# 15. POSIX IPC

SOA, Deemed to be University
ITER, Bhubanewar

# Book(s)

## Text Book(s)

**Kay A. Robbins, & Steve Robbins**

**Unix™ Systems Programming**

**Communications, concurrency, and Treads**

**Pearson Education**

## Reference Book(s)

**Brain W. Kernighan, & Rob Pike**

**The Unix Programming Environment**

**PHI**

# Introduction

- The POSIX:XSI Extension standardize the classical UNIX interprocess communication (IPC) mechanisms **shared memory**, **message queues**, and **semaphore sets** respectively.
- These mechanisms allow unrelated processes to exchange information in a reasonably efficient way, use a key to identify, create or access the corresponding entity.
- The entities may persist in the system beyond the lifetime of the process that creates them.
- POSIX:XSI also provides shell commands, (`ipcs`) to list and remove them.

# POSIX:XSI Interprocess Communication

- The POSIX interprocess communication (IPC) is part of the POSIX:XSI Extension and has its origin in UNIX System V interprocess communication.

- Types of IPC mechanisms
  - **shared memory**
  - **message queues**
  - **semaphore sets**

- They provides mechanisms for sharing information among processes on the same system.

# POSIX:XSI Shared Memory

- Shared memory allows processes to read and write from the same memory segment.Header file: **#include<sys/shm.h>**.
- The kernel maintains a structure, **shmid_ds** with the following members for each shared memory segment:

```
struct shmid_ds {
struct ipc_perm shm_perm; /*operation permission
                             structure */
size_t shm_segsz;          /* size of segment in bytes */
pid_t shm_lpid;          /*process ID of last operation */
pid_t shm_cpid;            /*process ID of creator */
shmatt_t shm_nattch;     /* number of current attaches */
time_t shm_atime;          /* time of last shmat */
time_t shm_dtime;          /* time of last shmdt */
time_t shm_ctime;          /* time of last shctl */
```

# Permission Structure

IPC associates an **ipc_perm** structure with each IPC structure. This structure defines the permissions and owner and includes the following members:

```
struct ipc_perm {
    uid_t          cuid;      /* creator user ID */
    gid_t          cgid;      /* creator group ID */
    uid_t          uid;       /* owner user ID */
    gid_t          gid;       /* owner group ID */
    unsigned short mode;      /* r/w permissions */
};
```

# Creating/Accessing a Shared Memory Segment

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

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

```
Returns:

(1) If successful, shmget returns a nonnegative integer
    corresponding to the shared memory segment identifier.

(2) If unsuccessful, shmget returns -1 and sets errno.
```

# `key` Generation

**`key`** can be selected one of the three ways:

**`IPC_PRIVATE`:** System select

**Pick a key directly:** Select a random kay

**Using `ftok()`:** System generate using the function **`ftok()`**. **`ftok`** convert a pathname and a project identifier to a System V IPC key

```
#include <sys/types.h>
#include <sys/ipc.h>
key_t ftok(const char *pathname, int proj_id);


(1) On success, the generated key_t value is returned.
(1) On failure -1 is returned.
```

# The value of `shmflag`

The value of **`shmflag`** is composed of:

**`IPC_CREAT`:** to 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.

**`IPC_EXCL`:** used with **`IPC_CREAT`** to ensure failure if the segment exists.

**`mode_flags`:** (lowest 9 bits) specifying the permissions granted to the owner, group, and other.

# Attaching a shared memory segment

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

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

```
Returns:

(1) If successful, shmat returns the starting address of
    the segment.

(2) If unsuccessful, shmat returns -1 [ i.e (void *) -1 ]
    and sets errno
```

- **shmat()** attaches the shared memory segment identified by **shmid** to the address space of the calling process. It can be done in three ways.
- If **shmaddr** is NULL, the system chooses a suitable (unused) address at which to attach the segment.
- By default the shared memory segment is attached for both reading and writing by the calling process, if the process has read-write permissions for the shared memory segment. So, set the **shmflg** to 0.

# The value of `shmflag` in `shmat()` OPTIONAL

The value of **`shmflag`** in **`shmat`** is composed of:

**`0`:** By default the shared memory segment is attached for both reading and writing by the calling process, if the process has read-write permissions for the shared memory segment.

**`SHM_RDONLY`:** Attach the segment for read-only access.

**`SHM_RND`:** If **`shmaddr`** isn't `NULL` and **`SHM_RND`** is specified in **`shmflg`**, the attach occurs at the address equal to **`shmaddr`** rounded down to the nearest multiple of **`SHMLBA`**(Segment low boundary address multiple).

**`SHM_EXEC`:** Allow the contents of the segment to be executed. The caller must have execute permission on the segment.

**`SHM_REMAP`:** This flag specifies that the mapping of the segment should replace any existing mapping in the range starting at **`shmaddr`** and continuing for the size of the segment.

