When should I use mmap for file access?

Asked 11 years, 2 months ago Active 4 months ago Viewed 101k times

265

158

POSIX environments provide at least two ways of accessing files. There's the standard system calls `open()`, `read()`, `write()`, and friends, but there's also the option of using `mmap()` to map the file into virtual memory.

When is it preferable to use one over the other? What're their individual advantages that merit including two interfaces?

c file-io posix mmap

asked Nov 3 '08 at 7:56

Peter Burns
39.1k 7 32 54

15 See also mmap() vs. reading blocks and this post by Linus Torvalds referenced in one of the answers there. – MvG May 2 '14 at 21:27

6 Answers

290

`mmap` is great if you have multiple processes accessing data in a read only fashion from the same file, which is common in the kind of server systems I write. `mmap` allows all those processes to share the same physical memory pages, saving a lot of memory.

`mmap` also allows the operating system to optimize paging operations. For example, consider two programs; program `A` which reads in a `1MB` file into a buffer creating with `malloc`, and program B which `mmaps` the 1MB file into memory. If the operating system has to swap part of `A`'s memory out, it must write the contents of the buffer to swap before it can reuse the memory. In `B`'s case any unmodified `mmap`'d pages can be reused immediately because the OS knows how to restore them from the existing file they were `mmap`'d from. (The OS can detect which pages are unmodified by initially marking writable `mmap`'d pages as read only and catching seg faults, similar to Copy on Write strategy).

`mmap` is also useful for inter process communication. You can `mmap` a file as read / write in the processes that need to communicate and then use synchronization primitives in the `mmap'd` region (this is what the `MAP_HASSEMAPHORE` flag is for).

One place `mmap` can be awkward is if you need to work with very large files on a 32 bit machine. This is because `mmap` has to find a contiguous block of addresses in your process's address space that is large enough to fit the entire range of the file being mapped. This can become a problem if your address space becomes fragmented, where you might have 2 GB of address space free, but no individual range of it can fit a 1 GB file mapping. In this case you may have to map the file in

Another potential awkwardness with `mmap` as a replacement for read / write is that you have to start your mapping on offsets of the page size. If you just want to get some data at offset `x` you will need to fixup that offset so it's compatible with `mmap` .

And finally, read / write are the only way you **can** work with some types of files. `mmap` can't be used on things like pipes and ttys.

edited Sep 20 '19 at 8:21

patryk.beza
3,508 4 29 46

answered Nov 3 '08 at 8:01

Don Neufeld
20.3k 10 48 47

9 Can you use mmap() on files that are growing? Or is the size fixed at the point when you allocate the mmap() memory/file? – Jonathan Leffler Nov 3 '08 at 23:44

27 When you make the mmap call you have to specify a size. So if you want to do something like a tail operation it's not very suitable. – Don Neufeld Nov 4 '08 at 2:13

5 Afaik `MAP_HASSEMAPHORE` is specific to BSD. – Patrick Schlüter May 24 '10 at 8:13

5 @JonathanLeffler Certainly you can use mmap() on files that are growing, but you have to call mmap() again with the new size when the file reaches the limit of the space you initially allocated. LevelDB's PosixMmapFile give you a good example. But it stopped using mmap from 1.15. You can get the old version from Github – baotiao Mar 4 '15 at 9:16

2 mmap could also be useful in case a file needs to be processed in multiple passes : the cost of allocating virtual memory pages is only paid once. – Jib Dec 7 '16 at 13:26

65

One area where I found mmap() to not be an advantage was when reading small files (under 16K). The overhead of page faulting to read the whole file was very high compared with just doing a single read() system call. This is because the kernel can sometimes satisify a read entirely in your time slice, meaning your code doesn't switch away. With a page fault, it seemed more likely that another program would be scheduled, making the file operation have a higher latency.

answered Nov 3 '08 at 22:16

Ben Combee
14.7k 6 36 38

4 +1 I can confirm that. For small files it's faster to `malloc` a piece of memory and making 1 `read` into it. This allows to have the same code that handles memory maps handle malloc'ed . – Patrick Schlüter May 24 '10 at 8:18

34 This said, your justification for it is not right. The scheduler has nothing at all to do with the difference. The difference comes from the write accesses to the page tables, which is a global structure of the kernel holding what processes hold which memory page and its access rights. This operation can be very costly (it can invalided cache lines, it can through away TLB, the table is global so has to be protected against concurrent access, etc.). You need a certain size of map so that the overhead of `read` accesses is higher than the overhead of virtual memory manipulation. – Patrick Schlüter May 24 '10 at 8:36

@PatrickSchlüter Okay, I understand that there is overhead at the start of mmap() which involves modifying the page table. Say we map 16K of a file to memory. For a page size of 4K, `mmap` has to update 4 entries in the page table. But using `read` to copy into a buffer of 16K also involves updating 4 page table entries, not to

44

`mmap` has the advantage when you have random access on big files. Another advantage is that you access it with memory operations (memcpy, pointer arithmetic), without bothering with the buffering. Normal I/O can sometimes be quite difficult when using buffers when you have structures bigger than your buffer. The code to handle that is often difficult to get right, mmap is generally easier. This said, there are certain traps when working with `mmap` . As people have already mentioned, `mmap` is quite costly to set up, so it is worth using only for a given size (varying from machine to machine).

