**Procesos Pesados**

•pid\_t fork();

Devuelve 0 en el hijo, y el process ID (del hijo) para el padre. El hijo comienza a ejecutarse una línea debajo del fork().

•pid\_t getpid();

Devuelve el process ID del proceso que lo ejecuta.

•pid\_t getppid();

Devuelve el process ID (del padre) del proceso que lo ejecuta.

•int execv (const char \*nombrearchivo, char \*const argv[]);

Cambia la imagen del proceso por el archivo nombrearchivo.

El argumento argv[] es un array (terminado en NULL) que contiene los argumentos para la función main del nombrearchivo.

El primer elemento del array es el mismo nombrearchivo.

•pid\_t waitpid(pid\_t pid, int \*status, int options);

Se usa para pedir información del estado a un proceso hijo, cuyo process ID es pid. Normalmente, el proceso se suspende esperando al proceso hijo.

Si pid es: -1 o WAIT\_ANY espera información de cualquier hijo.

Si hay más de un hijo que puede dar información, se elige uno al azar.

Sólo espera a un hijo, y devuelve el process ID del hijo que contestó. Devuelve -1 en caso de error.

•pid\_t wait(int \*status);

Es una versión simplificada de waitpid, que espera hasta que cualquier hijo termine.Hacer wait(&status) es equivalente a waitpid(-1, &status, 0).

**Procesos Livianos**

•pthread\_create(pthread\_t \* thread, pthread\_attr\_t \* attr, void \* (\*start\_routine)(void \*), void \* arg);

El primer argumento es una dirección a un tipo de dato pthread\_t y es inicializado por la llamada con el fin de que se use como argumentos a otras llamadas.

El segundo argumento es una lista de atributos que se indican para la ejecución del hilo, si no se quiere indicar nada, se pone como nulo

El tercer parametro es un apuntador a una función que recibe cualuqier tipo de dato (void \*) y retorna cualquier tipo de dato (void \*)

El cuarto parametro es un apuntador a cualquier tipo de datos e indica la direccion del parametro que se pasa a la rutina de ejecución

•pthread\_join(pthread\_t th, void \*\*value\_ptr);

Espera a que finalice el thread th. Devuelve cero si sale todo bien

•pthread\_exit(void \*value\_ptr);

Termina el thread, y deja value\_ptr para que sea recogido por un join.

**Pipes**

Propiedades

* Se usan solo entre procesos emparentados
* El canal es unidireccional
* Método FIFO
* La lectura es autodestructiva
* Si el pipe está lleno, el proceso escritor s bloquea en escritura si
* existe al menos un lector son finitos
* Si el pipe este vacío, el proceso lector se bloque si existe al menos escritor
* Si existe un escritor y no existe ningún lector, no tiene sentido; entonces el si envía una señal sigpipe al escritor y este muere
* Si existe un lector y no existe ningún escritor, el lector no muere y además no se bloque cuando el pipe esta vacíos; el lector vacía el pipe

• int pipe(int filedes[2])

Crea el pipe y pone los files descritories de lectura y escritura en pipe[0] y pipe[1]

•close(filedes[0]);

Cierra el extremo de lectura, el proceso que hace el close se convierte en escritor.

•close(filedes[1]);

Cierra el extremo de escritura, el proceso que hace el close se convierte en lector.

•int read( filedes[0], void \*buffer, int tamaño);

Lee del pipe los primeros (tamaño) bytes, y los guarda en buffer.

Devuelve la cantidad de bytes leída, o -1 en caso de error.

•int write( filedes[1], void \*buffer, int tamaño);

Escribe en el pipe los primeros (tamaño) bytes de buffer.

Devuelve la cantidad de bytes escrita, o -1 en caso de error.

**Semáforos**

Es un recurso de IPC que da acceso a un recurso a unos de los procesos y se lo niega a los demás mientras el proceso no termine. Funciona como una variable contadora, donde cada vez que se lo usa su valor decrementa. Una vez que se sale de la sección crítica, el proceso debe incrementar el valor del semáforo para que otros procesos puedan seguir su ejecución.

