

CREACIÓN DE PROCESOS

La función que nos permite crear procesos por programa es **fork()**, esta función duplica el proceso que la invoca copiando el código y la memoria, al que llamaremos padre, generando un nuevo proceso al que llamaremos hijo, tiene la particularidad de retornar 2 valores distintos en caso de éxito, uno para el proceso padre y otro para el hijo.

Para utilizar fork() en nuestros programas debemos incluir la biblioteca **unistd.h**

Retorno de fork():

- Entero
 - en caso de error de creación del nuevo proceso retorna -1 al llamador .
 - en caso de éxito retorna
 - 0 al hijo.
 - positivo (PID del hijo) al padre.

Ambos procesos no comparten la memoria.

Getpid: obtener mi pid

Getppid: obtener el pid de mi padre.

Veamos un ejemplo:

```
#include <stdio.h>
#include <sys/types.h> //pid_t
#include <unistd.h>    //fork

int main
{
    pid_t pid = fork();
    if ( pid == -1)
    {
        printf ("error de creación de proceso");
        exit(1);
    }
    else
    {
        if ( pid == 0)
            printf ("soy el proceso hijo y mi pid es %d\n", getpid());
        else
        {
            printf ("soy el proceso padre y mi pid es %d\n", getpid());
            wait(NULL);
        }
    }
}
```

Espera de la finalizacion de un proceso:

pid_t **wait**(int *status), que espera la finalización del proceso (de cualquier hijo) y permite obtener información sobre el estado de finalización del mismo, dando a conocer al padre si el proceso hijo terminó normalmente o si no. En otras palabras, suspende la ejecución del proceso actual hasta que termina el proceso hijo (uno cualquiera). Devuelve el número del proceso hijo que ha terminado o el valor -1 en caso de fallo.

pid_t **waitpid**(pid_t pid, int *status, int options), que espera la finalización del proceso (de un hijo en concreto) y conoce su estado. En otras palabras, suspende la ejecución del proceso actual hasta que un proceso hijo especificado por pid termina. Devuelve el número del proceso hijo que ha terminado o el valor -1 en caso de fallo.

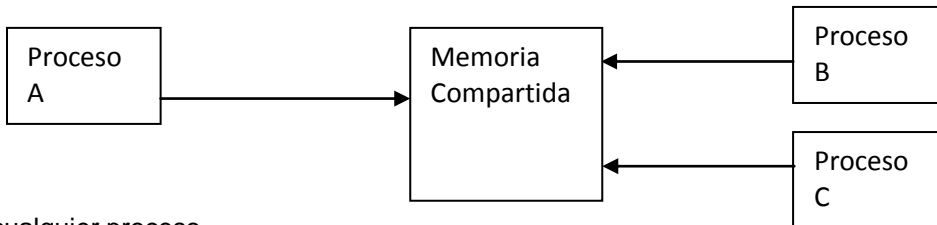
Al finalizar, los procesos hijos envían una señal llamada **SIGCHLD** a sus padres. Si el padre no espera a sus hijos con wait o waitpid, esa señal será SIGCHLD será capturada por el init, se dice que el proceso hijo quedó huérfano. Si el padre espera al hijo pero este finaliza y envía la SIGCHLD antes que el padre llegue a ejecutar wait, durante ese lapso el proceso hijo queda en un estado zombie.

COMUNICACIÓN Y SINCRONIZACIÓN DE PROCESOS

♦ MECANISMOS IPC

- Memoria compartida.
- Señales.
- Semáforos.
- Tuberías.
- Mensajes o colas de mensajes.
- Sockets.

MEMORIA COMPARTIDA (ESTÁNDAR SYSTEM-V)



Para cualquier proceso

Sincronizar el acceso a memoria (recurso crítico).

Enviar datos = escribir en la memoria.

Recibir datos = leer de la memoria.

Es el método más rápido de comunicación entre procesos.

♦ CREACION

SHMGET: permite crear una zona de memoria compartida o si ya está creada, se crea un acceso a dicha zona de memoria compartida.

<sys/shm.h>

int shmget(key_t key, int size, int option)

key: Clave para el acceso a memoria compartida.

<sys/ipc.h>

key_t ftok(char *, char)

size: Tamaño del segmento de memoria. Múltiplo del tamaño de página (PAGE_SIZE) y si la zona ya ha sido creada, el tamaño indicado debe ser menor a considerado en la creación.

option:

IPC_CREAT: Permite crear un nuevo segmento.

