Operating System Lab-8

Q1> Consider a system with one parent process and two child processes A and B. There is a shared signed integer X initialized to 0. Process A increments X by one, 100000 times in a for loop. Process B decrements X by one, 100000 times in a for loop. After both A and B finish, the parent process prints the final value of X.

Hint: Declare a shared memory variable to hold X (using system calls shmget(), shmat(), shmdt(), and shmctl()). Write the programs for processes A and B. Do not put any synchronization code in the code for A and B. You should write the code in such a way so that you can simulate race condition in your program by slowing down A or B appropriately by using sleep() calls at appropriate points. Note that if there is no race condition, the value of X finally should be 0. Simulating race conditions means that if you run the program a few times, sometimes the final value of X printed by your program should be non-zero.

Desired Output:

```
vboxuser@blockchainexplorer:~$ ./race
Final value of X: 0
vboxuser@blockchainexplorer:~$ ./race
Final value of X: -703
vboxuser@blockchainexplorer:~$ ./race
Final value of X: -8
vboxuser@blockchainexplorer:~$ ./race
Final value of X: -3
vboxuser@blockchainexplorer:~$ ./race
Final value of X: 7
vboxuser@blockchainexplorer:~$
```

Q2> Add synchronization code based on semaphores to process A and B in question 1 so that there is no possibility of race conditions. Use the calls semget(), semop(), semctl() to create and manage semaphores.

Desired Output:

```
vboxuser@blockchainexplorer:~$ ./race2
Final value of X: 0
vboxuser@blockchainexplorer:~$ ./race2
Final value of X: 0
vboxuser@blockchainexplorer:~$ ./race2
Final value of X: 0
vboxuser@blockchainexplorer:~$ ./race2
Final value of X: 0
vboxuser@blockchainexplorer:~$ ./race2
Final value of X: 0
vboxuser@blockchainexplorer:~$
```

Note:

The important headers to be included are:

- **<sys/types.h>:** Contains various data types and symbolic constants used in system calls. This header file is often included to ensure portability of code between different Unix-like systems.
- **<sys/ipc.h>:** Provides definitions for interprocess communication data structures. It includes structures and constants used with functions like ftok() for generating unique keys for IPC objects.
- **<sys/shm.h>:** Contains definitions for shared memory operations. Shared memory is a way for processes to share data by mapping a common memory area into their address spaces. Functions like shmget(), shmat(), and shmdt() are used for shared memory operations.
- **<sys/sem.h>:** Contains definitions for semaphores, which are synchronization mechanisms used for process coordination. Functions like semget(), semop(), and semctl() are used to work with semaphores.
- **<sys/wait.h>:** Provides functions and constants related to process management, specifically waiting for child processes to terminate. Functions like wait(), waitpid(), and macros like WEXITSTATUS are commonly used for process control and synchronization.

The system calls to be used for performing the above tasks are as follows: -

- **shmget():** shmget() is a system call in Unix-like operating systems (including Linux) that is used for working with shared memory segments. Shared memory is a form of inter-process communication (IPC) that allows multiple processes to share a common memory region. shmget() is used for creating, accessing, or manipulating shared memory segments. The syntax for the shmget() function is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

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

where,

key: This is an identifier for the shared memory segment. It can be generated using the ftok function, which converts a pathname and a project identifier into a key. The same key is used to identify and access the shared memory segment across different processes.

size: This parameter specifies the size (in bytes) of the shared memory segment you want to create or access.

shmflg: This is a set of flags that control the behavior of the shmget function.

- **shmat():** The shmat() function is used in Unix-like operating systems (including Linux) for attaching a shared memory segment to the address space of a process. shmat() is used to establish a connection between a process and a shared memory segment, allowing the process to read from and write to that shared memory. The syntax for shmat() function is as follows:-

```
#include <sys/types.h>
#include <sys/shm.h>

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

where,

shmid: This is the identifier of the shared memory segment, typically obtained using shmget(). It identifies the specific shared memory segment you want to attach to.

shmaddr: This parameter specifies the desired address at which you want to attach the shared memory segment in the process's address space. If you pass NULL, the system chooses a suitable address for attachment.

shmflg: This is a set of flags that control the behavior of the shmat() function.

The return value of shmat() is a pointer to the attached shared memory segment in the process's address space. If the attachment is unsuccessful, shmat() returns (void*) -1, and you can check the specific error condition by examining the errno variable.

- **shmdt():** The shmdt() function is used in Unix-like operating systems (including Linux) for detaching a process from a previously attached shared memory segment. When a process is done using the shared memory, it should detach from it using shmdt() to release the associated resources. The syntax for shmdt() function is as follows:

```
#include <sys/types.h>
#include <sys/shm.h>

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

where,

shmaddr: This is a pointer to the address where the shared memory segment is attached in the process's address space. You provide the same pointer that was returned by the shmat() function when you originally attached to the shared memory segment. The return value of shmdt() is an integer. It returns 0 if the operation is successful (i.e., the process successfully detaches from the shared memory segment). If there's an error, it returns -1, and you can check the specific error condition by examining the errno variable.

- **shmctl():** The shmctl() function is used in Unix-like operating systems (including Linux) for controlling and managing shared memory segments. Shared memory is a form of inter-process communication (IPC) that allows multiple processes to share a common memory region. shmctl() provides a means to perform various operations on shared memory segments, including creating, deleting, and modifying their properties. The syntax for shmctl function is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

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

where,

shmid: This is the identifier of the shared memory segment you want to perform operations on. You typically obtain this identifier using shmget() when creating or accessing a shared memory segment.

cmd: This parameter specifies the command or operation you want to perform on the shared memory segment. The possible values for cmd are defined as constants, including:

IPC_STAT: Retrieve information about the shared memory segment (fills buf with the information).

IPC_SET: Set information and permissions for the shared memory segment based on the values in buf.

IPC_RMID: Remove (delete) the shared memory segment.

buf: This is a pointer to a structure of type struct shmid_ds where information about the shared memory segment is stored or from which information is read, depending on the value of cmd. The structure may contain data such as segment size, permissions, user-defined data, etc.