System V inter-process communication (IPC), shared memory and semaphores

MEKENTICHI NEJEMEDDINE

December 2025

I. Inter-Process Communication (IPC):

We will use “shared resource, resource” to refer to shared memory and semaphores.

Inter-process communication is possible through “sys/ipc.h”, this header offers a set of macros that allow implementing shared resources as follows:

- IPC_CREAT: this macro used when creating a shared memory segment, or a semaphore, it allows the creation, while allowing duplicate keys.
- IPC_EXCL: this macro is used alongside IPC_CREAT as follows: IPC_EXCL | IPC_CREAT and it does not allow duplicate IDs, i.e: the creation happens only if the specified key does not exist.
- IPC_RMID: used with ***ctl to delete a specified resource, via its ID.
- IPC_NOWAIT: this macro is used quite often with as a flag when calling semop. When sem_op = -1, i.e semop will perform test (P, decrement the value of the semaphore). If the process will be blocked after P, the operation fails and returns -1 instead of blocking, and one can find errno = EAGAIN.
- IPC_PRIVATE: this key can be assigned to a resource to become private and accessible only by the process that created it.

key_t ftok(const char *filepath, int project_num):

this function is used to generate a key associated with the public resource we want to create such as a semaphore or a shared memory. It returns a key_t type (basically an integer).

It takes a file path as a first argument, and a project number as a second argument. Both can be anything as the function works as a regular key generator, one thing! The file must exist and its read flag is set for the user calling this function. Otherwise, it fails and returns -1. project_num is an int, usually between 0 and 255 to help generating a new key.

ftok() fails if the file doesn’t exist or isn’t accessible.

It’s only convenient to generate reproducible keys between processes.

II. Shared Memory:

In this section, we will discuss the process of creation of a shared memory segment and how to use it.

- **One must first include the header "sys/shm.h"**
- **Shared memory segment creation:**

int shmget(key_t key, size_t _size, int shmflg):

this function, returns the id og the shared memory sgment we created. It requires:

- key: the key of the shared memory, generated by ftok(), or manually assigned.
- _size: the size of the shared memory, for example, sizeof(int) to get a 4 bytes shared memory on Linux.
- shmflg: flags, of the shared memory, usually we use: IPC_CREAT | 0666; in C, preceding a number with a 0, makes it octal, then 0666 is 110-110-110 in binary which sets the read and write flags of the shared memory for all users (owner, group, other).
  Note that we can also do IPC_CREAT | IPC_EXCL | 0666 if we don't want duplicate keys
- **Shared memory usage:**
  To use a shared memory segment, one must first attach it to the virtual memory addressing of the process using void *shmat(int _shmid, void *addr, int _shmflg):
  - This returns a pointer to the shared memory segment.
  - Requires:
    - shmid: the id of the shared memory segment
    - addr: the address where we want the memory to be attached. Put NULL to let the kernel find a suitable free address, which is much more recommended.
    - _shmflg: specifies how we want to attach that memory, for this we have some flags:
      - SHM_RDONLY: attach in read only.
      - 0: attach for read and write.
      - SHM_RND: round down to a multiple of SHMLBA (don't bother yourself with this), used only if you provide an address upon creation.

  Accessing the attached memory is straight forward through the provided pointer from shmat.

  When done using the shared memory, it should be detached using shmdt(void *_shmaddr) where _shmaddr is the address provided by shmat.

  To delete a shared memory, you can use shmctl(int _shmid, int _cmd, struct shmid_ds * _buf):

- Here we have:
    - _shmid: is the Id of the shared memory.
    - __cmd: is the command we want to execute, usually IPC_RMID to remove it.
    - _buf: this buffer is used to set or get the metadata of the shm, it is not NULL only if the _cmd is IPC_STAT or IPC_SET.

Example of creating, using and deleting a shm:

