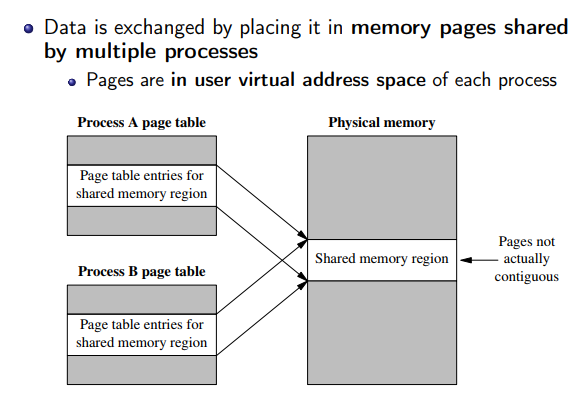
**SHARED MEMORY (POSIX)**

Shared memory allows two or more processes to share the same region (usually referred to as a segment) of physical memory. Since a shared memory segment becomes part of a process’s user-space memory, no kernel intervention is required for IPC. All that is required is that one process copies data into the shared memory; that data is immediately available to all other processes sharing the same segment. This provides fast IPC by comparison with techniques such as pipes or message queues, where the sending process copies data from a buffer in user space into kernel memory and the receiving process copies in the reverse direction. (Each process also incurs the overhead of a system call to perform the copy operation.) . Here is the figure taken from http://man7.org/training/download/posix\_shm\_slides.pdf



To use a POSIX shared memory object, we perform three steps :

 1.      Use the shm\_open() function to open an object with a specified name. The shm\_open() function is analogous to the open()system call. It either creates a new shared memory object or opens an existing object.  shm\_open() returns a file descriptor referring to the object.

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

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

#include <sys/mman.h>

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

Returns file descriptor on success, or –1 on error.  The name argument identifies the shared memory object to be created or opened.

One of the purposes of the oflag argument is to determine whether we are opening an existing shared memory object or creating and opening a new object. If oflag doesn’t include O\_CREAT, we are opening an existing object. If O\_CREAT is specified, then the object is created if it doesn’t already exist.  The oflag argument also indicates the kind of access that the calling process will make to the shared memory object, by specifying exactly one of the values O\_RDONLY or O\_RDWR.

2. ftruncate(): set size of the shared memory object.

2.  Memory Mapping: Mapping a portion of a file to the virtual address space is called memory mapping. This is done by passing the file descriptor created by the previous function shm\_open to mmap. Once a file is mapped, its contents can be seen by other processes depending on if the mapping is private or shared. As name suggests, private is not shared with other processes while shared means it is shared with others.

The function is described as

*#include <sys/mman.h>*

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

*Returns starting address of mapping on success, or MAP\_FAILED on error*

The addr argument indicates the virtual address at which the mapping is to be

located. If we specify addr as NULL, the kernel chooses a suitable address for the

mapping.

Though the length argument specifies the size of the mapping in bytes, the kernel creates mappings in units of page size

The prot argument is a bit mask specifying the protection to be placed on the

mapping. It can have anyone of these values

PROT\_NONE The region may not be accessed

PROT\_READ The contents of the region can be read

PROT\_WRITE The contents of the region can be modified

PROT\_EXEC The contents of the region can be executed

The flags argument is a bit mask of options controlling various aspects of the mapping operation. Exactly one of the following values must be included in this mask: MAP\_PRIVATE or MAP\_SHARED

The fd argument is a file descriptor identifying the file to be mapped.

The offset argument specifies the starting point of the mapping in the file, and must be a multiple of the system page size. To map the entire file, we would specify offset as 0 and length as the size of the file.

A sample code snippet is :

 addr = mmap(NULL, length, PROT\_READ | PROT\_WRITE, MAP\_PRIVATE, fd, 0) ;

if (addr == MAP\_FAILED)

errExit("mmap") ;

**Removing Shared Memory Objects:**

When a shared memory object is no longer required, it should be removed using shm\_unlink().

#include <sys/mman.h>

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

The shm\_unlink() function removes the shared memory object specified by name. Removing a shared memory object doesn’t affect existing mappings of the object (which will remain in effect until the corresponding processes call munmap() or terminate), but prevents further shm\_open() calls from opening the object. Once all processes have unmapped the object, the object is removed, and its contents are lost.

**How to unmap shared memory** :

The munmap() system call performs the converse of mmap(), removing a mapping

from the calling process’s virtual address space.

#include <sys/mman.h>

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

Returns 0 on success, or –1 on error

The addr argument is the starting address of the address range to be unmapped and length argument is a nonnegative integer specifying the size (in bytes) of the region to be unmapped.

Commonly, 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. Here’s an example:

/\* Code for working with mapped region \*/

if (munmap(addr, length) == -1)

errExit("munmap") ;