



System V IPCs


Inter Process Communication

Informática II
R2004 - 2020

¿Qué es un proceso? (Repaso de Info I)

Un proceso es una INSTANCIA de un programa en ejecución. Esto implica que un proceso está “haciendo algo” (ejecutando un programa), y que a su vez tiene un bloque de memoria dispuesto para este fin, además de un número de proceso (PID), y un conjunto de variables que permiten al SO identificarlo y controlarlo (detener su ejecución, reanudarla, etc.).

Un mismo programa puede ejecutarse más de una vez, por lo que generará más de un proceso. A su vez, durante la ejecución de un programa, podemos generar más de un proceso para hacer varias cosas “a la vez” (Ya veremos que la ejecución no se desarrolla exactamente al mismo tiempo, pero para el usuario pareciera que si).




¿Cómo se ejecutan procesos en paralelo?

Para “bifurcar” mi proceso (generar un nuevo proceso, o un proceso “hijo”) puedo utilizar la función ***fork()***. Esta función duplica el proceso (genera un nuevo bloque de memoria para el proceso hijo, y COPIA TODAS LAS VARIABLES que se venían usando, y devuelve un valor distinto para los dos procesos generados.

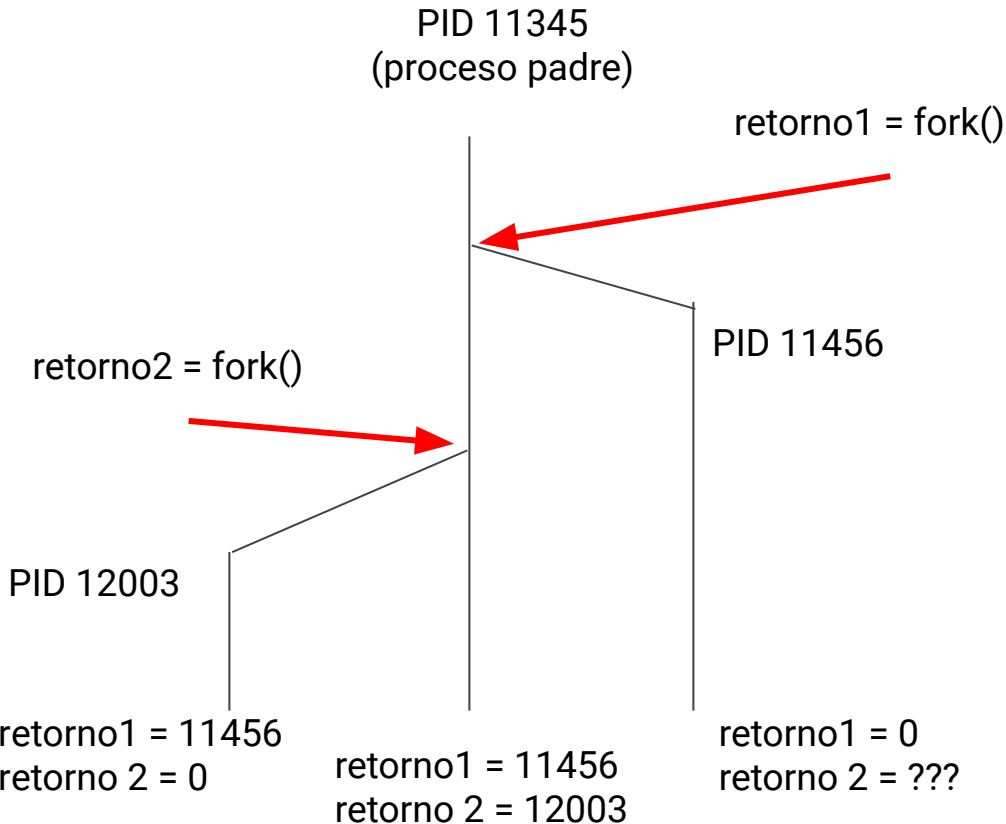
Al proceso padre le devuelve el valor del proceso hijo (el PID)

Al proceso hijo le devuelve 0.