# Detaching a shared memory segment

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

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

```
Returns:

(1) If successful, shmdt returns 0.

(2) If unsuccessful, shmdt returns -1 and sets errno.
```

- The **shmaddr** parameter is the starting address of the shared memory segment.
- **shmdt()** detaches the shared memory segment located at the address specified by **shmaddr** from the address space of the calling process.
- The last process to detach the segment should deallocate the shared memory segment by calling **shmctl**.

# Controlling Shared Memory

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

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

```
Returns:

(1) If successful, shmctl returns 0.

(2) If unsuccessful, shmctl returns -1 and sets errno.
```

- The **shmctl** function provides a variety of control operations on the shared memory segment **shmid** as specified by the **cmd** parameter.

- The interpretation of the **buf** parameter depends on the value of **cmd**.

# POSIX:XSI Values of `cmd` for `shmctl`

| **cmd** | **Description** |
|---|---|
| IPC_RMID | remove shared memory segment **shmid** and destroy corresponding **shmid_ds** |
| IPC_SET | set values of fields for shared memory segment **shmid** from values found in **buf** |
| IPC_STAT | copy current values for shared memory segment **shmid** into **buf** |

# Shared Memory Writer

**Program 1:** shmwriter.c

```
#include<stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include<sys/shm.h>
int main()
{
 int id,*var;
 key_t key;
 key=ftok("key.txt",65);
 id=shmget(key,50,0664|IPC_CREAT);
 printf("Shared memory Identifier=%d\n",id);
 var=(int *)shmat(id, NULL,0);
 *var=50;
 shmdt(var);
 return 0;
}
```

# Shared Memory Reader

**Program 2:** shmreader.c

```
#include<stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include<sys/shm.h>
int main()
{
 int id,*rvar;
 key_t key;
 key=ftok("key.txt",65);
 id=shmget(key,50,0664);
 printf("Shared memory Identifier=%d\n",id);
 rvar=(int *)shmat(id, NULL,SHM_R);
 printf("Value in shared memory=%d\n",*rvar);
 shmdt(rvar);
 shmctl(id,IPC_RMID,NULL);
 return 0;
}
```

# Shared Memory after `fork()`, `exec()`, & `exit()`
# OPTIONAL

**`fork()`:** After **`fork`** the child inherits the attached shared memory segments.

**`exec()`:** After an **`exec`** all attached shared memory segments are detached (not destroyed).

**`exit()`:** After **`exit`** all shared memory segments are detached (not destroyed).

# Shared Memory in Related Processes

```
int main(){
  pid_t pid;int shmid,*shvar;
  key_t key=ftok(".",45);
  shmid=shmget(key,20,0664|IPC_CREAT);
  printf("Key=%x ........Shmid=%d\n",key,shmid);
  shvar=shmat(shmid,NULL,0);
  printf("Default initial value of shvar=%d\n",*shvar);
  *shvar=10;
  pid=fork();
  if(pid==0){
      *shvar=*shvar+90;
      printf("child update=%d\n",*shvar);
      exit(0);
  }
  else{
       wait(NULL);
       *shvar=*shvar+110;
       printf("parent updates=%d\n",*shvar);
 }
 return 0;
}
```

# Modification to the above Code

Replace the statement **`wait(NULL);`** after the line (**`printf()`** in the parent part and check the output. State the reason for such output.

```
int main(){
  pid_t pid;int shmid,*shvar;key_t key=ftok(".",45);
  shmid=shmget(key,20,0664|IPC_CREAT);
  printf("Key=%x ........Shmid=%d\n",key,shmid);
  shvar=shmat(shmid,NULL,0);
  printf("Default initial value of shvar=%d\n",*shvar);
  *shvar=10;pid=fork();
  if(pid==0){
       *shvar=*shvar+90;
       printf("child update=%d\n",*shvar);
       exit(0);
  }
  else{
        *shvar=*shvar+110;
        printf("parent updates=%d\n",*shvar);
        wait(NULL);
 }
 return 0;
}
```

# System Limits for Shared Memory OPTIONAL

**SHMMAX:** Maximum size in bytes for a shared memory segment.

```
$ cat /proc/sys/kernel/shmmax
```

**SHMMIN:** Minimum size in bytes for a shared memory segment: implementation dependent (currently 1 byte, though PAGE_SIZE is the effective minimum size).

**SHMMNI:** System wide maximum number of shared memory segments:

```
$ cat /proc/sys/kernel/shmmni
```

**SHMALL:** System wide limit on the total amount of shared memory, measured in units of the system page size.

```
$ cat /proc/sys/kernel/shmall
```

# Race Condition and Critical Section

```
do {
    Entry section

    Critical section

    Exit section

    Remainder section

} while(true);
```

# Peterson's Solution to Critical Section

do {

```
flag[i] = true ;
turn = j;
while (flag[j] && turn == j);
```

Critical section

```
flag[i] = false;
```

Remainder section

} while(true);

# Peterson's Solution for Two Processes