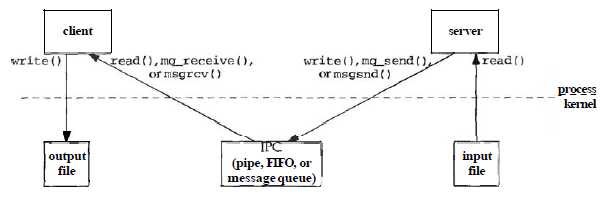
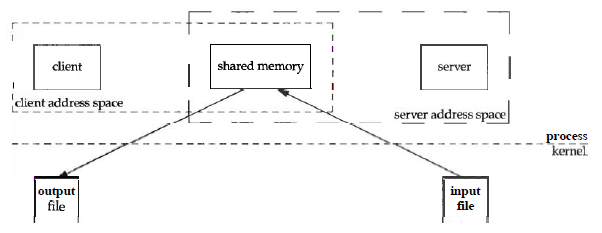
# Shared Memory

* Fastest form of IPC available
* Once the memory is mapped, no kernel involvement occurs in passing data between the processes
* Synchronization between the processes is required
* "no kernel involvement' means that the processes do not execute any system calls into the kernel to pass the data.
* But, the kernel must establish the memory mappings that allow the processes to share the memory, and then manage this memory over time (handle page faults, and the like).



**Flow of file data from server to client using message queues**



**Copying file data from server to client using shared memory**

# mmap()

**#include <sys/mman.h>**

**void \*mmap(void \**addr*, size\_t *length*, int *prot*, int *flags*, int *fd*, off\_t *offset*);**

**Returns:**

* On success, mmap() returns **a pointer to the mapped area**
* On error, the value **MAP\_FAILED** (that is, ***(void \*) -1***)

maps either a file or a POSIX shared memory object into the address space of a process

3 purposes of mmap:

1. to provide memory-mapped I/O with a regular file
2. to provide anonymous memory mappings with special files
3. to provide POSIX shared memory between unrelated processes with shm\_open

**addr**:

* starting address for the new mapping
* Starting address within the process of where the descriptor *fd* should be mapped
* Normally, this is specified as a null pointer, telling the kernel to choose the starting address

**len:**

* length of the mapping
* number of bytes to map into the address space of the process

**prot:**

* describes the desired memory protection of the mapping
* must not conflict with the open mode of the file

**PROT\_READ** Pages may be read

**PROT\_WRITE** Pages may be written

**PROT\_EXEC** Pages may be executed

**PROT\_NONE** Pages may not be accessed

**flags:**

* determines whether updates to the mapping are visible to other processes mapping the same region

**MAP\_SHARED** Changes are shared

**MAP\_PRIVATE** Changes are private

**MAP\_FIXED** Interpret the addr argument exactly

* Either the **MAP\_SHARED** or the **MAP\_PRIVATE** flag must be specified, optionally ORed with **MAP\_FIXED**
* **MAP\_PRIVATE** modifications to the mapped data by the calling process are visible only to that process and do not change the underlying object (either a file object or a shared memory object)
* **MAP\_SHARED** modifications to the mapped data by the calling process are visible to all processes that are sharing the object, and these changes do modify the underlying object
* **For portability**, **MAP\_FIXED** should not be specified and addr should be a null pointer

**fd:**

* descriptor to be mapped

**offset:**

* length from the beginning of the file from where mapping starts
* Normally, ***offset*** is 0
* After map returns success, the **fd argument can be closed. This has no effect on the** mapping that was established by mmap()
* **One way to share memory between a parent and child is to call map with MAP\_SHARED before calling fork**

**Why use mmap?**

* The nice feature in using a memory-mapped file is that all the I/O is done under the covers by the kernel, and
* we just write code that fetches and stores values in the memory-mapped region. We never call read, write, or lseek. Often, this can simplify our code.
* not all files can be memory mapped (ex: terminal or socket descriptor file)
* Another use of map is to provide shared memory between unrelated processes
* the return value from mmap can be different for each process that calls map for the same shared memory object

# munmap()

**#include <sys/mman.h>**

**int munmap(void \**addr*, size\_t *length*);**

**Return:** On success: 0, On failure: -1

**addr:** address that was returned by map, must be a multiple of the page size

**length:** the size of that mapped region

* To remove a mapping from the address space of the process
* Further references to these addresses result in the generation of a **SIGSEGV** signal to the process (invalid memory references)
* If the mapped region was mapped using **MAP-PRIVATE**, the changes made are discarded
* The region is also automatically unmapped when the process is terminated
* closing the file descriptor does not unmap the region
* It is not an error if the indicated range does not contain any mapped pages