IPC_EXCL: Se utiliza con IPC_CREAT para que si el segmento existe, falle la creación.

mode_flags: Los 9 bits restantes permiten especificar permisos.

Valor de retorno: Si hubo éxito se retorna el identificador de la zona compartida

Si se produce un error se devuelve -1.

♦ VINCULACION

SHMAT: asocia el segmento identificado por shmid con el segmento de datos del proceso que realiza la llamada.

void *shmat(int shmid, const void *shmaddr, int option)

shmid: Identificador del segmento de memoria obtenido por shmget.

shmaddr: Permite indicar la dirección de la memoria compartida.

NULL, el sistema operativo intenta encontrar una zona libre.

No NULL, el S.O. intenta vincular la memoria compartida a la dirección indicada.

option:

SHM_RDONLY, solo lectura de la memoria.

Valor de retorno: éxito → devuelve un puntero, la dirección del segmento de memoria compartida.

char *buffer;

clave = ftok(".", 'M');

```
shmid = shmget(clave, 4096, IPC_CREAT | 0660);  
buffer = (char *)shmat(shmid, NULL, 0);
```

♦ DESVINCULACION

SHMDT

Permite desanexar una zona de memoria del espacio de direccionamiento del proceso, no se destruye.

void *shmdt(const void *shmaddr)

shmaddr: Permite indicar la dirección de la memoria compartida.

♦ CONTROL

SHMCTL

Controla la gestión del segmento de memoria compartida.

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

cmd: Tipo de operación de control a realizar.

IPC_RMID: Marca un segmento de memoria compartida que debe ser borrado.

Valor de retorno: 0 si se ejecuta con éxito.

-1 en casos de error.

SEMAFOROS (ESTÁNDAR POSIX)

• NOMBRADOS

Se identifican por un nombre de la siguiente forma: **/nombre**, que como máximo puede contener NAME_MAX - 4 caracteres. Varios procesos pueden operar sobre el mismo semáforo nombrado a través de utilizar el mismo nombre.

CREACION O APERTURA:

sem_t *sem_open(const char *name, int oflag);

sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value);

name: Nombre del semáforo.

oflag: Determina si se crea o se utiliza un semáforo existente.

0 (cero): Asume semáforo existente y no hace falta mode y value.

Si existe lo accede.

Si no existe, error (ENOENT).

O_CREAT: Crea el semáforo (son necesarios el mode y value) o lo accede.

Si no existe lo crea.

Si existe lo accede ignorando el mode y value (funciona como con oflag=0).


O_CREAT|O_EXCL: Crea exclusivamente el semáforo.

Si no existe lo crea.

Si ya existe arroja error (EEXIST).

mode: Permisos del semáforo. (Por ejemplo 0644).

value: Valor inicial del semáforo. (Máximo es **SEM_VALUE_MAX**).

Retorno  Creación EXITOSA **dirección** del semáforo creado.
Creación FALLIDA **SEM_FAILED**.

↓

Errores

EACCES:

Permisos sobre el semaforo.

EINVAL:

Valor supera SEM_VALUE_MAX.

EINVAL:

Error en el nombre.

ENAMETOOLONG:

Nombre muy largo.

OPERACIONES:

Operación	Descripción	Errores
int sem_wait(sem_t *sem)	Equivalente a P()	EINTR: Interrupción por una señal. EINVAL: Nombre del semáforo invalido.
int sem_trywait(sem_t *sem)	Equivalente a P() sin bloquear.	EINTR – EINVAL EAGAIN: No se pudo realizar el bloqueo
int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout)	Equivalente a P() durante un tiempo dado.	EINTR – EINVAL ETIMEDOUT: No se pudo realizar el bloqueo en el tiempo dado.
int sem_post(sem_t *sem)	Equivalente a V()	EINVAL: sem no es un semaforo valido. Eoverflow: SEM_VALUE_MAX excedido.
int sem_getvalue(sem_t *sem, int *sval)	Obtiene el valor del semáforo	

BORRADO:

