GWDG AG-C

Memory Mapping for IO and Sharing Data Between Processes

Dr. Freja Nordsiek

Table of contents

Why

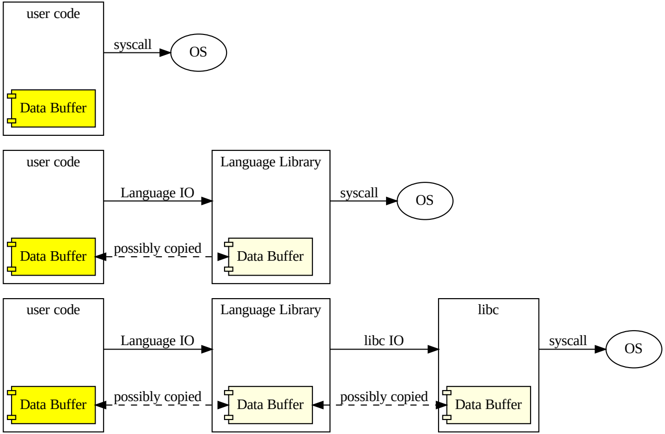
Why do IO?

- Give program data
- Get program results
- Move data to another device

Why share data between processes?

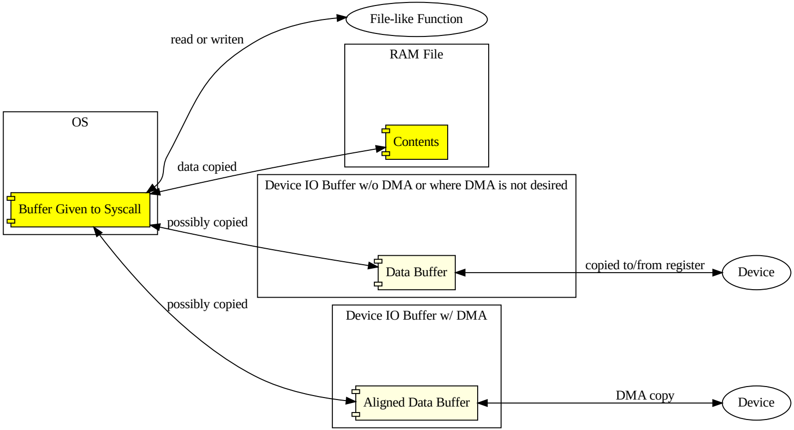
- Use more cores to get results faster
- Output requires two or more programs to work together
- intra-node MPI communication

How IO Is Done Without Memory Mapping– User Side



Note: functions can return earlier on the chain depending on buffering, size of data, previous operations, etc.

How IO Is Done Without Memory Mapping – OS Side



Note: depending on buffering/caching and previous operations, not every syscall results in a read/write.

The Two Ways to Access A File

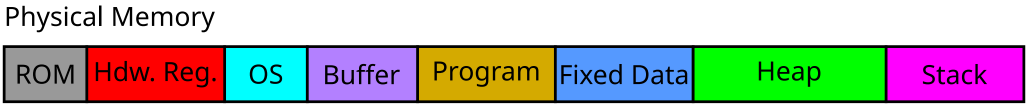
Standard IO

- Includes POSIX, libc, libstdc++, Python, etc.
- Keep track of a file position that can be moved
- Read and/or write bytes after the current file position

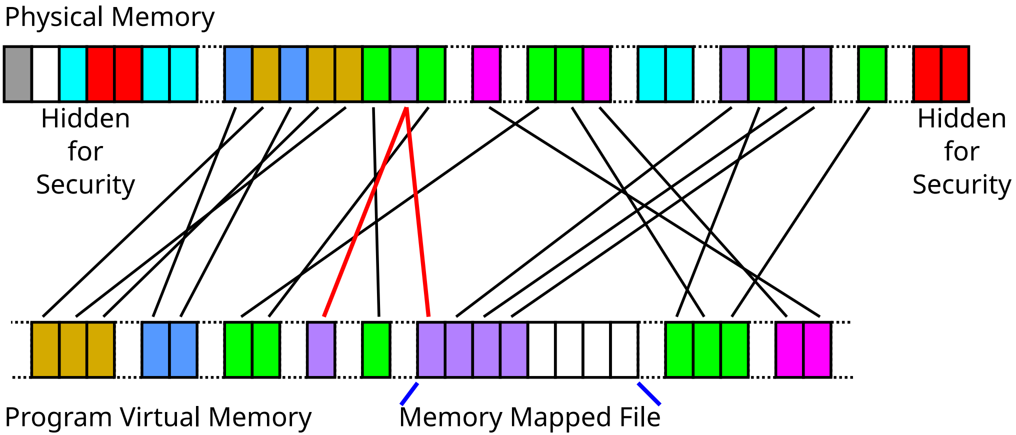
Memory mapped IO

- Access file as if it was an array of bytes
- Requires an MMU (Memory Management Unit) – Virtual Memory
- OS transparently handles the actual low-level reading/writing the file to/from memory

