Memory-mapped I/O

`Mmap.Anonymous` — Type

```
Mmap.Anonymous(name::AbstractString="", readonly::Bool=false, create::Bool=true
```

Create an `IO`-like object for creating zeroed-out mmapped-memory that is not tied to a file for use in `Mmap.mmap`. Used by `SharedArray` for creating shared memory arrays.

Examples

```
julia> using Mmap

julia> anon = Mmap.Anonymous();

julia> isreadable(anon)
true

julia> iswritable(anon)
true

julia> isopen(anon)
true
```

`Mmap.mmap` — Function

```
Mmap.mmap(io::Union{IOStream,AbstractString,Mmap.AnonymousMmap}[, type::Type{Ar
Mmap.mmap(type::Type{Array{T,N}}, dims)
```

Create an `Array` whose values are linked to a file, using memory-mapping. This provides a convenient way of working with data too large to fit in the computer's memory.

The type is an `Array{T,N}` with a bits-type element of `T` and dimension `N` that determines how the bytes of the array are interpreted. Note that the file must be stored in binary format, and no format conversions are possible (this is a limitation of operating systems, not Julia).

`dims` is a tuple or single `Integer` specifying the size or length of the array.

The file is passed via the stream argument, either as an open `IOStream` or filename string. When you initialize the stream, use `"r"` for a "read-only" array, and `"w+"` to create a new array used to write values to disk.

If no `type` argument is specified, the default is `Vector{UInt8}`.

Optionally, you can specify an offset (in bytes) if, for example, you want to skip over a header in the file. The default value for the offset is the current stream position for an `IOStream`.

The `grow` keyword argument specifies whether the disk file should be grown to accommodate the requested size of array (if the total file size is < requested array size). Write privileges are required to grow the file.

The `shared` keyword argument specifies whether the resulting `Array` and changes made to it will be visible to other processes mapping the same file.

For example, the following code

```
# Create a file for mmapping
# (you could alternatively use mmap to do this step, too)
using Mmap
A = rand(1:20, 5, 30)
s = open("/tmp/mmap.bin", "w+")
# We'll write the dimensions of the array as the first two Ints in the file
write(s, size(A,1))
write(s, size(A,2))
# Now write the data
write(s, A)
close(s)

# Test by reading it back in
s = open("/tmp/mmap.bin")   # default is read-only
m = read(s, Int)
n = read(s, Int)
A2 = Mmap.mmap(s, Matrix{Int}, (m,n))
```

creates a `m`-by-`n` `Matrix{Int}`, linked to the file associated with stream `s`.

A more portable file would need to encode the word size – 32 bit or 64 bit – and endianness information in the header. In practice, consider encoding binary data using standard formats like HDF5 (which can be used with memory-mapping).

```
Mmap.mmap(io, BitArray, [dims, offset])
```

Create a `BitArray` whose values are linked to a file, using memory-mapping; it has the same purpose, works in the same way, and has the same arguments, as `mmap`, but the byte representation is different.

Examples

```
julia> using Mmap

julia> io = open("mmap.bin", "w+");

julia> B = Mmap.mmap(io, BitArray, (25,30000));

julia> B[3, 4000] = true;

julia> Mmap.sync!(B);

julia> close(io);

julia> io = open("mmap.bin", "r+");

julia> C = Mmap.mmap(io, BitArray, (25,30000));

julia> C[3, 4000]
true

julia> C[2, 4000]
false

julia> close(io)

julia> rm("mmap.bin")
```

This creates a 25-by-30000 `BitArray`, linked to the file associated with stream `io`.

`Mmap.sync!` — Function

```
Mmap.sync!(array)
```

Forces synchronization between the in-memory version of a memory-mapped `Array` or `BitArray` and the on-disk version.

« Markdown Pkg »