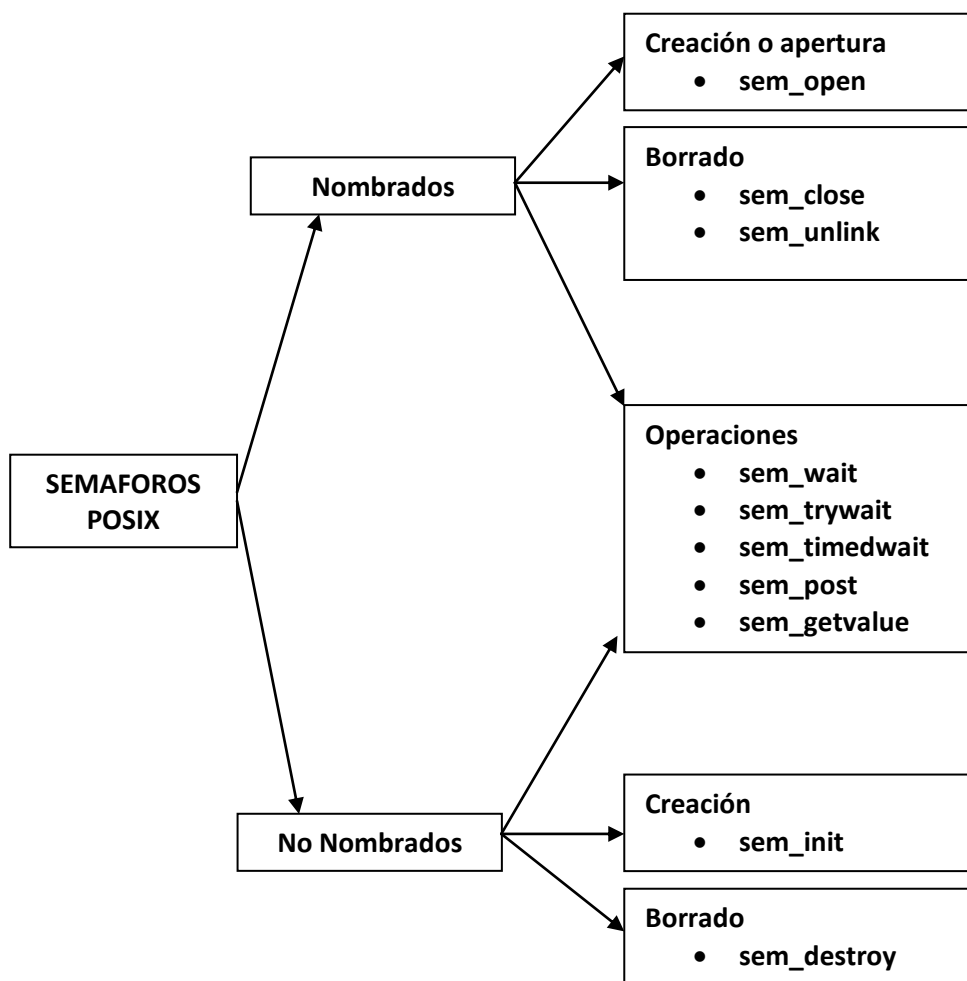
int sem_close(sem_t *sem) Libera los recursos asignados por parte del S.O. al proceso. Vuelve inaccessible al semaforo para el proceso que hace la llamada.
(EINVAL: sem no es un semaforo valido).

int sem_unlink(const char *name) Elimina el semáforo, si otros procesos hacen referencia al semáforo se pospone la destrucción del mismo.

EACCES: El llamador no tiene permiso para desvincular el semaforo.

ENAMETOOLONG.

ENOENT: No hay un semaforo con el nombre dado.



HILOS

Podemos definir a un **hilo** como la unidad más pequeña que puede ser planificada, que además de tener información propia, comparte entre otras cosas la memoria con otros hilos.

Información Propia	Información Compartida
CP	Memoria
Pila	Variables Globales
Registros	Archivos Abiertos
Estado	Señales y semáforos

Es similar en ciertos aspectos a un proceso. Pero si se pasa a ejecutar un hilo distinto de la misma tarea, el cambio de contexto es menor. En los hilos, los datos no se copian, se comparten y cada hilo accede por tanto a los mismos datos y las modificaciones serán vistas por todos los hilos de la misma tarea.

Para utilizar hilos, es necesario una biblioteca externa <pthread.h>.