De esta manera, podemos saber que proceso es el que se está ejecutando chequeando el valor de retorno de `fork()`



Fork() gráficamente



```
int main (void)
{
    short retorno1, retorno2;

    ...

    retorno1 = fork();

    if ( !retorno1 ) {
        //Proceso Hijo1
    }
    else {
        //Proceso Padre

        ...

        retorno2 = fork();
        if (!retorno2) {
            //Proceso Hijo2
            ...
        }
        else{
            //Proceso Padre
            ...
        }
    }
}
```

Comunicación entre procesos: Inter Process Communication (IPCs)

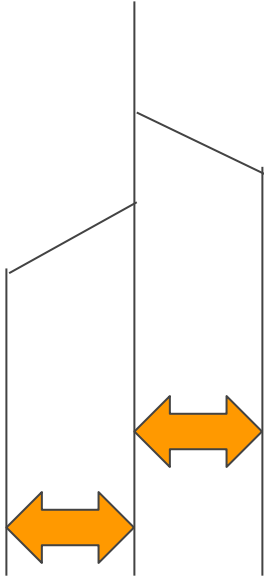
¿Cómo podemos comunicar los procesos que corren en paralelo?

Existen diferentes métodos, entre los cuales están los IPCs System V.

Estos son un conjunto de dispositivos integrados en el Kernel de Linux (o sea, son **funciones que nos da el SO**), que están diseñados para que 2 o más procesos puedan comunicarse y compartir información entre sí.

Existen 3 tipos de IPCs System V:

- Colas de mensajes (message queues)
- Memoria compartida (shared memory)
- Semáforos (semaphores)



¿Cómo lo hacíamos hasta ahora? Pipes

En informática I ya vimos algunos mecanismos de comunicación entre procesos

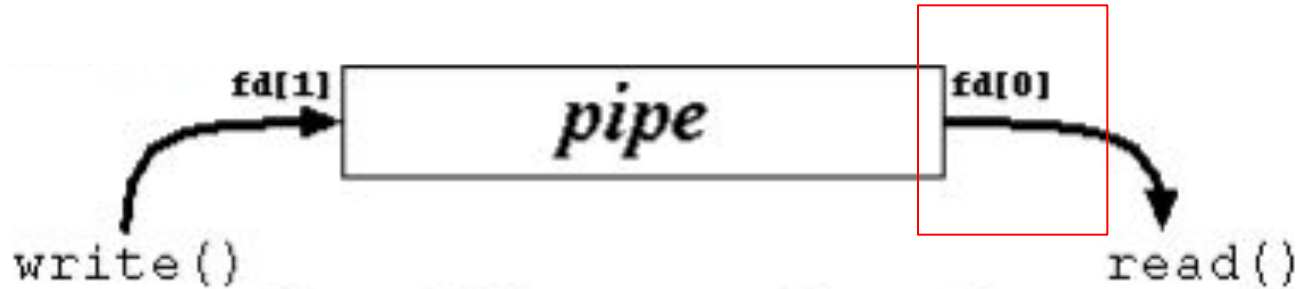


Figura 1. Cómo se organiza un pipe.

Ejemplo de pipes - Repaso Info I

```
/******  
 *           MAIN           *  
******/  
int main()  
{  
    int fd[2];  
    int pid_child;  
    char buf[BUF_LEN] = {0};  
    if (pipe (fd) == -1)  
    {  
        perror ("Error pipe");  
        return RET_MAIN_ERROR_PIPE;  
    }  
}
```

La función pipe crea 2 fd (file descriptors). Uno de ellos se usa para lectura (fd[0]), y el otro se usa para escritura (fd[1]). De esa manera, luego de llamar a pipe(), todo lo que yo escriba en fd[0] mediante la función write(), aparecerá en el fd[1]

