IPC

Two processes will use shared memory to communicate and some mechanism for synchronise their actions. This is necessary because shared memory does not come with any synchronisation tools: if you can access it at all, you can access it anytime regardless of the other process.

Semaphores were chosen for synchronisation (out of several options).

Note that there is a much simpler way to implement a solution to the producer–consumer (with confirmations) problem: two message queues.

References

- Beej's guide to shared memory
- Beej's guide to semaphores
- Linux guide (sketchy)
- Marshall's guide to shared memory
- Marshall's guide to semaphores
- Another attempt to explain shared memory

IPC identifier

Each **IPC** object is identified by two labels:

key: which is an external identifier. All the processes that want a given object must produce the one and only key that allows access to it.

identifier: once a key was accepted, a process will refer to an **IPC** object by an internal id (an object will have a different id in each process using it).

A key identifies uniquely an object in a particular **IPC** domain but can be used simultaneously in several domains.

There are two ways of getting a key:

Invent one

Pick any number such as 4561111 and use it. As long as nobody picks the same number, you are fine (the number above should be avoided: it is the telephone of the White House).

Get one from the system

A special system call ftok() exists solely to give you a unique key based on two arguments:

```
key = ftok( char * , char ) ;
```

where the first argument is the path of any existing (and accessible) file in the system and the second argument is a number between 0 and 255 (every character has this property). domains.

Acquiring some shared memory

You already have your favourite key and you want a shared memory area of `size` bytes with access rights shmflg. A call like this will create one:

```
if ((shmid = shmget (key, size, shmflg )) == −1) {
    perror("shmget failed");
    exit(−1);
}
```

The access rights could be: 0666 (anybody can do anything) plus create or fail:

```
    shmflg = 0666 | IPC_CREAT | IPC_EXCL ;
```

It could also be: 0640 (the owner can do anything, group members may read) plus create or fail:

```
    shmflg = 0640 | IPC_CREAT | IPC_EXCL ;
```

shmget() got you a shared memory identifier but it is not a pointer to a location in memory and thus is useless in itself. What you need is a pointer that you can use to access the memory area:

```
shmptr = shmat( shmid , mychoice , 0 ) ;
if( shmptr == (char *) -1 ) {
    perror( "shmat" ) ;
    exit( -1 ) ;
}
```

The second argument mychoice is almost always 0; if not, it asks that the value returned by shmat be equal to this argument, if possible. The last argument gives access rights again.

Now you can access the shared memory using the pointer provided by shmat().

Here is some nonsensical code copied from Marshall:

```
main()
{
  char c;
  int shmid;
  key_t key = 5678 ;
  char *shm, *s;
  if ((shmid = shmget(key, 27, IPC_CREAT | 0666)) < 0) ...
  if ((shm = shmat(shmid, 0, 0)) == (char *) -1) ...
  s = shm;
  for (c = 'a'; c <= 'z'; *s++ = c++) ;
  *s = NULL;

  while (*shm != '*')
    sleep(1);
}
```

Creating a semaphore

You cannot get one semaphore; you must ask for an array of them. The code below asks for `nsems` semaphores.

```
if ((semid = semget(key, nsems, semflg)) == −1) {
    perror("semget failed");
    exit(−1);
}
```

You give a magic key which is the external name (“public name”) of your semaphore cluster and you provide flags that indicate what access permission you are willing to grant to users of this cluster. The standard permission is `0600` (I can read and write and nobody else can).

The returned value is an identifier (not a pointer). A semaphore is a tightly controlled entity and you cannot simply access it directly as in:

```
semid[2] = 0 ;
```

This will not compile. If you want to set the third semaphore of the cluster identified by `semid` to 0, you must use the system call semctl() which has most obscure semantics.

Acquiring a semaphore (again)

The key is the public name of the semaphore cluster you want; the same name will be used by all the processes that will share this semaphore cluster.

Consider using the inode of the current directory (if you are there, it is accessible) to create 2 semaphores with the requirement that they must be brand new:

```
key = ftok( "." , 'Q' ) ;
semid = semget( key , 2 , 0600 | IPC_CREAT | IPC_EXCL ) ;
if( semid == −1) {
   perror( "semget refused" );
   kill( getpid() , SIGINT ) ;
}
```

The ’Q’ argument is an integer between 0 and 255 (as required).

Clean after your code

```
void delete( int sig )
{
    printf( "Cleaning\n" ) ;
    shmctl( shmid , IPC_RMID , 0 ) ;
    semctl( semid , IPC_RMID , 0 ) ;
    exit( 0 ) ;
}

void start( )
{
    signal( SIGINT , delete) ;
    ... ...
```

Semaphore operations

Two operations are of real interest:

semctl() allows to manipulate the values of semaphores.

semop() provides the basic $\mathcal{P}$ and $\mathcal{V}$ operations on a semaphore. You can use semop() to define your own operations (such as a non–blocking Poll()).

Basic use of semaphores

```
P( semaphore ) ;
... modify the shared memory ...
V( semaphore ) ;
```

If the semaphore is properly initialised (to 1), the $\mathcal{P}$ operation will block any process that wants to touch the shared memory when another process is doing so.

```
void V( int s )
{
    struct sembuf  S ;

    S.sem_num = s ;
    S.sem_op = 1 ;
    if( semop( semid , &S , 1 ) == −1 ) {
        perror( "V failed" ) ;
        kill( getpid() , SIGINT ) ;
    }
}
```

P

```
void P( int s )
{
    struct sembuf  S ;

    S.sem_num = s ;
    S.sem_op = −1 ;
    while( semop( semid , &S , 1 ) == −1 ) {
        perror( "P failed" ) ;
        sleep( 1 ) ;
    }
}
```

This code assumes that **semop()** failed due to an interrupted system call ordue to a race condition (this is the purpose of the **sleep()**).

It will not work if there is asynchronisation error; in such case, it will loop forever.

semun

The system call **semctl** requires a union type called **semun**.

Some systems have it defined in bits/sem.h (called inside sys/sem.h); other systems require that you define it yourself.

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


#ifdef _SEM_SEMUN_UNDEFINED
union semun {
    int val ;
    struct semid_ds *buf ;
    unsigned short *array ;
} ;
#endif // _SEM_SEMUN_UNDEFINED
```

Consult the file /usr/include/bits/sem.h for details.

Two non–standard operations on semaphores: a non–blocking **Poll()** and an initialisation function **I()**.

```
int Poll( int s )
{
   return semctl( semid , s , GETVAL ) ;
}

void I( int s )
{
   union semun arg ;
   arg.val = 1 ;   // sets it to 1
   if( semctl( semid , s , SETVAL , arg ) == −1 )
      perror( "semctl" ) ;
}
```

```
struct VID {
    int code ;  // the current state of this entry
    // = 0 empty    (sem 0 == 1)
    // = 1 vid inside   (sem 1 == 0)
    // = 2 confirmed vid    (sem 1 == 0)
    pid_t pid ; // Validator's pid
    int vid ;   // the vid to be recorded
    int confirmation ;  // passed back from Tallier
} ;
```

```
fh = fopen( "Booth pid pid" , "w" ) ;
if( fh == NULL )
    perror( "Could not open the pid file" ) ;
fprintf( fh , "%d\n" , getpid() ) ;
fclose( fh ) ;

key = ftok( "/dev/null" , 'B' ) ;
if( (key_t) key == −1 )
    perror( "ftok" ) ;
shmid = shmget( key , ...
shmptr = (struct VID *)shmat( shmid , 0 , 0 ) ;

semid = semget( key , 2 , 0600 | IPC_CREAT | IPC_EXCL
) ;
```