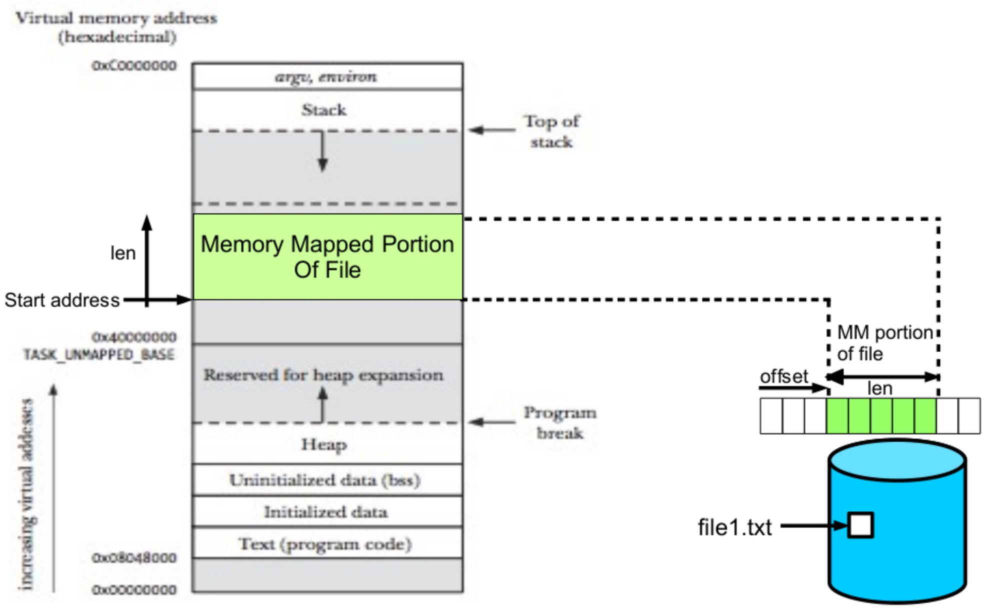
Memory Mapped Files

- 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 using `read()` or `write()` system calls
- There are two types of memory mapped files:
  - ➔ Persisted / File Mapping
  - ➔ Non-Persisted / Anonymous Mapping

Location of Memory Mappings in Virtual Memory



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 uses of shared file mapping is:
  - Memory-mapped I/O
  - IPC in a manner similar to System V shared memory segments between related or unrelated processes

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 and initialized data segments from the corresponding parts of a binary executable file or a shared library file

mmap() System Call

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

- The `mmap()` system call is used to request the creation of memory mappings in the address space of the calling process. On success, it returns the starting address of the mapping
- The first 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 second argument `len` specifies the size of the mapping in bytes. To map an entire file, we put `len` as size of the file. Normally, Kernel creates mappings rounded up to the next multiple of the page size
- The third argument `prot` is a bit mask specifying the permissions (`PROT READ, PROT WRITE, PROT EXEC`)

mmap() System Call (cont...)

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

- The fourth argument `flags` can be either `MAP_PRIVATE` or `MAP_SHARED` (as discussed)
- The fifth argument `fd` is a file descriptor identifying the file to be mapped
- The sixth argument `offset` specifies the starting point of the mapping in the file. To map the entire file, we would specify `offset` as 0 and `len` as the size of the file
- The last two arguments are ignored for non-persisted or anonymous mapping. We normally put -1 for `fd` and a zero for `offset` for anonymous mapping

msync() System Call

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

- The `msync()` function causes the changes in part or all of the memory segment to be written back to (or read from) the mapped file
- The first argument `addr` is the address that is returned by the mmap() call
- The second argument `len` specifies the length of mapping
- The third argument `flags` controls how the update should be performed. It can have following three values:

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

munmap() System Call

```
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 the address returned by a previous call to `mmap()`, and 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