Ejemplo de pipes - Repaso Info I

```
pid_child = fork ();

if (pid_child == -1)
{
    perror ("error fork");

    return RET_MAIN_ERROR_FORK;
}
else if (pid_child == 0)
{
    // soy el hijo
    // El hijo solo lee

    close (fd[1]);

    while (strcmp(buf, LAST_MSG) != 0)
    {
        read(fd[0], buf, BUF_LEN);
        printf("El hijo recibe: %s\n", buf);
    }

    printf("Comunicacion terminada\n");

    close (fd[0]);
}
```

```
else
{
    // soy el padre
    // El padre solo escribe

    close (fd[0]);

    printf("El padre escribe: ");
    scanf("%s", buf);
    write(fd[1], buf, strlen(buf) + 1);

    while (strcmp(buf, LAST_MSG) != 0)
    {
        sleep(1);
        printf("El padre escribe: ");
        scanf("%s", buf);
        write(fd[1], buf, strlen(buf) + 1);
    }

    wait (NULL);

    close (fd[1]);
}
```


Named pipes - Repaso Info I

Una segundo método para comunicar 2 procesos son los named pipes. A diferencia de los pipes, un named pipe hace referencia a un NODO del filesystem (esto es, a un archivo cualquiera). De esta manera, puedo comunicar procesos que no tengan relación entre sí (no necesariamente deben ser padres e hijos).

Un named pipe se crea mediante la función `mknod()`, y uno de los argumentos que debo enviarle es el nombre del nodo (archivo) al que voy a hacer referencia (si no existe, lo crea).

Luego mediante sucesivos llamados a las funciones `write` y `read`, puedo escribir y leer desde 2 procesos cualesquiera. Los named pipes son FIFOs (First In, First Out), lo que implica que voy a ir leyendo la información en el mismo orden en el que la fui escribiendo, y una vez leída la información desaparece del nodo.

Ejemplo Named Pipes - Repaso Info I

```
else if (pid_child == 0)
{
    // soy el hijo
    // El hijo solo escribe

    fd = open(NAMED_FIFO_NAME, O_WRONLY);

    if (fd == -1)
    {
        perror ("error open");

        return RET_MAIN_ERROR_OPEN_FIFO;
    }

    while (strcmp(buf, LAST_MSG) != 0)
    {
        read(fd, buf, BUF_LEN);
        printf("El padre recibe: %s\n", buf);
    }

    printf("Comunicacion terminada\n");

    close (fd);
}
```

```
else
{
    // soy el padre
    // El padre solo lee

    fd = open(NAMED_FIFO_NAME, O_RDONLY);

    if (fd == -1)
    {
        perror ("error open");

        return RET_MAIN_ERROR_OPEN_FIFO;
    }

    printf("El hijo escribe: ");
    scanf("%s", buf);
    write(fd, buf, strlen(buf) + 1);

    while (strcmp(buf, LAST_MSG) != 0)
    {
        sleep(1);
        printf("El hijo escribe: ");
        scanf("%s", buf);
        write(fd, buf, strlen(buf) + 1);
    }

    wait (NULL);

    close (fd);
}
```

Ejemplo Named Pipes - Repaso Info I

```


/*****
 *          MAIN
 *****/
int main()
{
    int fd;
    int pid_child;
    char buf[BUF_LEN] = {0};

    if (mkfifo(NAMED_FIFO_NAME, 0600) == -1 && errno != EEXIST)
    {
        perror ("Error: mkfifo");

        return RET_MAIN_ERROR_MKFIFO;
    }
}

```


Si no existe, lo creo



¿Por qué no son suficientes?

Los System V IPCs traen varias ventajas con respecto a los pipes que veníamos usando. En primer lugar son más seguros, utilizan recursos que no dependen del proceso que los crea, permitiendo que varios procesos puedan acceder al mismo recurso en forma simultánea.

Por otro lado, cada IPC posee una interfaz (un grupo de funciones) y algunas características (estructuras asociadas) que permiten darle una mayor funcionalidad y características más diversas que los pipes, dándoles asimismo una mayor robustez.



¿Para qué se utilizan los distintos IPCs?