Memory Layout – No MMU



Memory Layout – With MMU



Pages

Memory broken into pages

- CPU has fixed allowed sizes
- OS picks one or more
- Default size is usually 4 KiB (x86), 16 KiB, or 64 KiB
- OS sometimes support huge pages (2 MiB, 1 GiB, etc.) at the same time

Virtual Memory

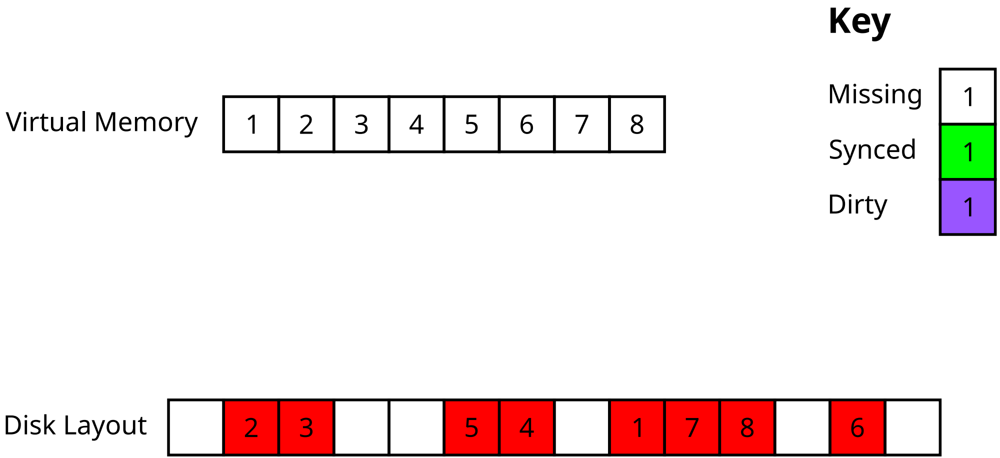
- Page table translates the virtual pages to their physical page
- Physical page can be mapped to 1+ virtual pages
- Page fault if program accesses page not in the page table
- OS provides functionality on page fault (e.g segfault, allocation, IO, etc.)
- Pages with writes are marked as dirty for OS to respond to

Get Page Size on POSIX

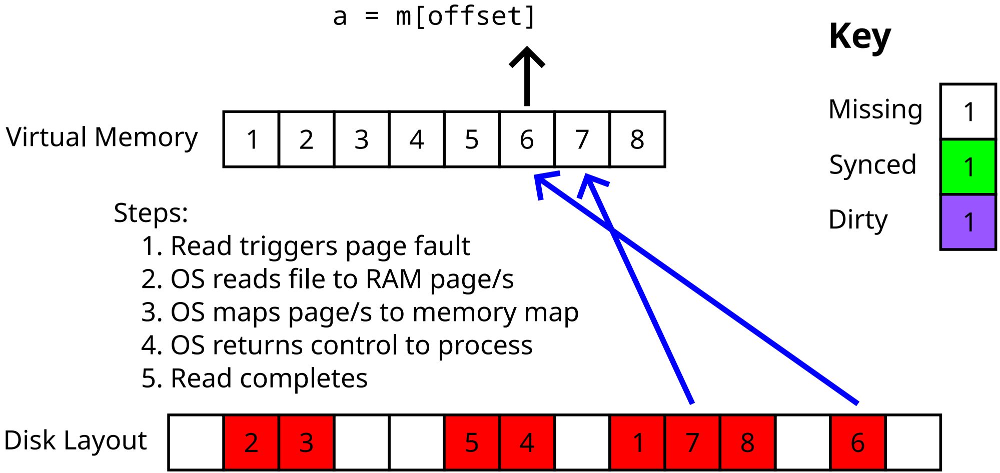
Get PAGE_SIZE

```
#include <unistd.h>
long page_size = sysconf(_SC_PAGE_SIZE);
```