# msync()

**#include <sys/mman.h>**

**int msync(void \**addr*, size\_t *length*, int *flags*);**

**Return:** On success: 0, On failure: -1

* synchronize a file with a memory map
* flushes changes made to the in-core copy of a file that was mapped into memory using mmap() back to the filesystem
* Without use of this call, there is no guarantee that changes are written back before munmap() is called
* **addr** and **length** normally refer to the entire memory-mapped region of memory
* **flags:** formed from the combination of constants

**MS\_ASYNC, MS\_SYNC and MS\_INVALIDATE**

**MS\_ASYNC**

Specifies that an update be scheduled, but the call returns immediately

returns once the write operations are queued by the kernel

**MS\_SYNC**

Requests an update and waits for it to complete

returns only after the write operations are complete

**MS\_INVALIDATE**

Asks to invalidate other mappings of the same file (so that they can be updated with the fresh values just written)

all in-memory copies of the file data that are inconsistent with the file data are invalidated

Subsequent references will obtain data from the file

# Notes:

* the fastest form of IPC available
* Once a file is memory mapped, read, write, or lseek is no longer used to access the file
* we just fetch or store the memory locations that have been mapped to the file by mmap()
* Changing explicit file I/O into fetches and stores of memory can often

simplify our programs and sometimes increase performance

* Use anonymous memory mapping instead of a regular file to map, When the memory is to be shared across a subsequent fork

MAP-ANON or mapping /dev/zero

4 additional functions defined by POSIX dealing with memory management

* **mlockall():** causes all of the memory of the process to be memory resident
* **munlockall():** undoes this locking
* **mlock():** causes a specified range of addresses of the process to be memory resident
* **munlock():** unlocks a specified region of memory

1. way to use map to provide shared memory between a parent and child (**related process**):

* using a memory-mapped file
* using 4.4BSD anonymous memory mapping
* using /dev/zero anonymous memory mapping

# POSIX Shared Memory

# Introduction

two ways to share memory between unrelated processes:

1. Memory-mapped files
2. Shared memory objects
3. **Memory-mapped files**

* a file is opened by open
* resulting descriptor is mapped into the address space of the process by map
* used to share memory between related and unrelated processes

1. **Shared Memory Objects**

* shm\_open() opens a Posix.1 IPC name (a pathname in the file system, it returns a descriptor

# System V Shared Memory

# Introduction

* similar in concept to POSIX shared memory
* we call shmget() followed by shmat()

For every shared memory segment, the kernel maintains the following structure:

**#include <sys/shm.h>**

**struct shmid\_ds** {

struct ipc\_perm shm\_perm;/\*operation permission

struct \*/

size\_t shm\_segsz ; /\* segment size \*/

pid\_t shm\_lpid; /\* pid of last operation \*/

pid\_t shm\_cpid; /\* creator pid \*/

shmatt\_t shm\_nattch; /\* current # attached \*/

shmat\_t shm\_cnattch; /\* in-core # attached \*/

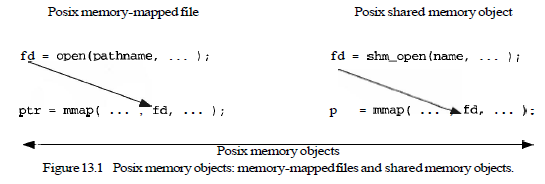
time\_t shm\_atime; /\* last attach time \*/

time\_t shm\_dtime; /\* last detach time \*/

time\_t shm\_ctime; /\* last change time of this structure \*/

};

* this descriptor is then mapped into the address space of the process by map



**struct ipc\_perm** {

key\_t \_\_key; /\* Key supplied to shmget(2) \*/

**uid\_t uid;** /\* Effective UID of owner \*/

**gid\_t gid;** /\* Effective GID of owner \*/

uid\_t cuid; /\* Effective UID of creator \*/

gid\_t cgid; /\* Effective GID of creator \*/

**unsigned short mode;**/\* Permissions + SHM\_DEST

and SHM\_LOCKED flags \*/

unsigned short \_\_seq; /\* Sequence number \*/

};

* contains the access permissions for the shared memory segment

# shm\_open() function

POSIX shared memory requires two-step process

1. **calling shm\_open()**

* specifying a name argument, to
* create a new shared memory object or
* open an existing shared memory object

1. **calling mmap()**

* to map the shared memory into the address space of the calling process
* reason for this two-step process, map already existed when POSIX invented its form of shared memory