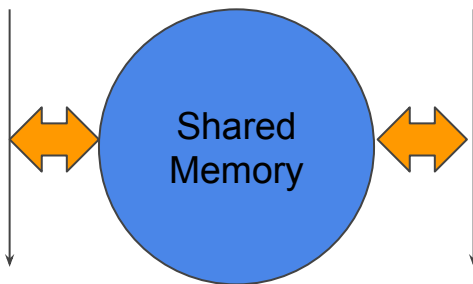
Message queues

- Sirven para enviar mensajes (cadenas, números, etc.) entre procesos.
(Similares a los named pipes)



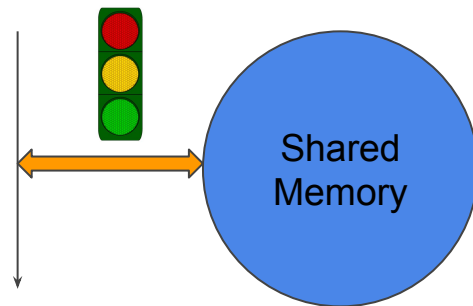
Shared memory

- Es un área de memoria compartida entre varios procesos (buffer)



Semáforos

- Sirven controlar el acceso a información compartida entre varios procesos.



¿Cómo funcionan los IPCs System V?

Todos los IPCs funcionan de una manera similar:

- En primer lugar, deben **crear una llave** (key) compartida, para identificar unívocamente el IPC en cuestión (porque puede haber muchos procesos usando muchos IPCs al mismo tiempo)
- Ambos procesos deben **conectarse** al IPC, utilizando la llave generada.
- Una vez conectados al IPC, los procesos pueden **utilizar** los IPCs o **configurarlos**.



Generación de la llave:

La llave es una variable de tipo **key_t**. Para asegurar que sea unívoca (y no intentar acceder a otro IPC en uso), se suele usar la función `ftok`.

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

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

```
key_t ftok(const char *pathname, int proj_id);
```

Utilización:

```
key_t clave;
```

```
clave = ftok (".",2);
```



Los argumentos pueden ser cualquiera, y a partir de los mismos la función genera una llave aleatoria. Lo importante es que los procesos que se conecten al mismo IPC llamen a `ftok` con los mismos argumentos (de manera de tener la misma llave)

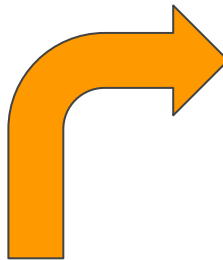
Conectarse a un IPC

Cada IPC tiene una función que permite a los procesos conectarse al mismo. Asimismo, las funciones suelen tener un parámetro que indica que de no existir el IPC, se lo cree:

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

```
int msgget(key_t key, int msgflg);
int shmget(key_t key, size_t size, int shmflg);
int semget(key_t key, int nsems, int semflg);
```

Llave
(generada con ftok)



El flag **IPC_CREAT** crea un nuevo IPC si el mismo no existe, y si se lo combina con **IPC_EXCL** intenta crearlo y si ya existe devuelve error

(para combinar flags se utiliza el operador 'OR' |)

Por ejemplo...

Si quisiera conectarme con un Message Queue (o crear uno en caso de que no esté creado):

```
key_t    llave;  
int      msgid;
```

1. Creo una llave:

```
llave = ftok (".",2);
```

2. Me conecto al msgQueue:

```
msgid = msgget ( llave , IPC_CREAT | 0666 );
```

Mismos permisos que para crear un archivo.

User	Groups	Others
6	6	6

6 equivale a lectura escritura

En este primer ejemplo no estamos chequeando errores, pero en cualquier caso si las funciones devuelven -1 indica que por alguna razón no se pudo llevar adelante la operación (un indicador del error se puede ver imprimiendo por pantalla con la funcion perror());

Liberando los IPCs

- Los IPCs generados no dependen de que los programas estén siendo ejecutados, por lo que permanecen reservados a pesar de que los mismos hayan terminado
- Para ver los IPCs que se encuentran en uso, se puede invocar el comando **ipcs** desde la terminal
- Para liberar los IPCs (cuando ya no vayan a ser utilizados) se pueden usar las funciones de control (ctl) con el flag **IPC_RMID**:

```
int msgctl (int msgid,  
int shmctl (int shmid,  
int semctl (int semid, int semnum,  
int cmd, struct msgid_ds *buf);  
int cmd, struct shmid_ds *buf);  
int cmd);
```



ID del IPC
(generada con la función get)

IPC_RMID

Utilizando los IPCs: Message Queues (I)

Todos los mensajes que se envían a través de la cola de mensajes irán con un número de mensaje asociado.