After mmap


Key
Missing
1
Synced
1
Dirty
1
Virtual Memory
1
2
3
4
5
6
7
8
Disk Layout
2
3
5
4
1
7
8
6

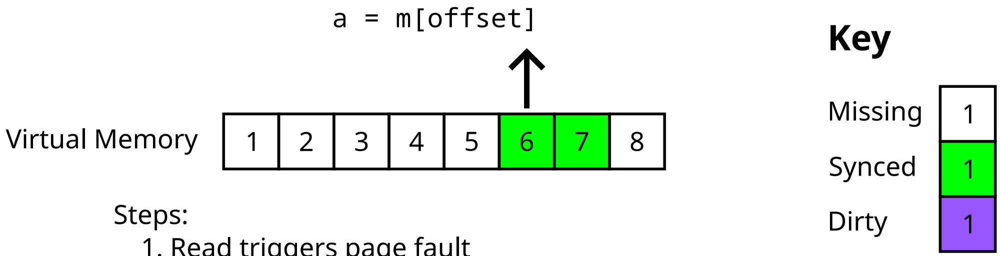
Start of First Read



Steps:

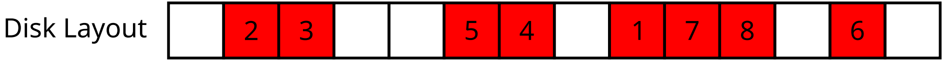
1. Read triggers page fault
2. OS reads file to RAM page/s
3. OS maps page/s to memory map
4. OS returns control to process
5. Read completes

After First Read

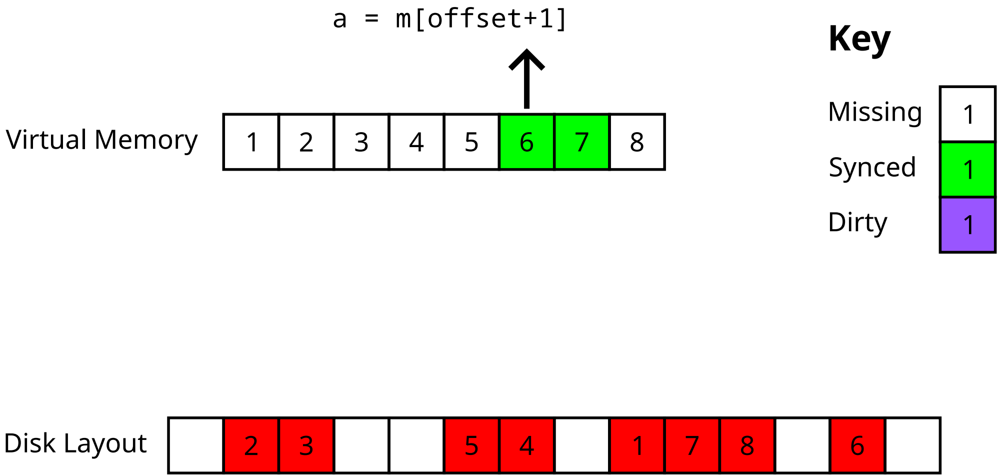


Steps:

1. Read triggers page fault
2. OS reads file to RAM page/s
3. OS maps page/s to memory map
4. OS returns control to process
5. Read completes

