National University of Computer and Emerging Sciences



Laboratory Manual

for

Operating Systems Lab

(CS 205)

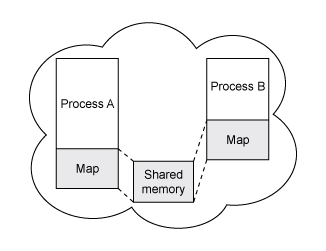
|  |  |
| --- | --- |
| Course Instructor | Mr. Ibrahim Nadir  Dr. Usman Awais |
| Lab Instructor(s) | Mr. Asad Ullah  Mr. Usman Khan |
| Section | CS-A  CS-B |
| Semester | FALL 2019 |

Department of Computer Science

FAST-NU, Lahore, Pakistan

**Objectives:**

**-**Shared Memory through POSIX API



Problem with pipes: requires mode shift.

-Shared memory quickest means of IPC.

API to shared memory.

**System-V vs POSIX**

|  |  |
| --- | --- |
| Shared Memory Interface Calls shmget(), shmat(), shmdt(), shmctl() | Shared Memory Interface Calls shm\_open(), mmap(), shm\_unlink() |

* You use shm\_open to open the shared memory object instead of calling [open(2)](https://docs.oracle.com/cd/E23824_01/html/821-1463/index.html).
* You use shm\_unlink to close and delete the object instead of calling [close(2)](https://docs.oracle.com/cd/E23824_01/html/821-1463/index.html) which does not remove the object

**How a file on a disk can be mapped in the address space of a process (use mmap system call) no need to use read and write system call.**

**POSIX shared memory** is an inter-process communication (IPC) mechanism defined in POSIX specification. After setting up the shared memory, two (or more) processes may read from and write to the shared memory region. Compared to other IPC mechanisms (e.g. pipe, socket, etc), POSIX shared memory does not impose copy overheads, thus it is appealing to some applications.

A program using POSIX shared memory usually consists of these steps:

1. Create or open a *shared memory object* with shm\_open(). A file descriptor will be returned if shm\_open() creates a shared memory object successfully.
2. Set the shared memory object size with ftruncate().
3. Map the shared memory object into the current address space with mmap() and MAP\_SHARED.
4. Read/write the shared memory.
5. Unmap the shared memory with munmap().
6. Close the shared memory object with close().
7. Delete the shared memory object with shm\_unlink().

The pattern for shared memory is to create a segment with shm\_open(), size it with write() or ftruncate(), map it into process memory with mmap(), and do the work required with one or more additional participants. To finish, the originating process calls munmap() and shm\_unlink(), and then exits.

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

The first argument name must start with a slash / character and continue with several non-slash characters. The second argument flag can be the combination of O\_RDONLY, O\_RDWR, O\_CREAT, and/or O\_EXCL.

* O\_RDONLY stands for *read-only*. If a program opens a shared memory object with O\_RDONLY, it can only read the shared memory and must not write to the shared memory.
* O\_RDWR stands for *read* and *write*. If a program opens a shared memory object with O\_RDWR, it can read from or write to the shared memory.
* O\_CREAT stands for *create*. If the shared memory object does not exist, a new shared memory object will be created. Conversely, if O\_CREAT is not set and the shared memory object does not exist, an error will be returned.
* O\_EXCL stands for *exclusive*. This must be set with O\_CREAT. If the shared memory object does not exist, a new shared memory object will be created. If the shared memory object exists, an error will be returned.

The third argument mode is the file permission of the created shared memroy object. If O\_CREAT is not specified or the shared memory object exists, then mode is ignored.

Shared memory objects are kernel persistent. Unless they are deleted, they are kept until the computer reboots. shm\_unlink() is the shared memory object anology of unlink():

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

The first argument name is the name of the shared memory object which you would like to delete.

**shm\_unlink**

#include <sys/mman.h>

#include <sys/stat.h>

#include <fcntl.h>

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

shm\_unlink removes the previously created POSIX shared memory object. The *name* is the name of the shared memory object as described under shm\_open, above.

**ftruncate**

#include <unistd.h>

#include <sys/types.h>

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

The ftruncate system call makes the object referred to by the file descriptor, *fd*, of size *length* bytes. When a POSIX shared memory is created, it is of size zero bytes. Using ftruncate, we can make the POSIX shared memory object of size *length* bytes. ftruncate returns zero on success. In case of error, ftruncate returns -1 and errno is set to the cause of the error.

**mmap**

#include <sys/mman.h>

void \*mmap (void \*addr, size\_t length, int prot, int flags,

int fd, off\_t offset);

With the mmap system call, we can map a POSIX shared memory object to the calling process's virtual address space. *addr* specifies the address at which it should be mapped. In most cases, we do not care at what address mapping is done and a value of NULL for *addr* should suffice. *length* is the length of shared memory object that should be mapped. To keep things simple, we will map the whole object and length for us will be the length of the shared memory object. *prot* can have the values, PROT\_EXEC, PROT\_READ, PROT\_WRITE and PROT\_NONE. PROT\_EXEC means that the mapped pages may be executed and PROT\_NONE means that the mapped pages may not be accessed. These two values do not make sense for a shared memory object. So we will use PROT\_READ | PROT\_WRITE value for *prot*. There are many *flags* but the only one meaningful for shared memory is MAP\_SHARED, which means that the updates to the mapped shared memory are visible to all other processes immediately. *fd* is, of course, the file descriptor for the shared memory received from an earlier shm\_open call. *offset* is the location in the shared memory object at which the mapping starts; we will use the value zero for offset and map the shared memory object starting right from the beginning. On success, mmap returns the pointer to the location where the shared memory object has been mapped. In case of error, MAP\_FAILED, which is, (void \*) -1, is returned and errno is set to the cause of the error.

**munmap**

#include <sys/mman.h>

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

munmap unmapps the shared memory object at location pointed by *addr* and having size, *length*. On success, munmap returns 0. In case of error, munmap returns -1 and errno is set to the cause of the error.

$ gcc -o sender sender.c -lrt

After running the above code, a shared memory object is created. On Linux, it can be found under /dev/shm:

$ ls -l /dev/shm/

**InLab Questions:**

1. Create a shared memory such that one process writes an array of integers of size 5 taken from the user and the second process read from the shared memory and display the sum and average the integers.
2. Create a memory mapped file for a given file lab.txt. You are then required to create a child process that will count the number of words and will write it at the end of the file (lab.txt).

The parent process will count the number of sentences and will write it at the end of the file. (lab.txt).