For pure sequential accesses to the file, it is also not always the better solution, though an appropriate call to `madvise` can mitigate the problem.

You have to be careful with alignment restrictions of your architecture(SPARC, itanium), with read/write IO the buffers are often properly aligned and do not trap when dereferencing a casted pointer.

You also have to be careful that you do not access outside of the map. It can easily happen if you use string functions on your map, and your file does not contain a \0 at the end. It will work most of the time when your file size is not a multiple of the page size as the last page is filled with 0 (the mapped area is always in the size of a multiple of your page size).

edited May 2 '18 at 13:22

answered May 24 '10 at 8:53

Patrick Schlüter

9,914 36 41

25

In addition to other nice answers, a quote from Linux system programming written by Google's expert Robert Love:

> ### Advantages of `mmap( )`
>
> Manipulating files via `mmap( )` has a handful of advantages over the standard `read( )` and `write( )` system calls. Among them are:
>
> - Reading from and writing to a memory-mapped file avoids the extraneous copy that occurs when using the `read( )` or `write( )` system calls, where the data must be copied to and from a user-space buffer.
> - Aside from any potential page faults, reading from and writing to a memory-mapped file does not incur any system call or context switch overhead. It is as simple as accessing memory.
> - When multiple processes map the same object into memory, the data is shared among all the processes. Read-only and shared writable mappings are shared in their entirety; private writable mappings have their not-yet-COW (copy-on-write) pages shared.
> - Seeking around the mapping involves trivial pointer manipulations. There is no need for the `lseek( )` system call.
>
> For these reasons, `mmap( )` is a smart choice for many applications.

- Memory mappings are always an integer number of pages in size. Thus, the difference between the size of the backing file and an integer number of pages is "wasted" as slack space. For small files, a significant percentage of the mapping may be wasted. For example, with 4 KB pages, a 7 byte mapping wastes 4,089 bytes.
- The memory mappings must fit into the process' address space. With a 32-bit address space, a very large number of various-sized mappings can result in fragmentation of the address space, making it hard to find large free contiguous regions. This problem, of course, is much less apparent with a 64-bit address space.
- There is overhead in creating and maintaining the memory mappings and associated data structures inside the kernel. This overhead is generally obviated by the elimination of the double copy mentioned in the previous section, particularly for larger and frequently accessed files.

For these reasons, the benefits of `mmap( )` are most greatly realized when the mapped file is large (and thus any wasted space is a small percentage of the total mapping), or when the total size of the mapped file is evenly divisible by the page size (and thus there is no wasted space).

edited Dec 20 '17 at 18:20

Toby Speight
20.5k 13 49 74

answered Sep 27 '17 at 7:46

Miljen Mikic
12.3k 5 47 54

12

Memory mapping has a potential for a huge speed advantage compared to traditional IO. It lets the operating system read the data from the source file as the pages in the memory mapped file are touched. This works by creating faulting pages, which the OS detects and then the OS loads the corresponding data from the file automatically.

This works the same way as the paging mechanism and is usually optimized for high speed I/O by reading data on system page boundaries and sizes (usually 4K) - a size for which most file system caches are optimized to.

edited Jul 7 '14 at 23:28

AndyG
31.5k 7 81 118

answered Nov 3 '08 at 8:08
grover

14 Note that mmap() is not always faster than read(). For sequential reads, mmap() will give you no measurable advantage - this is based on empirical and theoretical evidence. If you don't believe me, write your own test. – Tim Cooper Dec 9 '08 at 4:40

1 I can give numbers coming from our project, a kind of text index for a phrase database. The index is several Gigabyte big and the keys are held in a ternary tree. The index is still growing in parallel to read access, access outside the mapped parts are made via `pread` . On Solaris 9 Sparc (V890) the pread access are between 2 and 3 times slower than the `memcpy` from the mmap. But you're right that sequential access are not necessarly faster. – Patrick Schlüter May 24 '10 at 8:25

17 Just a little nitpick. It doesn't work like the paging mechanism, it is the paging mechanism. Mapping a file is assigning a memory area to a file instead of the anonymous swap file. – Patrick Schlüter May 24 '10 at 8:59

1

pages. If one allocates a buffer in the process's address space, then uses `read()` to fill the buffer from a file, the memory pages corresponding to that buffer are now *dirty* since they have been written to.

Dirty pages can not be dropped from RAM by the kernel. If there is swap space, then they can be paged out to swap. But this is costly and on some systems, such as small embedded devices with only flash memory, there is no swap at all. In that case, the buffer will be stuck in RAM until the process exits, or perhaps gives it back with `madvise()` .

Non written to `mmap()` pages are clean. If the kernel needs RAM, it can simply drop them and use the RAM the pages were in. If the process that had the mapping accesses it again, it cause a page fault the kernel re-loads the pages from the file they came from originally. The same way they were populated in the first place.

This doesn't require more than one process using the mapped file to be an advantage.

answered Sep 17 '19 at 6:04

TrentP

2,116 9 25