Memory mapped files

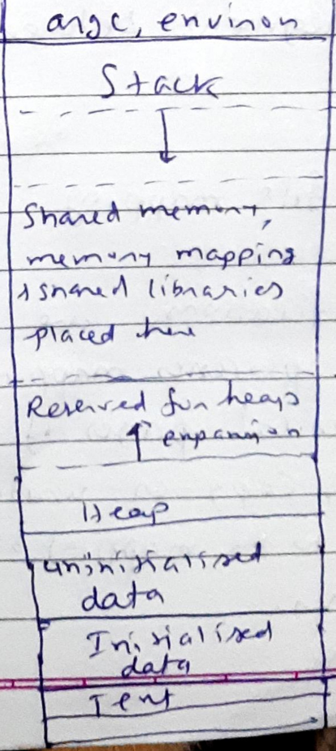
→ A memory mapped file is mostly a segment of virtual memory that has been assigned a direct byte-for-byte correlation with some portion of a file on disk.

→ memory mapped I/O let us map a file on disk into a buffer in process address space, so that when we fetch bytes from the buffer, the corresponding bytes of the file are read. Similarly, when we store data in the buffer, the corresponding bytes are automatically written to the file. This lets us perform I/O without read or write() system calls.

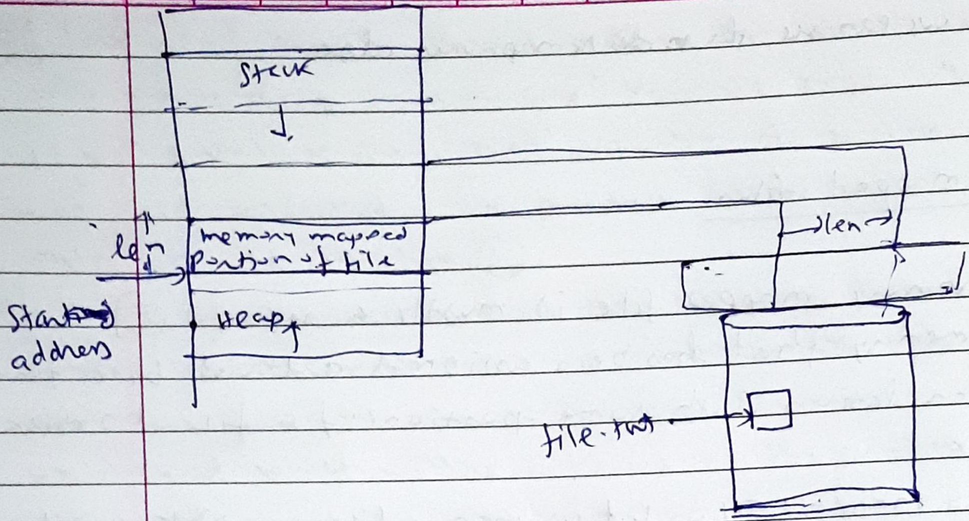
→ there are 2 types of memory mapped files:
- ⓐ persisted / File mapping :- data is saved in a file on disk
- ⓑ non-persisted / anonymous mapping

→ How a file on disk can be mapped to the address space of a process.

Let file.txt is a file on disk whose some portion we want to map in the address space of a process. Here, mmap() system call can be used for this. The call will return the starting address where the portion of the file is mapped.



argc, environ
Stack
Shared memory, memory mapping & shared libraries placed here
Reserved for heaps expansion
Heap
uninitialised data
Initialised data
Text



Shared file mapping

→ multiple processes mapping the same region of a file share the same physical pages of memory. Whenever a process tries to write, the modifications to the contents of the mapping are carried through to the file.

→ main use of shared ~~map~~ file mapping is:
- → memory-mapped I/O
- → IPC in a manner similar to shared memory segments between related or unrelated process.

Private file mapping

→ modifications are not visible to other processes. Multiple processes mapping the same file initially share the same physical pages of memory. Whenever a process tries to write, copy-on-write technique is employed, so that changes to the mapping by one process are invisible to other processes.

-) the main use of private file mapping is initializing a process's text & initialized data segments from the corresponding parts of a binary executable file or a shared library file.

mmap () system call
-) It is used to request the creation of memory mapping in the address space of the calling process. On success, it returns the starting address of the mapping.

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

-) the 1st argument addr argument indicates the virtual address at which the mapping is to be located. Preferably, we should give null, so that the kernel chooses a suitable address for the mapping that doesn't conflict with any existing mapping.

-) The 2nd argument len specifies the size of the mapping in bytes. ~~The ma~~ To map an entire file, we put len as size of the file. Normally, kernel creates mapping rounded up to the next multiple of page size.

-) The 3rd argument prot is a bit mask specifying the permissions (PROT_READ, PROT_WRITE, PROT_EXEC)

-) flags can be either MAP_PRIVATE or MAP_SHARED

-) 'fd' is the file descriptor of the file to be mapped.

-) offset specifies the starting point of the mapping in the file. To map entire file, we would specify offset as 0 & len as size of file.

-) The last two arguments are ignored for non persisted or anonymous mapping. we normally put -1 for 'fd' & a zero for offset for anonymous mapping.

msync () system call

int msync (void *addr, size_t len, int flag);

-) It causes the changes in part or all of the memory segment to be written back to (or read from) the mapped file

-) the 1st argument 'addr' is the address that is returned by the mmap () call

-) 'len' specifies the length of the mapping

-) 'flag' controls how ~~to~~ the update should be performed. It can have following 3 values:

Flag	Description
MS_ASYNC	Request update and returns immediately
MS_SYNC	Request update & waits for it to complete
MS_INVALIDATE	Invalidate other mappings of same file

int munmap (void *addr, size_t len)

-) this simply unmaps the memory mapped region pointed to by addr with length 'len'.

-) Normally, we unmap an entire mapping. Thus, we specify 'addr' as address returned by mmap() & specify the same length value as was used in the mmap() call.

-) The memory mapped region is automatically unmapped when a process terminates or performs an exec

-) closing the file 'fd' does not unmap the region.