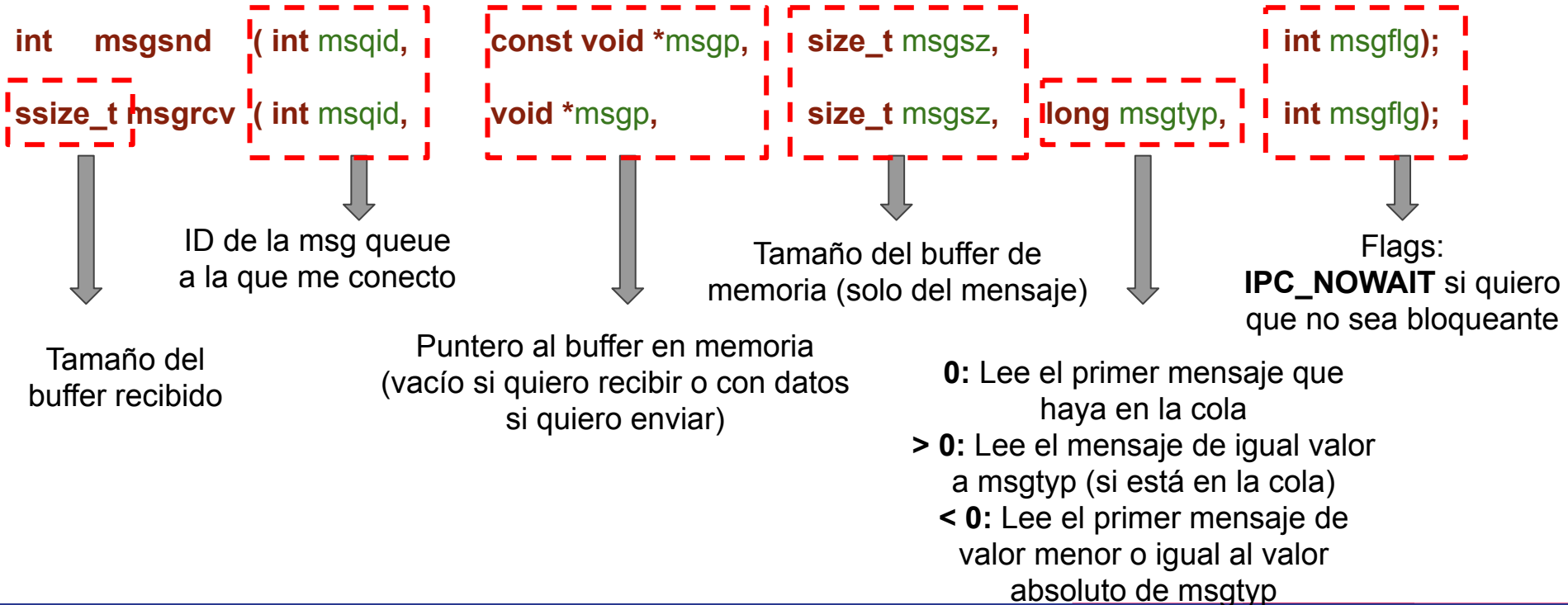
La información por ende tendrá el siguiente formato:

```
struct msgbuf {  
    long mtype;  Número de mensaje  
    void* data;  Puntero al mensaje (variable o zona de memoria en  
};                                     general)
```

Todos los mensajes que se envíen en una msgQueue deberán mandarse a través de una estructura que posea 2 campos: un long y a continuación un puntero al mensaje que se envía. Este formato garantiza que todos los mensajes tengan la misma forma, y por ende que se puedan utilizar las funciones de IPCs asociadas para enviar o recibir mensajes. El void* hace referencia a una variable de tipo puntero, pero no especifica un puntero a QUE TIPO de dato. En función del tipo de mensaje que querramos mandar, esta variable apuntará al tipo de dato adecuado (int *, char *, float *, etc.)