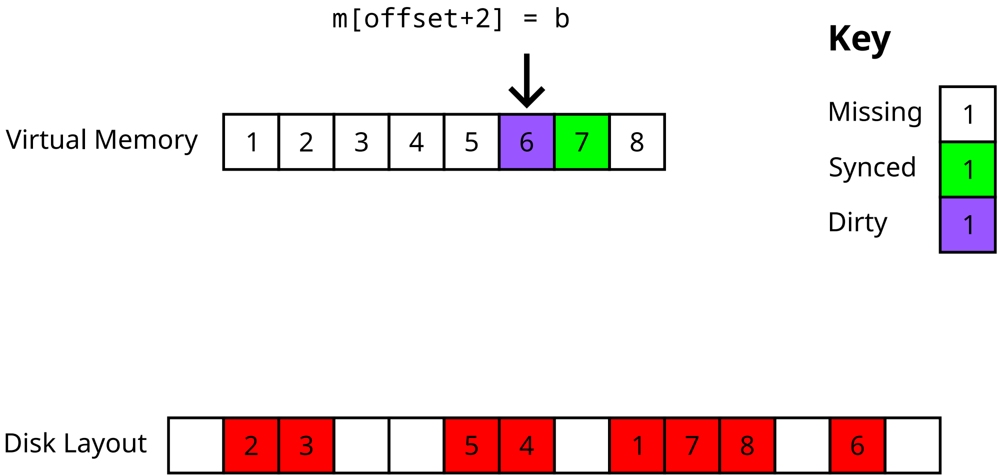


Second Read

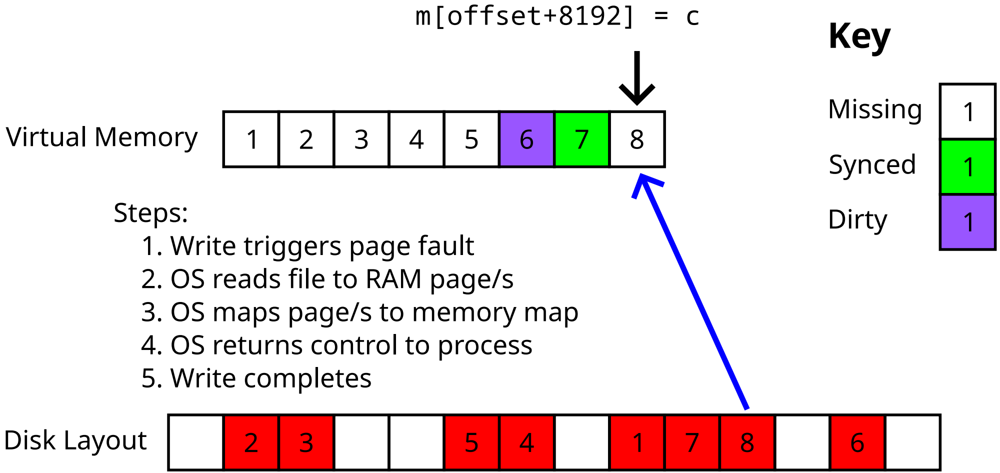


a = m[offset+1]
Virtual Memory
1
2
3
4
5
6
7
8
Key
Missing
Synced
Dirty
1
1
1
Disk Layout
2
3
5
4
1
7
8
6

Write to Read Page



Start of Write to Unread Page



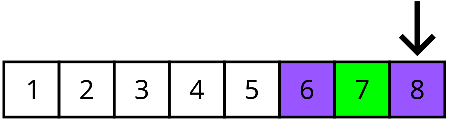
Steps:

1. Write triggers page fault
2. OS reads file to RAM page/s
3. OS maps page/s to memory map
4. OS returns control to process
5. Write completes

After Write to Unread Page

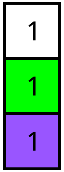
```
m[offset+8192] = c
```

Virtual Memory



Key

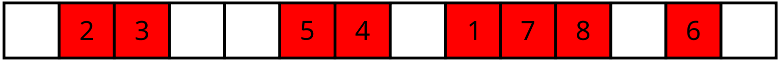
Missing
Synced
Dirty



Steps:

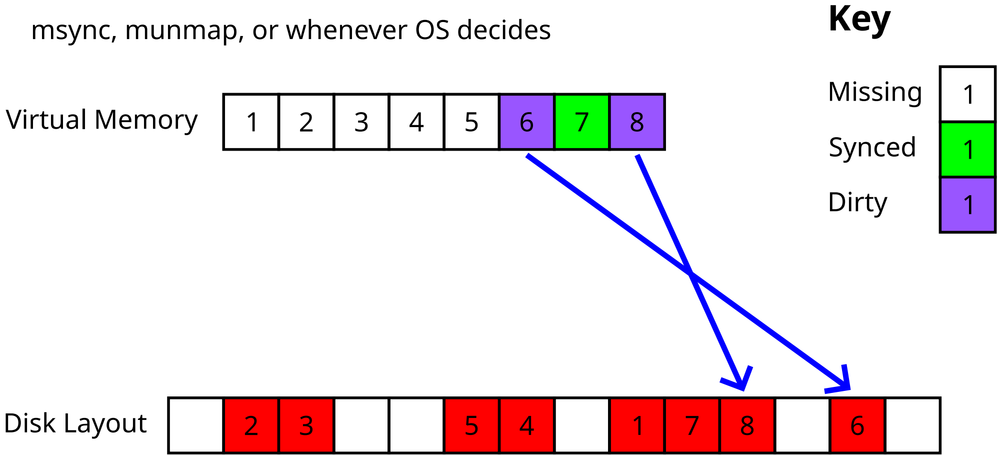
1. Write triggers page fault
2. OS reads file to RAM page/s
3. OS maps page/s to memory map
4. OS returns control to process
5. Write completes