•int semget( key\_t key, int cantidad, int semflg);

Crea tantos semáforos como indique cantidad.

Algunos de los flags son:

* IPC\_CREATE: si no está creado, lo crea.
* 0600: permiso de lectura y escritura para el propietario.
* Devuelve el id de los semáforos.

•int semctl(int semid, int semnum, int cmd, ...);

Para remover el semáforo del sistema operativo

* El identificador del semáforo, dado por la llamada semget
* El número de semáforo del conjunto sobre el cual se aplica el comando.
* El comando a aplicar, que en esta caso es IPC\_RMID
* El cuarto parámetro es opcional, dependiendo del comando a aplicar.

•int semop(int semid, struct sembuf \*sops, size\_t nsops);

Permite modificar el valor de un semáforo, pero de una forma atómica

struct sembuf {

short sem\_num; /\* semaphore number \*/

short sem\_op; /\* semaphore operation \*/

short sem\_flg; /\* operation flags \*/

}

* El primer campo indica a que elemento del conjunto de semáforos se le aplican las operaciones.
* El segundo campo indica el tipo de operacion a aplicar sobre el elemento del semáforo, cumpliendo las siguientes reglas:
* Si sem\_op es positivo o negativo implica que se debe alterar el valor del semáforo.
* Si sem\_op es cero, indica lectura sobre el semáforo.

Las siguientes condiciones se aplican si:

* Si el valor es positivo, el semáforo ve incrementado su valor por lo indicado en sem\_op.
* Si sem\_op es cero y el valor del semáforo es cero, retorna de inmediato de la llamada.
* Si sem\_op es cero y el valor de semáforo NO ES cero, suspende la ejecución del proceso. Esto es, el proceso se queda esperando hasta que el semáforo tenga valor 0.
* Si sem\_op es negativo y el valor del semáforo es igual o mayor al valor absoluto de sem\_op, retorna de la llamada, de lo contrario, se queda esperando hasta que esta condición sede.

El tercer campo indica algunas banderas de control de la operación, que afecta

el comportamiento de la operación a realizar.

**Memoria Compartida**

Es una técnica de IPC donde hay un área de memoria donde los datos están disponibles para todos los procesos y no requiere que se realicen copias privadas de los mismos. Hay que resolver el problema de la exclusión mutua mediante el uso de semáforos.

Implementar un área de memoria compartida, implica obtener una clave de IPC, luego crear un área de memoria compartida usando dicha clave e indicando su tamaño en bytes y los permisos sobre la misma. Una vez creada la memoria compartida, ésta es accesible a través de un puntero local, y se puede desplazar, leer y cambiar dicha memoria como si se tratase de un área de memoria local del proceso.

•int shmget(key\_t key, size\_t size, int shmflg)

* La llave o etiqueta dentro del kernel.
* El tamaño del segmento.
* Una bandera, si esta bandera contiene IPC\_CREAT, crea el segmento y se puede combinar con los permisos de lectura y escritura para el propietario, grupo y otros. Si es 0, solo se conecta a memoria compartida.

•void \*shmat(int shmid, const void \*shmaddr, int shmflg)

Permite leer o escribir a un segmento de memoria

* El identificador de memoria compartida, generado por shmget.
* La dirección a partir de la cual se quiere indexar el segmento, generalmente

es (void \*) 0.

* Banderas.

•int shmdt(char \*shmaddr)

Para desconectarse de un segmento de memoria.

**Colas**

Una cola de mensajes es una lista enlazada de mensaje almacenada dentro del kernel del S.O. e identificada a través de un id (del tipo key\_t). Se pueden enviar mensajes a la cola en un orden, y ser extraídos de distintas maneras. Todos los mensajes tienen una estructura idéntica, que está compuesta por un número del tipo long, que representa el tipo de mensaje, y luego continúa con una serie de caracteres acorde con la necesidad de cada aplicación.