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdlib.h>

int main() {
    key_t key = ftok(SOME_FILE_PATH, 21);
    if(key == -1)
        return 1;
    size_t shmzise = sizeof(int);
    int shmid = shmget(key, shmsize, IPC_CREAT | 0666);
    if(shmid == -1)
        return 2;
        // here we  d o n t  care much,
        // but we can check errno to see
        // why shmid failed.
    // in another process, or even in the same process.
    int *addr = (int *)shmat(shmid, NULL, 0);
    // or void * then cast later to int *
    // shmat returns (void *)-1 when it fails and not NULL
    if(addr == (int *)-1)
        return 3;
    // now we can use our shared  m e m o r y
    addr[0] = 127;
    if(shmdt((void *)addr) == -1)
        return 4;
        // now we can safely remove our shared memory.
        // any attempt to access addr from now on will cause
        // a  SEG FAULT
    If(shmctl(shmid, IPC_RMID, NULL) == -1) // returns 0 when
    // success
        Return 5; // now the shm is deleted.
    return 0;
    // wherever I returned directly, you can check errno
    // to see the reason of failure.
}
```

III. Semaphores:

The use of semaphores is pretty similar to the shared memory in terms of creation and deletion.

First of all, one must include “sys/sem.h”.

Then, to create a semaphore, we should follow these steps:

- Generate a key for the semaphore.
- Set some number of semaphores you want to use.

Then, we call int semget(key_t key, int semnum, int _semflg) where:

- key: the key associated to the semaphore we want.
- semnum: number of sempahores we want.
- _semflg: flags that define access rights. IPC_CREAT | 0666 as an example.

This will return the semid (id of the semaphore) which we will use later.

Now, we can set the value of a semaphore as follows:

union semun init = {

.arg = some value, say 1

}

And then we call semctl(int semid, int semnum, int _cmd, …) where:

- semid: the id of the semaphore.
- semnum: the index of the semaphore we want to set (since semget creates an array of semaphores, it's 0-indexed). Use 0 when you have one semaphore.
- _cmd: the commend we want:
  - SETVAL: when we want to set the value of one single semaphore.
  - SETALL, when we want to set values of all semaphores we have. In this case, we use that unsigned short *array in our semun, and we set semnum to the length of the array.
- semctl is variadic, meaning that after this we can put anyting, we usually use that semun union and pass it to this function. An example will be provided later.

Now, if we want to perform semaphores operations, we can do the following:

Struct sembuf operation = {

.semnum // index of the wanted semaphore

.sem_op = +1 to increment (equivalent to V primitive)

-1 to decrement (equivalent to P primitive)

0 to wait until the value of the semaphore is 0

.sem_flg = 0 // default behaviour

SEM_UNDO // undo the operation if the process crashes

IPC_NOWAIT // if the operation will get the process blocked, semop will fail and set errno to EAGAIN instead of getting blocked.

}.

Then we can call int semop(int semid, sembuf *_buff, size_t nsops) where:

- semid: the id of the semaphore.
- _buff, is the address of the buffer where we specified the operations to do
- nsops is the number of operations included, simply, the length of buff.

To get the value of a semaphore, we can call semget(semid,index, GETVAL)

To remove a semaphore, we call semctl(semid,semnum,IPC_RMID) where:

- semid: is the id of the semaphore.
- semnum: ignored
- IPC_RMID: the cmd used to remove the specified ID.

Here is a simple example:

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <errno.h>

int main() {
    key_t key = ftok("SOME_FILE_PATH", 20);
    if (key == -1)
        return 1;

    int semnum = 1;
    int sem_init_val = 1; // used for mutexes
    int semid = semget(key, semnum, IPC_CREAT | 0666);
    if (semid == -1)
        return 2;

    union semun u;
    u.val = sem_init_val;
    if (semctl(semid, 0, SETVAL, u) == -1)
        return 3;

    struct sembuf P_mutex = {0, -1, 0}; // P operation
    struct sembuf V_mutex = {0, +1, 0}; // V operation

    // Before entering critical section
    if (semop(semid, &P_mutex, 1) == -1) {
        perror("semop P_mutex failed");
        return 4;
    }

    /* Critical section */
    printf("In critical section\n");

    // After leaving critical section
    if (semop(semid, &V_mutex, 1) == -1) {
        perror("semop V_mutex failed");
        return 5;
    }

    // Remove the semaphore
    if (semctl(semid, 0, IPC_RMID) == -1) {
        perror("semctl IPC_RMID failed");
        return 6;
    }

    return 0;
}
```

Here, are some of the return values of semctl based on macros used:

Macro/Command	Purpose	Return Value on Success	Notes / Arg Used
SETVAL	Set a single semaphore’s value	0	Uses arg.val
SETALL	Set all semaphores in a set	0	Uses arg.array
IPC_RMID	Mark semaphore set for deletion	0	semnum ignored
IPC_SET	Modify metadata (permissions, UID/GID)	0	Uses arg.buf
GETVAL	Get value of a single semaphore	Current value (≥0)	semnum specifies index
GETALL	Get values of all semaphores in the set	0	Values stored in arg.array
GETPID	Get PID of last process that performed operation	PID of process	semnum specifies index
GETNCNT	Number of processes waiting for sem > 0	Count (≥0)	semnum specifies index
GETZCNT	Number of processes waiting for sem = 0	Count (≥0)	semnum specifies index
IPC_STAT	Retrieve semaphore set info	0	Stored in arg.buf