Disk Layout



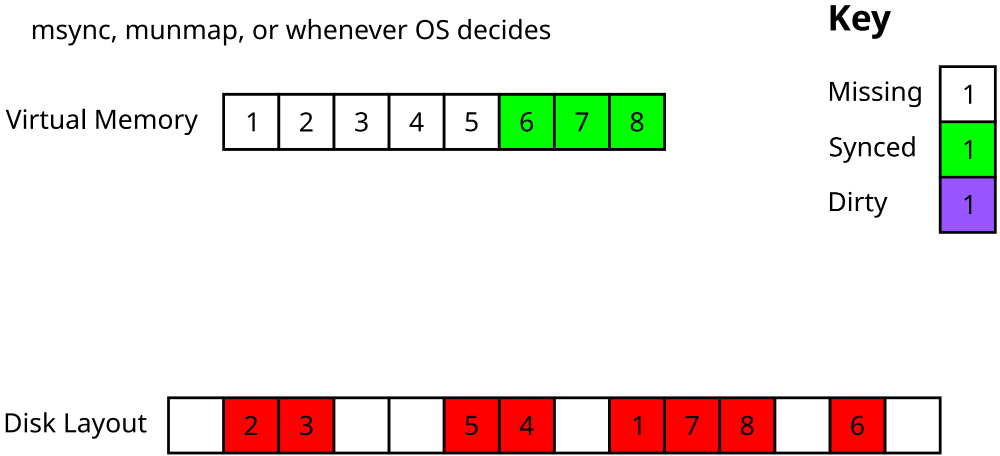
Start of Synchronization

msync, munmap, or whenever OS decides



After Synchronization

msync, munmap, or whenever OS decides



Introduction

Basic Steps

1. Open file with POSIX open
2. Do any desired preparations with standard POSIX IO including `ftruncate`
3. Memory map file with `mmap`
4. Do any desired reading, writing, and synchronizing on memory mapped area
5. Delete the mapping with `munmap`

Notes

- POSIX file handle can be closed at any point after `mmap`
- **DON’T ftruncate FILE TO SHORTER THAN END OF MAPPING!**

Review – libc and POSIX Standard IO Functions

libc calls	POSIX calls
`#include <stdio.h>`	`#include <fcntl.h>` `#include <unistd.h>`
`FILE *fopen(char *filename, char *mode)`	`int open(const char *pathname, int flags)`
`int fclose(FILE *f)`	`int close(int fd)`
`int fflush(FILE *f)`	
	`int fdatasync(int fd)` `int fsync(int fd)`
	`int ftruncate(int fd, off_t length)`
`int fseek(FILE *f, long offset, int origin)`	`off_t lseek(int fd, off_t offset, int whence)` `off64_t lseek64(int fd, off64_t offset, int whence)`†
`long ftell(FILE *f)`	`off_t lseek(int fd, 0, SEEK_CUR)` `off64_t lseek64(int fd, 0, SEEK_CUR)`†
`size_t fread(void *buf, size_t size, size_t count, FILE *f)`	`ssize_t read(int fd, void *buf, size_t count)`
`size_t fwrite(void *buf, size_t size, size_t count, FILE *f)`	`ssize_t write(int fd, const void *buf, size_t count)`

†Linux extensions to POSIX

POSIX file handles are `int`.