•int msgget(key\_t key, int msgflg)

* El primer parámetro es la llave del IPC, que es un numero entero.
* El segundo parámetro es un conjunto de banderas, en caso de aplicar una creación de la cola de mensajes se debe indicar la bandera IPC\_CREAT combinada con el esquema de permisos (lectura y escritura sobre la cola; en caso de que la cola ya esté creada se debe indicar un 0.
* Esta llamada retorna un número entero, que indica el identificador de la cola. Si es negativo, existe un error en esta llamada.

Para remover una cola

•int msgctl(int msqid, int cmd, struct msqid\_ds \*buf )

* msqid es el identificador de la cola, retornado por msgget
* Cmd es un comando, en este caso IPC\_RMID
* El tercer parámetor se pone a nulo cuando se pone IPC\_RMID.

Para enviar poder enviar un mjs a una cola

•int msgsnd(int msqid, const void \*msgp, size\_t msgsz,

int msgflg);

* msqid es el identificador de la cola, proporcionado por una llamada msgget.
* msgp es un apuntador a void y debe apuntar a la estructura de datos que contiene los datos del mensaje o el mensaje como tal.
* msgsz indica el tamaño del mensaje.
* msggflg son banderas.

La estructura de datos a la que debe apuntar el segundo parámetro debe

ser del siguiente tipo:

struct mensaje {

long mtype; /\*tipo de mensaje\*/

char mtext[1]; /\*campo de cualquier tipo, aqui se especifica un

solo caracter\*/

}

Para poder recibir un mensaje

•int msgrcv(int msqid, void \*msgp, size\_t msgsz, long msgtyp,

int msgflg);

* msqid es el identificador de la cola, proporcionado por una llamada msgget.
* msgp es un apuntador a void y debe apuntar a la estructura de datos que contiene los datos del mensaje o el mensaje como tal.
* msgsz indica el tamaño del mensaje a recibir.
* msgtyp es el tipo de mensaje a recibir, si es 0, se recibe cualquier mensaje, si es positivo se recibe un mensaje con ese tipo.
* msgflg son banderas, si se desea bloquear al proceso hasta que reciba un
* mensaje de tipo msgflg, se debe poner a 0; de lo contrario con la bandera IPC\_NOWAIT.
* Esta llamada retorna el número de bytes colocados en la estructura apuntada por msgp o un valor negativo ante un error.

**Señales**

Una señal es un "aviso" que puede enviar un proceso a otro proceso. El sistema operativo Unix se encarga de que el proceso que recibe la señal la trate inmediatamente. De hecho, termina la línea de código que esté ejecutando y salta a la función de tratamiento de señales adecuada. Cuando termina de ejecutar esa función de tratamiento de señales, continua con la ejecución en la línea de código donde lo había dejado.

•signal(int señal, void (\*)(int) funcionHandler);

Ej: signal(SIGUSR1,handler)

Si en funcionHandler pongo SIG\_IGN, se ignora a la señal.

Si no hay handler para una señal, el proceso muere.

•alarm(); Se envía una señal SIGALRM al proceso que la ejecuta luego de pasar los segundos de alarm().

•pause(); hace que el proceso quede suspendido hasta que llegue una señal

•raise(); la funcion envía una señal al programa de ejecución.

•sigset\_t set; define variable para manejar conjunto d señales.

•sigemptyset(&set); Inicializa conjunto de señales set vacío.

•sigaddset(&set,SIGINT); Agrega SIGINT al conjunto set.

•sigprocmask (int how, const sigset\_t \*set, sigset\_t \*o\_set)

* Examina o modifica la máscara de señales activa para un proceso
* Si una señal está bloqueada por la máscara, no es procesada hasta el momento en que deja de estar bloqueada por la máscara (se memoriza que llegó).
* Parámetro how
* SIG\_BLOCK: añade un conjunto de señales a las que se encuentran bloqueadas en la máscara actual
* SIG\_UNBLOCK: elimina un conjunto de señales de las que se encuentran bloqueadas en la máscara actual
* SIG\_SETMASK: especifica un conjunto de señales que serán bloqueados
* Parámetro set: conjunto de señales que serán utilizadas para la modificación (puede ser NULL)
* Parámetro o\_set: máscara previa a la modificación (puede ser NULL)

Si funciona retorna 0, en caso de error -1.