

Memory Allocation

File Access

- File I/O
 - Calls to file I/O functions (e.g., `read()` and `write()`)
 - First copy data to a kernel's intermediary buffer
 - Then transfer data to the physical file or the process
 - Intermediary buffering is slow and expensive
- Alternative: Memory Mapping
 - Eliminate intermediary buffering
 - Significantly improve performance

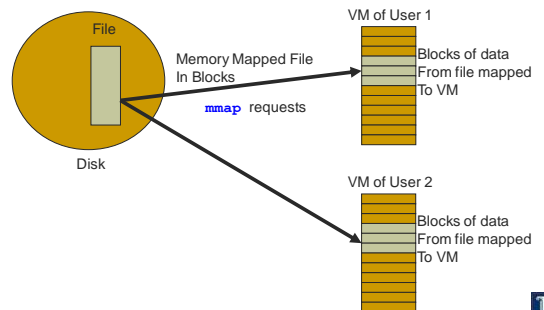


Memory Mapped Files

- Memory-mapped file I/O
 - Map a disk block to a page in memory
 - Allows file I/O to be treated as routine memory access
- Use
 - File is initially read using demand paging
 - A page-sized portion of the file is read from the file system into a physical page
 - Subsequent reads/writes to/from the file are treated as ordinary memory accesses



Memory Mapped Files



Memory Mapped Files: Benefits

- Treats file I/O like memory access rather than `read()`, `write()` system calls
 - Simplifies file access
- Several processes can map the same file
 - Allows pages in memory to be shared
- Dynamic loading
 - Map executable files and shared libraries into address space
 - Programs can load and unload executable code sections dynamically



Memory Mapped Files: Benefits

- Streamlining file access
 - Access a file mapped into a memory region via pointers
 - Same as accessing ordinary variables and objects
- Memory persistence
 - Enables processes to share memory sections that persist independently of the lifetime of a certain process



POSIX Memory Mapping

```
#include <sys/mman.h>
void *mmap(void *addr, size_t len, int prot,
           int flags, int fd, off_t off);
```

- Memory map a file
 - Establish a mapping between the address space of the process to the memory object represented by the file descriptor
- Parameters:
 - `addr`: the address to map the file into
 - `len`: the length of the data to map into memory
 - `prot`: the kind of access to the memory mapped region
 - `flags`: flags that can be set for the system call
 - `fd`: file descriptor
 - `off`: the offset in the file to start mapping from



POSIX Memory Mapping

```
#include <sys/mman.h>
void *mmap(void *addr, size_t len, int prot,
           int flags, int fd, off_t off);
```

- Memory map a file
 - Establish a mapping between the address space of the process to the memory object represented by the file descriptor
- Return value
 - On success, implementation-defined function of `addr` and `flags`
 - On failure, `MAP_FAILED`



Protection and Flags

- Protection Flags
 - `PROT_READ` Data can be read
 - `PROT_WRITE` Data can be written
 - `PROT_EXEC` Data can be executed
 - `PROT_NONE` Data cannot be accessed
- Flags
 - `MAP_SHARED` Changes are shared.
 - `MAP_PRIVATE` Changes are private.
 - `MAP_FIXED` Interpret `addr` exactly



mmap Example

- Map first 4kb of file and read int

```
#include <errno.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/types.h>
int main(int argc, char *argv[]) {
    int fd;
    void *pregion;
    if (fd = open(argv[1], O_RDONLY) < 0) {
        perror("failed on open");
        return -1;
    }
}
```



mmap Example

```
pregion = mmap(NULL, 4096, PROT_READ,
               MAP_SHARED, fd, 0);
if (pregion==(caddr_t)-1) {
    perror("mmap failed")
    return -1;
}
close(fd); /*close the physical file */
/*access mapped memory; read the first int in
the mapped file */
int val= *((int*) pregon);
}
```



munmap

```
#include <sys/mman.h>
int munmap(void *addr, size_t len);
```

- Remove a mapping
- Return value
 - 0 on success
 - -1 on error, sets `errno`
- Parameters:
 - `addr`: returned from `mmap()`
 - `len`: same as the `len` passed to `mmap()`



msync

```
#include <sys/mman.h>
int msync(void *addr, size_t len, int flags);
```

- Write all modified data to permanent storage locations
- Return value
 - 0 on success
 - 1 on error, sets `errno`
- Parameters:
 - `addr`: returned from `mmap()`
 - `len`: same as the `len` passed to `mmap()`
 - `flags`:
 - `MS_ASYNC` = Perform asynchronous writes
 - `MS_SYNC` = Perform synchronous writes
 - `MS_INVALIDATE` = Invalidate cached data



sysconf

```
#include <unistd.h>
long sysconf(int name);
```

- Determine the current value of a configurable system variable
- Return value
 - 0 on success
 - 1 on error, sets `errno`
- Parameters:
 - `name`: the system variable to be queried
 - `_SC_PAGESIZE`



More Examples

```
#include <errno.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/mman.h>
main(void) {
    size_t bytesWritten = 0;
    int fd;
    int PageSize;
    const char text = "This is a test";
```



More Examples

```
if ((PageSize = sysconf(_SC_PAGE_SIZE)) < 0) {
    perror("sysconf() Error=");
    return -1;
}
fd = open("/tmp/mmsyncTest", (O_CREAT | O_TRUNC |
    O_RDWR), (S_IRWXU | S_IRWXG | S_IRWXO));
if (fd < 0) {
    perror("open() error");
    return fd;
}
off_t lastoffset = lseek(fd, PageSize, SEEK_SET);
bytesWritten = write(fd, " ", 1);
if (bytesWritten != 1) {
    perror("write error. ");
    return -1;
}
```



[More Examples]

```
/* mmap the file. */
void *address;
int len;
off_t my_offset = 0;
len = PageSize;

/* Map one page */
address = mmap(NULL, len, PROT_WRITE, MAP_SHARED, fd,
              my_offset);

if (address == MAP_FAILED) {
    perror("mmap error. ");
    return -1;
}
```



[More Examples]

```
/* Move some data into the file using memory map. */
(void) strcpy((char*) address, text);

/* use msync to write changes to disk. */
if (msync(address, PageSize, MS_SYNC) < 0 ) {
    perror("msync failed with error:");
    return -1;
} else
    (void) printf("%s", "msync completed successfully.");

close(fd);
unlink("/tmp/msyncTest");
}
```



[Illegal Memory Access]

- Use signals!
 - **SIGSEGV** signal allows you to catch references to memory that have the wrong protection mode