Utilizando los IPCs: Message Queues (II)

Una vez que nos hayamos conectado a un `MsgQueue`, tenemos 2 funciones para enviar o recibir mensajes - `msgsnd` y `msgrcv`:



Ejemplo: IPCsMsgQueues1.zip



Utilizando los IPCs: Shared Memory (I)

La memoria compartida es un bloque de memoria en donde los procesos podrán leer o escribir información.

La única funcionalidad que necesito del SO en este caso es pedir que esa memoria sea reservada para mis procesos (attach), o en caso de que ya no sea necesaria, liberar ese bloque (detach)

```
void * shmat (int shmid, char* shmaddr, int flags);
```

(Típicamente nullptr)

→

- 0 para R/W
- **SHM_RDONLY** para sólo lectura

```
int shmdt (char* shmaddr );
```

(Puntero al bloque de memoria que quiero liberar)

Utilizando los IPCs: Shared Memory (II)

```
buffer = (char *) shmat ( shm_id , NULL , 0 );
```

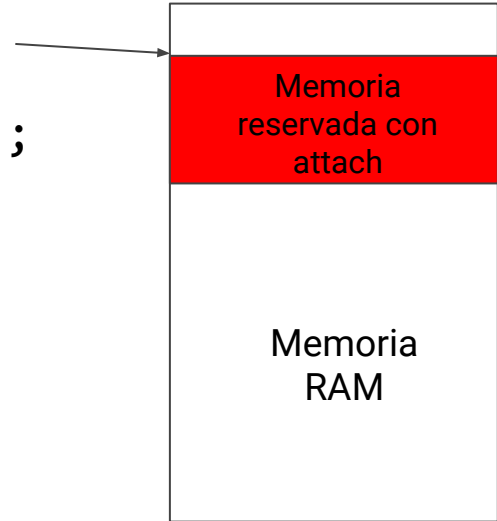
Para escribir en la memoria solamente debemos:

```
*buffer = datos;
```

Para leer en la memoria solamente debemos:

```
datos = *(buffer + 5);
```

buffer



En el caso de la memoria compartida (shm), la función get (shmget) intenta reservar un bloque de un tamaño determinado, o se conecta a un bloque ya reservado.

La función shmat solo VINCULA un puntero de nuestro programa con la memoria previamente reservada. SHMAT no reserva la memoria.

Otro tema interesante es: si hay muchos procesos conectados al mismo bloque de memoria, tenemos que garantizar de alguna manera que no se este modificando la información mientras la estoy leyendo, o que 2 procesos no intenten modificar al mismo tiempo la información. Ya veremos como hacerlo más adelante

Utilizando los IPCs: Shared Memory (II)

Para “liberar” la memoria:

```
shmdt(buffer);
```

De la misma manera que `shmat` no **RESERVA** la memoria, sino que solo vincula la memoria reservada a un puntero, la función `shmdt` **DESvincula** este puntero a esa posición de memoria, pero no la libera.

Para liberar la memoria se utiliza la función `shmctl()`, que veremos más adelante.



Ejemplo: IPCsShm1.zip

ESTE EJEMPLO ES SOLO A FINES DIDÁCTICOS, YA QUE UTILIZA UN RECURSO COMPARTIDO DE UNA MANERA QUE NO ES ACONSEJABLE (DEBEREMOS USAR SEMÁFOROS PARA COMPLETAR EL EJEMPLO)



Utilizando los IPCs: Semáforos (I)

Un semáforo es una estructura que puede utilizarse para saber cuando otros procesos están accediendo a un recurso compartido entre todos ellos.

Esta indicación la haremos mediante un número. Típicamente, el número inicialmente indica cuántos procesos pueden conectarse a dicho recurso al mismo tiempo, y al llegar a 0, indica que no pueden seguir conectándose nuevos procesos.