On Linux, many things are file handles in addition to actual files

Open File

```
int fd = open("foo.txt", FLAGS);
int fd = open("foo.txt", FLAGS, MODE);
```

FLAGS are OR-ed together

Open File

```
int fd = open("foo.txt", FLAGS);
int fd = open("foo.txt", FLAGS, MODE);
```

FLAGS are OR-ed together

Access FLAGS

read	O_RDONLY
write	O_WRONLY
read and write	O_RDWR

Creation FLAGS

create if not exist	O_CREATE
must create	O_EXCL
truncate	O_TRUNC
append	O_APPEND

Open File

```
int fd = open("foo.txt", FLAGS);
int fd = open("foo.txt", FLAGS, MODE);
```

FLAGS are OR-ed together

Access FLAGS

read	`O_RDONLY`
write	`O_WRONLY`
read and write	`O_RDWR`

Creation FLAGS

create if not exist	`O_CREATE`
must create	`O_EXCL`
truncate	`O_TRUNC`
append	`O_APPEND`

Synchronization FLAGS

`fsync` every write	`O_SYNC`
non-blocking	`O_NONBLOCK`

Aligned IO FLAGS (Linux extension)

aligned reads/writes only	`O_DIRECT`

Open File

```
int fd = open("foo.txt", FLAGS);
int fd = open("foo.txt", FLAGS, MODE);
```

MODE are OR-ed together and set permissions

These are the standard values used with chmod

Owner

read	S_IRUSR
write	S_IWUSR
execute	S_IXUSR

Group

read	S_IRGRP
write	S_IWGRP
execute	S_IXGRP

Other

read	S_IROTH
write	S_IWOTH
execute	S_IXOTH

Important Note on Opening File

Readonly Mapping

`FLAGS` can include `O_RDONLY` or `O_RDWR`.

Writable Mapping

`FLAGS` must include `O_RDWR`.

- Even if no reads are planned
- This is because a page must be read before it can be written

Change File Length

Change file length

```
int err = ftruncate(fd, new_length);
```

Increasing file length

- `ftruncate` can take a new length that is bigger than the current length
- File will be pre-allocated on the filesystem
- Pre-allocation is much more efficient than writing zeros one at a time or even blocks at a time

Notes

- **NEVER REDUCE FILE SIZE TO LESS THAN THE MAPPED REGION**
- Pre-allocate a file for writing if you want one contiguous mapping

Review – Close and Standard Read And Write

Close file

```
int err = close(fd);
```

Read

```
ssize_t bytes_read = read(fd, buffer, bytes_to_read);
```

Write

```
ssize_t bytes_written = write(fd, buffer, bytes_to_write);
```

Memory Map File

```c
#include <sys/mman.h>
void * mmap(void *ADDR, size_t LENGTH, int PROT, int FLAGS, int FD, off_t OFFSET);
```

On Success

- Returns starting address of mapping
- `close`-ing file handle FD does not affect mapping

On Failure

- Returns `MAP_FAILED`
- Sets `errno`

Memory Map File

```c
#include <sys/mman.h>
void * mmap(void *ADDR, size_t LENGTH, int PROT, int FLAGS, int FD, off_t OFFSET);
```

ADDR

- Suggested mapping start address
- NULL to let OS decide
- FLAGS with MAP_FIXED makes it a demand
- Should be page aligned

FD

- File handle of file to map
- -1 to not map any file

Memory Map File

```c
#include <sys/mman.h>
void * mmap(void *ADDR, size_t LENGTH, int PROT, int FLAGS, int FD, off_t OFFSET);
```

OFFSET

- File offset to start mapping
- Relative to beginning
- Must be page aligned

LENGTH

- Number of bytes to map
- Must be postive

$$\mathtt{LENGTH} + \mathtt{OFFSET} < \mathtt{FILE\_LENGTH}$$

Memory Map File

```c
#include <sys/mman.h>
void * mmap(void *ADDR, size_t LENGTH, int PROT, int FLAGS, int FD, off_t OFFSET);
```

Protection PROT (OR together)

none	PROT_NONE
read	PROT_READ
write	PROT_WRITE
execute	PROT_EXEC

Memory Map File

```c
#include <sys/mman.h>
void * mmap(void *ADDR, size_t LENGTH, int PROT, int FLAGS, int FD, off_t OFFSET);
```

FLAGS for a file (OR together)

MAP_PRIVATE	Private copy, writes are not propogated
MAP_SHARED	All mappings synchronized, writes synchronize to file
MAP_SHARED_VALIDATE	Like MAP_SHARED but reject unknown FLAGS
MAP_FIXED	ADDR is a demand
MAP_FIXED_NOREPLACE	Like MAP_FIXED but don't clobber another mapping
MAP_POPULATE	Pre-pagefault whole mapping
MAP_32BIT	Map into the lower 2 GiB of memory

Reading and Writing

- Read and write just like an array
- OS transparently handles page faults and write-back
- Note, page faults do cause latency