#include <sys/mman.h>

#include <sys/stat.h> /\* for mode constants \*/

#include <fcntl.h> /\* for O\_\* constants \*/

**int shm\_open(const char \*name, int oflag, mode\_t mode);**

**Returns:**

success: a nonnegative file descriptor

failure: -1

# shmget() Function

* shmget - allocates a System V shared memory segment

#include <sys/ipc.h>

#include <sys/shm.h>

**int shmget(key\_t key, size\_t size, int shmflg);**

**Return**

success: a valid shared memory identifier

error: -1

* create a shared memory segment, or access an existing one
* create a new shared memory segment, with

size equal to the value of size rounded up to a multiple of PAGE\_SIZE, if key has the value IPC\_PRIVATE

if key isn't IPC\_PRIVATE, no shared memory segment corresponding to key exists

* size is fixed in System V shared memory object (in POSIX can be changed by ftruncate)
* reason that shm\_open() returns a descriptor is that an open descriptor is what map uses to map the memory object into the address space of the process
* shm\_open() creates a new, or opens an existing, POSIX shared memory object
* operation of shm\_open() is analogous to that of open(2)
* The returned file descriptor is guaranteed to be the lowest-numbered file descriptor **not previously opened within the process**
* **key** can be either a value returned by ftok() or the constant IPC\_PRIVATE
* **size** specifies the size of the segment, in bytes
* When a new shared memory segment is created,

a nonzero value for size must be specified

it is initialized to size bytes of 0

* If an existing shared memory segment is being referenced, size should be 0
* shmflg is a combination of the read-write permission values

This can be bitwise ORed with either IPC\_CREAT or IPC\_CREAT | IPC\_EXCL

**IPC\_CREAT** Create a new segment.

If this flag is not used, then shmget() will find the segment associated with key and check to see if the user has permission to access the segment

* The **FD\_CLOEXEC** flag (see fcntl(2)) is set for the file descriptor
* After a call to mmap(2) the file descriptor may be closed without affecting the memory mapping
* name argument used with shm\_open() is then used by any other processes that want to share this memory
* **For portable use**, a shared memory object should be identified by a name of the form "/somename"

a null-terminated string of up to NAME\_MAX (i.e., 255)

an initial slash, followed by one or more characters, none of which are slashes

**IPC\_EXCL** This flag is used with IPC\_CREAT to ensure if the segment already exists, the call fails

oflag is a bit mask created by ORing together exactly one of O\_RDONLY or O\_RDWR and any of the other flags listed following:

**O\_RDONLY:** A shared memory object opened in this way can be mmap(2)ed only for read (PROT\_READ) access

**O\_RDWR:** Open the object for read-write access

**O\_CREAT:** Create the shared memory object if it does not exist

**O\_EXCL:** check for the existence of the object, and its creation if it does not exist, are performed atomically

**O\_TRUNC:** If the shared memory object already exists, truncate it to 0 bytes

# shm\_unlink() function

#include <sys/mman.h>

**int shm\_unlink(const char \*name);**

**Returns:**

success: 0

failure: -1

* shm\_unlink() is analogous to unlink(2)
* shm\_unlink() function performs the converse operation, removing an object previously created by shm\_open()
* once all processes have unmapped the object, It de-allocates and destroys the contents of the associated memory region
* After a successful shm\_unlink(), attempts to shm\_open() an object with the same name will fail
* unlinking a name has no effect on existing references to the underlying object,

until all references to that object are closed

# shmat() function

#include <sys/types.h>

#include <sys/shm.h>

**void \*shmat(int shmid, const void \*shmaddr, int shmflg);**

**Returns:**

On success: starting address of the attached shared memory

on error: (void \*) -1 is returned

* attaches shared memory segment to our address space
* rules for determining this address are as follows:
* If shmaddr is a

**null pointer**, the system selects the address for the caller (recommended and portable)

**nonnull pointer**, the returned address depends on SHM\_RND value for the shmflg argument

**SHM\_RND is not specified**, the shared memory segment is attached at the address specified by the shmaddr argument

* Unlinking a name just prevents any subsequent call to open, mcopen, or sem\_open from succeeding

**SHM\_RND is specified**, the shared memory segment is attached at the address specified by the shmaddr argument, rounded down by the constant SHMLBA

LBA - "Lower Boundary Address"

* By default, the shared memory segment is attached for both reading and writing

if the process has read-write permissions for the segment

* The SHM\_RDONLY value can also be specified in the flag argument, specifying read-only access

# ftruncate() function

#include <unistd.h>

**int ftruncate(int *fd*, off\_t *length*);**

**Returns:**

success: 0

failure: -1

* to change the size of either a regular file or a shared memory object
* definition slightly differently for regular files vs. shared memory objects

**Regular file:**

* file size > length, the extra data is lost
* file size < length, whether the file is changed or its size is increased is unspecified (// BOOK)

it is extended, and the extended part reads as null bytes ('\0') (// man page)

* The file offset is not changed
* If the size changed, then the st\_ctime and st\_mtime fields for the file are updated,

and the set-user-ID and set-group-ID permission bits may be cleared.

