstraction to make the mapped file access as transparent as possible. The downside to this thin wrapper is that *R* performance is now tied to the underlying OS implementation. In practice this is an advantage, as no additional overhead costs are incurred.

The rest of this document will explore the design criteria of the package as well as its complete functionality. This will be accomplished with commentary as well as *R* examples. The paper will conclude with a discussion of further application and tools that are now possible.

2 Design: Thin Wrapper. System API. R Syntax.

The basic goals of the package were to provide direct access to the underlying mmap behavior, within the context of an *R* work-flow. This meant keeping the *R* calls very close to the semantics of the underlying system calls, as well as provide an *R*-like experience to the end user.

Mapping a file that contains integer values via a system call in C and most other interpretted languages would return single byte char values. This behavior would of course be possible within *R*, but wouldn't allow for the richness of the language to be exposed. Instead the design allows for automatic translation through a collection of special `Ctypes` that replicate native *R* and machine atomic data types.

This combines expected *R* results, with the expected OS semantics. The available system types even allow for virtual support of types not expressable in *R*, so as to allow for better disk space management, as well as allow for interoperability with non-*R* programs.

A motivating reason to use memory-mapped files in general is to simplify the interface to data residing on disk. In *R*, one would typically have to read in an entire file to operate on a subset of it. There are numerous packages that make this less of a problem, notably *bigmemory* (Kane and Emerson) and *ff* (), but both provide additional opacity to the underlying system calls. Ideally the data residing on disk should be treated as closely to possible from an interface perspective as data residing in memory, while at the same time preserving the feel of system `mmap` calls. This reduces the need for the programmer to distinguish between the two when designing code. Small data can scale to large data with little more than a specification of where to source the data from.

mmap makes this possible by allowing basic operations on vectors to be available on memory-mapped files as well. This includes all the basic mathematical functionality as well as some basic tools to investigate the mmap structure itself.

3 Using