Reading data

```c
uint8_t * m = (uint8_t *)mmap(...);

if (m == MAP_FAILED)
    _Exit(1);

uint8_t a = m[10];
```

Writing data

```c
uint8_t * m = (uint8_t *)mmap(...);

if (m == MAP_FAILED)
    _Exit(1);

m[391] = 4;
```

Synchronizing to File

```
int err = msync(void *ADDR, size_t LENGTH, int FLAGS);
```

- Synchronizes LENGTH bytes staring at address ADDR
- ADDR can be anywhere in mapping
- ADDR does not have to be page aligned

Synchronization FLAGS

MS_ASYNC	Schedule synchronization but return immediately
MS_SYNC	Start synchronization and return when complete
MS_INVALIDATE	Invalidate other mappings of same file to refresh them

Unmapping

```
int err = munmap(void *ADDR, size_t LENGTH);
```

- Unmaps pages starting from ADDR extenging out LENGTH bytes
- ADDR can be anywhere in mapping
- ADDR must be page aligned
- **If the address range partially overlaps a page, the whole page is unmapped**

Example

```c
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>

#define DATA "hello"

int main()
{
    int f = open("foo.txt", O_RDWR | O_CREAT | O_TRUNC);
    const size_t length = strlen(DATA);
    write(f, DATA, length);
    char * m = (char *)mmap(NULL, length, PROT_READ | PROT_WRITE, MAP_SHARED, f, 0);
    close(f);
    m[0] = 'H';
    fwrite(m, 1, length, stdout);
    fputc('\n', stdout);
    munmap(m, length);
    return 0;
}
```

Another Example

Example with full benchmarks comparing memory mapped writing to other output methods:

`https://gitlab-ce.gwdg.de/gwdg/hpc-usage-examples/-/tree/main/performance-engineering/sequential_file_write`

Ways to Share Data Between Processes

- Communicate by pipes
- Communicate by sockets
- Read/write to the same file/s
- Read/write to the same memory

Remember

```c
#include <sys/mman.h>
void * mmap(void *ADDR, size_t LENGTH, int PROT, int FLAGS, int FD, off_t OFFSET);
```

FD

- File handle of file to map
- -1 to not map any file

FLAGS for a file (OR together)

MAP_SHARED	All mappings synchronized, writes synchronize to file
MAP_SHARED_VALIDATE	Like MAP_SHARED but reject unknown FLAGS

Three Strategies

1. Processes map the same file on disk
2. First process maps no file (anonymous) and then forks (child processes inherit mapping)
3. Processes map the same file in memory (shared memory)

1 – Mapping Same File On Disk

- All process map file with MAP_SHARED
- Use msync with MAP_INVALIDATE to guarantee that changes are propagated to other process
- Not particularly efficient
- Wears down disk

2 – Anonymous Mapping and Then Fork

```c
#include <sys/mman.h>
void * mmap(void *ADDR, size_t LENGTH, int PROT, int FLAGS, int FD, off_t OFFSET);
```

FD

-1 to not map any file (anonymous mapping)

OFFSET

Zero

FLAGS for an anonymous mapping (OR together)

MAP_SHARED	All mappings synchronized
MAP_ANONYMOUS	Anonymous mapping (memory only, no file)

Use atomics and semaphores for synchronization!

3 – Map Shared Memory

- OS maintains a ramdisk for shared memory
- Processes used files in shared memory
- Work just like files except they are in memory (RAM and swap)
- Mappings don't require `msync` calls
- Use POSIX file calls are synchronized
- Use atomics or semaphores for synchronization after `mmap`

Open Shared Memory File

```c
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
int shm_open(const char *NAME, int OFLAG, mode_t MODE);
```

File NAME

Processes must agree on one

Creation OFLAG

read	O_RDONLY
read and write	O_RDWR
create if not exist	O_CREATE
must create	O_EXCL
truncate	O_TRUNC

Access MODE

Same as for open (standard POSIX permissions number value)

Close And Delete Shared Memory File

Close file – same as always

```
int err = close(fd);
```

Delete shared memory file

```
int shm_unlink(const char *NAME);
```

Memory Map Shared Memory File

- `mmap` the same as any other file
- `MAP_SHARED` required for actual passing data
- Is a shared array of bytes
- Each process might have a different starting address of the mapping
- For synchronization in the mapped area, use:
  - Atomic memory instructions
  - POSIX semaphores (type `man shm_overview` on Linux for more info)