OPERACIONES CON HILOS

1. CREACIÓN DE UN HILO

int pthread_create (pthread_t *x, pthread_attr_t *y, void *(*z)(void *), void *w)

x: Identificador del thread.
y: Atributos de los threads (prioridad). Recomendando pasar NULL.
z: Referencia a la función que ejecutará el thread.
w: Puntero al parámetro que se le pasará al thread. Puede ser NULL.

2. TERMINACIÓN DE UN HILO

Los hilos pueden ser Joineables (dependientes) o detachados (independientes) con respecto al proceso que los creo. Un hilo joineable no libera sus recursos al finalizar, el proceso que lo creo debe capturar la terminación del mismo y además se puede obtener su valor de retorno.

Para hilos dependientes, el proceso que los creo debe capturar su terminación con la función **pthread_join**.

int pthread_join(pthread_t x, void **ret)

x: Identificador del thread.
ret: Guarda el resultado retornado por el thread. (puede ser NULL).

Para especificar que un hilo es detachado se utiliza la función

int pthread_detach (pthread_t x)

Para terminar la función del thread se utiliza **int pthread_exit (void *retval)**

retval: Puntero genérico a los datos que se devolverán como resultado.

Para enviar señales a threads: **int pthread_kill (pthread_t x, int ns)**

Para obtener el propio TID: **pthread_t pthread_self (void)**

MANEJO DE LOS ATRIBUTOS DE UN HILO

Un hilo puede tener una serie de atributos que se manejan utilizando objetos atributos y una serie de funciones.

int pthread_attr_init (pthread_attr_t *y)

Inicia un objeto atributo que se puede utilizar para crear nuevos hilos.

int pthread_attr_destroy (pthread_attr_t *y)

Destruye el objeto de tipo atributo.

int pthread_attr_setdetachstate(pthread_attr_t *y, int detachstate)

Establece el estado de terminación del hilo.

PTHREAD_CREATE_DETACHED los hilos se crearan como independientes.

PTHREAD_CREATE_JOINABLE los hilos se crearan como dependientes.

int pthread_attr_getdetachstate(pthread_attr_t *x): Permite conocer el estado de terminación que se especifica en un objeto de tipo atributo.

SINCRONIZACIÓN DE HILOS

Como todos los hilos comparten el espacio de direcciones, se puede tener un **problema de exclusión mutua**. Para solucionarlo se implementan semáforos:

Inicializar un mutex (2 formas).

- A) `pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;`
- B) `pthread_mutex_init(pthread_mutex_t *m, const pthread_mutexattr_t *y)`
m: puntero a un parámetro del tipo `pthread_mutex_t`.
y: define el tipo de mutex.

`pthread_mutex_lock(pthread_mutex_t *m)`

Bloqueo sobre un semáforo.

`pthread_mutex_unlock(pthread_mutex_t *m)`

Libera el bloqueo existente sobre una variable mutex.

`pthread_mutex_trylock(pthread_mutex_t *m)`

Comprueba el estado del semáforo, si está libre lo bloquea y retorna 0, si ya está bloqueado retorna EBUSY, lo cual nos permitiera realizar otras tareas si no se puede bloquear el semáforo.

`pthread_mutex_destroy(pthread_mutex_t *m)`

Libera la memoria ocupada por mutex.

COMANDOS ÚTILES DE MONITOREO Y DIAGNÓSTICOS

Procesos

- ps
- top
- strace (trace de todos los syscalls y señales del proceso)
- nice, renice (setear/cambiar prioridad del proceso)

Memoria

- vmstat (estado de memoria)
- free (estado de memoria)
- /proc/meminfo (estado de memoria)
- pmap (muestra un mapa de memoria)

Recursos

- ipcs, ipcrm y /dev/shm (estado de memoria compartida y semáforos POSIX y System-V)
- lsof (lista file descriptors por proceso)

Conectividad

- netstat / ss (listado de sockets)
- ifconfig (muestra IP, máscara de red e información relacionada)
- nslookup (búsqueda DNS)

Entrada/Salida

- iostat (instalar sysstat) (muestra estadísticas de I/O)
- iotop (instalar) (como el top pero muestra I/O)
- ionice (setea prioridad de I/O)

Almacenamiento / FileSystem

- df (disk free)
- du (disk usage)
- fdisk (particiones GPT)
- mkfs (formatear fs)
- fsck (verificar fs) y badblock (verifica disco)

Otros

- /usr/bin/time (muestra todas las estadísticas del proceso)
- ssh, scp, sftp (ejecución de comandos de forma remota)