memory mapped files

- map a file into virtual memory (heap)
- no more `read()` or `write()`, just ordinary memory accesses
- map a file into the address space of a process - the file is mapped into virtual memory
- simplifies file access by treating file I/O through memory rather than `read()`/`write()` system calls
- page faults may read a page of file data from disk to memory
- allows several processes to map the same file, allowing pages in memory to be shared - permits different processes to communicate very efficiently
- requires synchronization between processes that are storing/fetching information to/from the shared memory region
- faster when copying one file to another

mmap()

- creates a new mapping in the virtual address space of the calling process

```
void * mmap(void *addr, size_t len, int prot, int flags, int fd, off_t
offset);
```

```
- returns de address of the new mapping
- addr
        - NULL: kernel chooses the address at which to create the mapping
(most portable)
        - not-NULL: kernel takes it as a hint - Linux will map at a nearby
page boundary
- length, offset and fd: initialisation uses `length` bytes starting at
`offset` in the file referred to by the file descriptor `fd`
- prot: describes the desired memory protection of the mapping
        - `PROT_NONE` or bitwise OR (`|`) of one or more of the following:
                - `PROT_EXEC` - Pages may be executed
                - `PROT_READ` - Pages may be read
                - `PROT_WRITE` - Pages may be written
                - `PROT_NONE` - Pages may not be accessed
- flags: determines whether updates to the mapping are visible to toher
processes mapping the same region, and whether updates are carried thorugh
to the underlying file
        - MAP_SHARED - Updates to the mapping are visible to other
processes that map this file and are carried through to the underlying
```

```
file. The file may not actually be updated until msync or munmap is
called.
        - MAP_PRIVATE - Create a private copy-on-write mapping. Updates to
the mapping are not visible to other processes mapping the same file. It
is unspecified whether changes made to the file after the mmap() call are
visible in the mapped region.
        - n MAP_ANONYMOUS - The mapping is not backed by any file; its
contents are initialized to zero (e.g. to share a mapped region between
two processes).
        - MAP_FIXED - Does not interpret `addr` as a hint: place the
mapping at exactly that address. `addr` must be a multiple of the page
size.
        - MAP_POPULATE - Populates page tables for a mapping. For a file
mapping, this causes read-ahead on the file. Later accesses to the mapping
will not be blocked by page faults.
```

⚠ Warning

`offset` must be a multiple of the page size as returned by sysconf(_SC_PAGE_SIZE)

⚠ Warning

`prot` must not conflict with the open mode of the file

🖉 Note

- Use of a mapped region can result in these signals:
  - SIGSEGV - Attempted write into a region mapped as read-only.
  - SIGBUS - Attempted access to a portion of the buffer that does not correspond to the file (for example, beyond the end of the file, including the case where another process has truncated the file).

⚠ Warning

- Memory mapped by `mmap()` is preserved across fork(), with the same attributes.
- A file is mapped in multiples of the page size. For a file that is not a multiple of the page size, the remaining memory is zeroed when mapped, and writes to that region are not written out to the file.
- The effect of changing the size of the underlying file of a mapping, on the pages that correspond to added or removed regions of the file, is unspecified.

- A file cannot be appended with mmap. The file size must be changed first.
- Closing the file descriptor of the mapped file does not unmap the file from memory.

mmap2()

- works like mmap(), but the final argument (`pgoffset`) is expressed in 4096-byte units - enables applications that use 32-bit off_t to map large files (up to 2^44 bytes)

```
void *mmap2(void *addr, size_t length,int prot, int flags,int fd, off_t
pgoffset);
```

munmap()

- deletes the mappings for the specified address range

```
int munmap(void *addr, size_t length);
```

```
- unmap an entire mapping:
        - `addr`: addres returned by `mmap()`
        - `length`: the value in the `mmap()` call
- unmap part of a mapping:
        - shrinks or cuts it in two, depending on where the unmapping
occurs
- regions are automatically unmapped when the process is terminated
- `addr`must be a multiple of page size
- all pages containing a part of the indicated range will be unmapped
- subsequent references to these pages will generate SIGSEGV
- it IS NOT an error if the indicated range does not contain any mapped
pages
```

msync()

- flushes the changes made in memory to the underlying file

- updates the part of the file that corresponds to the memory area, starting at `addr` and having a length `length`

```
int msync(void *adress, size_t length, int flags);
```

```
- flags:
        - `MS_SYNC`- This flag makes sure the data is actually written to
disk. Normally, msync only makes sure that accesses to a file with
conventional I/O reflect the recent changes.
        - MS_ASYNC - This tells `msync()` to begin the synchronization,
but not to wait for it to complete.
- returns o for success, -1 for error
```

> ⚠ **Warning**
>
> - in shared mappings, it is the kernel that decides when to write to the underlying file
> - without this call, there is no guarantee that changes are written back before munmap() is called

sysconf()

- Memory mapping only works on entire pages of memory.
- Addresses for mapping must be page-aligned, and length values will be rounded up.

To determine the size of a page, use:

```
size_t page_size = (size_t) sysconf(_SC_PAGE_SIZE);
```