Shared Memory

Learning Outcomes

At the end of this session, the students should be able to:

1. Explain the steps needed to be done to create, share, and delete shared memories amongst processes.
2. Implement matrix multiplication using two processes cooperating via Unix System V shared-memory IPC mechanism

Content

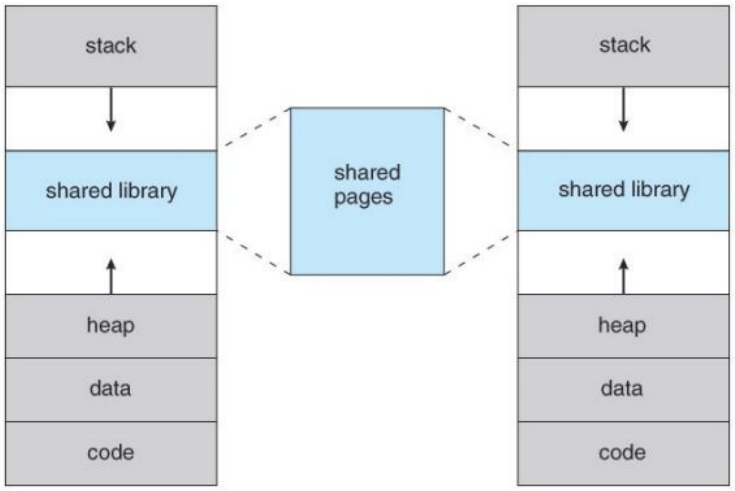
Required Packages

1. Ubuntu 16.04
2. gcc 5.x or higher
3. bash

Shared Memory

Recall that, a thread has access to all data (global variables) in the process where the thread is running. For multithreaded process, this means that all these threads have access to the data as well. Thus, sharing of information between threads is easy to implement.

However, processes has their own ***address space*** and the operating system is strict in enforcing security and protection such that, any one process cannot just access a region they don't own or is not publicly accessible. Shared access to the data amongst thread is allowed since the thread is still part of the process, and thus, it poses no security risk. But that is not the case for processes.



Thus, in order to share data amongst processes, one process should explicitly create a shared variable in the shared-memory area. The shared-memory area then becomes part (gets mapped with) of the process' address space. In effect, all other processes may also make the shared memory part of their address space. As a result, these processes can now read and write data on this shared-memory area (as shown in the figure on the right).

Unix System V Shared-Memory

The Unix System V shared-memory IPC mechanism provides a set of system calls to support the read and write of data stored in the shared memory. Aside from the V shared-memory, there are also other shared-memory IPC mechanism being supported in Linux.

Steps in using a shared-memory:

1. **Allocate/Create a shared-memory segment**

   ```
   int shmget(key_t key, size_t size, int shmflg);
   ```

   where,

   `key` is the key for the shared-memory segment,
   `size` is the size in bytes of the requested shared memory, and
   `shmflg` specifies the access permissions and creation control flags.

   Returns an integer which identifies the id of the shared memory segment.
   Note: You need to pass `IPC_CREAT` on the shmflg if you want the process to create its own segment. If `IPC_CREAT` is not passed, shmget will look for a segmented identified by the given `key`.

2. **Attach the shared-memory segment to the address space of a process**

   ```
   void *shmat(int shmid, const void *shmaddr, int mode);
   ```

   where,

   `shmid` is the value returned by shmet,
   `shmaddr` is the address where the process will be attached, and
   `mode` specifies how the shared segment will be used

   Returns a pointer for locating where the segment has been mapped.
   Note: if NULL is passed on `shmaddr`, the system will choose a suitable, unused address to attach the segment.

3. **After completion, detach the shared-memory segment**

   ```
   int shmdt(const void* addr);
   ```

   where,

   `addr` is the address where the shared-memory segment is mapped.

   Returns integer indicating the status of the detachment.

4. **Unallocate the shared memory**

   ```
   int shmctl(int shmid, int cmd, struct shmid_ds *buf);
   ```

   where,

   `shmid` is the segment id,
   `cmd` is the command that must be executed (in this case, `IPC_RMID`), and
   `buf` is the segment data structure (`NULL`).

The use of shared memory is more advantageous over message-passing since accessing data using system calls entails no overhead, and in turn provides faster communication. Synchronization primitives, like semaphores, can be used if needed for concurrent access of data on the shared-memory.

Learning Experiences

Students will be given sample codes for demonstration.

Assessment Tool

A programming exercise for implementing matrix multiplication utilizing the Unix System V shared-memory.