En el caso de un bloque de memoria (al que puede conectarse sólo un proceso por vez, para no leer información que puede estar modificándose en otro lado) el semáforo puede valer:

- 1 si está libre (indica que un proceso más puede conectarse)
- 0 si está ocupado (indica que ningún proceso más puede conectarse)

Semáforos: Particularidades (I)

Los semáforos fueron creados para “vigilar” el acceso de muchos recursos al mismo tiempo, por muchos procesos. Es por esto que cada vez que creo un semáforo, la función `semget` necesita saber cuántos recursos estaré vigilando, y en lugar de un sólo semáforo se crea un vector (array) de semáforos. A esto hace referencia el segundo parámetro en la función `semget`:

```
int semget (key_t key, int nsems, int semflg);
```

En general nuestros programas usarán los semáforos para proteger el acceso a un solo recurso. De aquí que, por lo tanto, `nsems = 1`. No tenemos que perder de vista, sin embargo, que el IPC está pensado como un array de semáforos, y por ende la función de operación nos pedirá el número de semáforo al que hacemos referencia cuando queremos bloquear o desbloquear el acceso a un recurso (que en nuestro caso, al ser un único semáforo, será el `sem_num 0`)

Semáforos: Particularidades (I)

La función semop está creada para modificar el valor de varios semáforos en un array.

La función permite sumarle o restarle un número entero a cada semáforo del array, y **en ningún momento un semáforo puede tomar un valor negativo** (la función semop queda bloqueada o devuelve error de intentar hacerlo)

En nuestro caso, solo nos interesa liberar el recurso o bloquearlo, por lo que tendremos 2 valores posibles: 0 (recurso bloqueado) o 1 (recurso libre). De aquí que nuestra operatoria sobre un semáforo sera:

1. *Intentar restarle 1 al valor del semáforo (solo puede hacerse si el valor del semáforo es 1, o sea, que está libre*
2. *Operar sobre el recurso que se encuentra protegido por el semáforo*
3. *Sumarle 1 al semáforo (indicando que nuevamente el recurso está libre y permitiendo que otro proceso lo "tome" para si)*

Utilizando los IPCs: Semáforos (II)

La función `semop` modifica un array de semáforos (de longitud `nsops`) de acuerdo a un array de estructuras de tipo `sembuf`.

Las acciones posibles sobre cada semáforo son: intentar tomar el recurso desde mi proceso actual o liberarlo.

```
int semop (int semid, struct sembuf* sop, unsigned int nsops);
```

```
struct sembuf {
```

```
    ushort sem_num;
```



Número de semáforo (puedo trabajar con más de uno)

```
    short sem_op;
```



Operación:

```
    short sem_flg;
```

```
};
```



Si `sem_flg = 0`, la función se queda esperando a que pueda realizarse la operación. si `sem_flg = IPC_NOWAIT`, retorna indicando error

> 0 : Le sumo este valor al semáforo (lo libero)

< 0 : Le intento restar este valor al semáforo (intento quedarme con el recurso)

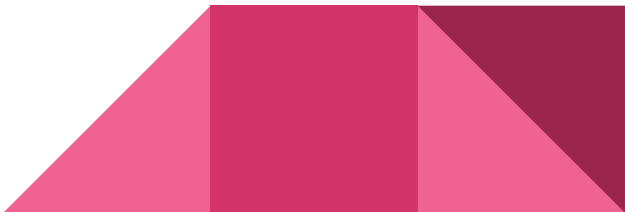
= 0 : Espero a que el valor sea 0

¡EL VALOR DEL SEMÁFORO NUNCA PUEDE SER NEGATIVO!

Utilizando los semáforos: Me “quedo” con un recurso (lo bloqueo para otros procesos)

Utilizando semop para acceder a un recurso, primero debo tomar el semáforo:

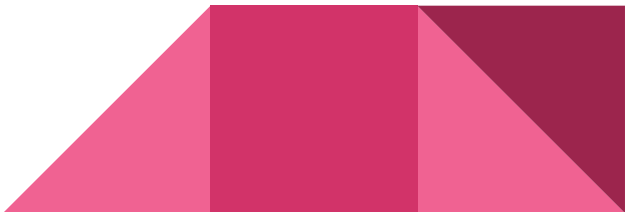
```
struct sembuf operacion;  
  
operacion.sem_num = 0;  
operacion.sem_op = -1;  
operacion.sem_flg = 0;  
  
semop( sem_id , &operacion , 1 );
```