# shmdt() function

#include <sys/types.h>

#include <sys/shm.h>

**int shmdt(const void \*shmaddr);**

**Returns**: success: 0 failure: -1

* detaches the shared memory segment located at the address shmaddr
* to-be-detached segment must be currently attached with shmaddr

* does not delete the shared memory segment
* When a process terminates, all shared memory segments currently attached by the process are detached
* On a successful shmdt() call, the system updates the members of the shmid\_ds structure

**shm\_dtime** is set to the current time

**shm\_lpid** is set to the process-ID of the calling process

* the file must be open for writing
* **portable way** to extend the size of the file to length bytes is to **lseek** to offset

**Shared memory object:**

* ftruncate() sets the size of the object to length
* nothing is said in the description of ftruncate about the new contents of a shared memory object that is extended
* If a newly extended piece of shared memory is not initialized to some value (contents are left as is), this could be a security hole

**shm\_nattch** is decremented by one. If it becomes 0 and the segment is marked for deletion, the segment is deleted

# fstat() function

#include <sys/types.h>

#include <sys/stat.h> **int fstat(int *fd*, struct stat \**buf*);**

**Returns:**

success: 0

failure: -1

* get file status
* No permissions are required on the file itself
* In struct stat only 4 contain information when fd refers to a shared memory object (highlighted)

**struct stat**{

dev\_t st\_dev; /\*ID of device containing file \*/

ino\_t st\_ino; /\*inode number \*/

**mode\_t st\_mode; /\*protection \*/**

nlink\_t st\_nlink; /\*number of hard links \*/

**uid\_t st\_uid; /\*user ID of owner \*/**

**gid\_t st\_gid; /\*group ID of owner \*/**

dev\_t st\_rdev; /\*device ID (if special file) \*/

**off\_t st\_size; /\*total size, in bytes \*/**

blksize\_t st\_blksize;/\*blocksize for file system I/O \*/

blkcnt\_t st\_blocks;

/\*number of 512B blocks allocated \*/

time\_t st\_atime; /\*time of last access\*/

time\_t st\_mtime; /\*time of last modification\*/

time\_t st\_ctime; /\*time of last status change\*/

};

# shmctl() function

#include <sys/ipc.h>

#include <sys/shm.h>

**int shmctl(int shmid, int cmd, struct shmid\_ds \*buf);**

**Returns:**

A successful **IPC\_INFO or SHM\_INFO** operation returns

the index of the highest used entry in the kernel's internal array recording information about all shared memory segments

A successful **SHM\_STAT** operation returns

the identifier of the shared memory segment whose index was given in shmid

**Other operations**

on success: 0 on error: -1

* shmctl - System V shared memory control
* performs the control operation specified by cmd on the System V shared memory segment with shmid
* buf argument is a pointer to a shmid\_ds structure

Example:

Simple program

lncrementing a Shared Counter

* Valid values for cmd are:

IPC\_STAT, IPC\_SET, IPC\_RMID, IPC\_INFO

SHM\_INFO, SHM\_STAT, SHM\_LOCK, SHM\_UNLOCK

**IPC\_STAT**

* copy information from the kernel data structure associated with shmid

into the shmid\_ds structure pointed to by buf.

* The caller must have read permission on the shared memory segment.

**IPC\_SET**

* Write the values of some members of the shmid\_ds structure pointed to by buf

to the kernel data structure associated with this shared memory segment

shm\_perm.uid, shm\_perm.gid, and shm\_perm.mode (9 LSB bits)

* shm\_ctime value is also updated with the current time.

effective UID of the calling process must match the owner (shm\_perm.uid) or creator (shm\_perm.cuid)

or the caller must be privileged

**IPC\_RMID**

* Remove the shared memory segment identified by shmid from the system and destroy it
* segment will actually be destroyed only after the last process detaches it (i.e., when the shm\_nattch is 0)

Example:

Simple Program