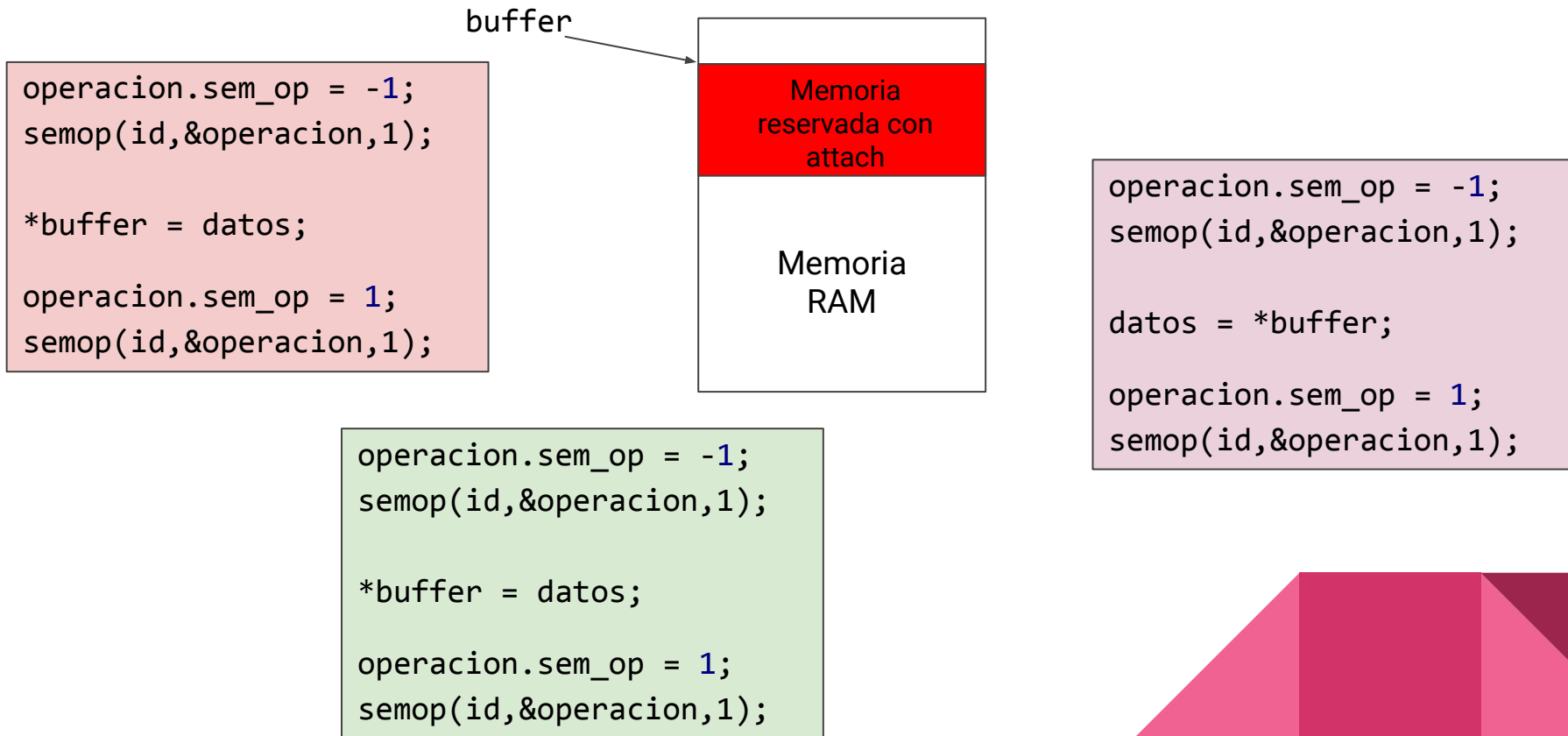
Utilizando los semáforos: “Libero” con un recurso (lo desbloqueo para otros procesos)

Después de acceder a un recurso, “libero” el semáforo usando semop:

```
struct sembuf operacion;  
  
operacion.sem_num = 0;  
operacion.sem_op = 1;  
operacion.sem_flg = IPC_NOWAIT;  
  
semop( sem_id , &operacion , 1 );
```



Múltiples procesos accediendo a ShMemory



Ejemplo: IPCsShmSems1.zip

