Comandos de APSO – Programación (C estándar)

Definición	Comando	Librería/s
HILOS – PRACTICA 2		
Crea un hilo	**pthread_create(a,NULL,c,NULL)** ➔ a = variable tipo hilo (pthread) por referencia ➔ c = función que ejecuta, formato: (void *)&hilo1	**<pthread.h>**
Lo ejecuta el proceso principal. Se queda esperando a que el hilo termine	**pthread_join(a,NULL)** ➔ a = variable tipo hilo (pthread) por referencia	**<pthread.h>**
Lo ejecuta el proceso principal. No espera a que termine el hilo	**pthread_detach(a)** ➔ a = variable tipo hilo (pthread) por referencia	**<pthread.h>**
Lo ejecuta el hilo	**pthread_exit(NULL)**	**<pthread.h>**
CREACIÓN DE PROCESOS – PRACTICA 3		
Hace una copia del proceso que llama al fork. Devuelve un entero (PID del nuevo proceso)	**fork()**	**<unistd.h>**
Crea un proceso nuevo. El proceso que llame al execl muere. Lo suyo es hacer una copia y en la copia hacer el execl	**execl(a,b,NULL)** ➔ a = nombre entre comillas “” ➔ b = nombre entre comillas “”	**<unistd.h>**
SINCRONIZACIÓN DE PROCESOS – PRACTICA 4		
El padre espera a que uno de sus hijos termine. Devuelve un entero que es el PID del proceso hijo que ha terminado	**wait()**	**<sys/wait.h>** **<sys/types.h>**
Lo usan los procesos hijos para indicarle al padre de que han terminado	**exit()**	**<stdlib.h>**
Prepara a un proceso para la llegada de una señal	**sigaction(a,b,NULL)** ➔ a = señal a la que se tiene que preparar ➔ b = & variable de tipo sigaction (struct sigaction)	**<signal.h>**
Hace lo mismo que sigaction pero es más sencillo de implementar	**signal(a,b)** ➔ a = señal de la que se tiene que preparar ➔ b = rutina que ejecuta al recibir la señal	**<signal.h>**
Envía una señal a un proceso	**kill(a,b)** ➔ a = PID del proceso al que envía la señal ➔ b = señal que envía	**<signal.h>** **<sys/types.h>**

Al pasar los segundos pasados por parámetro ejecuta la señal especial **14**. Si se hace otro alarm antes de que termine el antiguo, el antiguo se cancela. Para anular un alarm se ejecuta alarm(0).	**alarm(a)** ➔ a = número de segundos antes de ejecutar la señal	**<unistd.h>**
El proceso que ejecuta el **pause()** se queda esperando hasta recibir una señal cualquiera. Devuelve un entero que es la señal que ha llegado al pause.	**pause()**	**<unistd.h>**
Hace que le proceso que llame al **sleep()** se quede parado hasta haber transcurrido los segundos.	**sleep(a)** ➔ a = segundos que el proceso se queda parado	**<unistd.h>**
COMUNICACIÓN DE PROCESOS (FIFOS Y PIPES) – PRACTICA 5		
FIFOS		
Crea un fichero FIFO. Lo crea el proceso principal	**mkfifo(a,b)** ➔ a = nombre de la FIFO entre comillas “” ➔ b = Permisos en formato numérico (0600)	**<sys/stat.h>** **<sys/types.h>**
Borra el fichero FIFO. Se ejecuta tantas veces como FIFOs haya abiertas	**unlink(a)** ➔ a = nombre de la FIFO entre comillas “”	**<unistd.h>**
Sirve para abrir ficheros. Devuelve un entero que es la posicion en la T.C. en la que se ha colocado el fichero	**open(a,b)** ➔ a = nombre del fichero a abrir entre comillas “” ➔ b = modo de apertura (suele ser **O_RDWR**)	**<sys/stat.h>** **<sys/types.h>** **<fcntl.h>**
Cierra el fichero pasado por parametro	**close(a)** ➔ a = posicion de la T.C a cerrar	**<unistd.h>**
Escribe en un fichero	**write(a,b,c)** ➔ a = fichero en el que escribir ➔ b = variable que se va a escribir por referencia & ➔ c = sizeof(variable)	**<unistd.h>**
Lee de un fichero	**read(a,b,c)** ➔ a = fichero en el que leer ➔ b = variable que se va a leer por referencia & ➔ c = sizeof(variable)	**<unistd.h>**
Crea un fichero con el nombre pasado por parámetro.	**creat(a,b)** ➔ a = nombre entre comillas “” ➔ b = permisos en formato numérico (0600)	**<unistd.h>**

TUBERÍAS		
Crea una tubería. Es necesario crear antes el array de dos posiciones antes de pasarlo por parámetro	**pipe(a)** ➔ a = array de enteros de 2 posiciones	**<unistd.h>**
Duplica la posición de la T.C. pasada por parámetro y la añade en el primer hueco que vea de esta.	**dup(a)** ➔ a = posicion de la T.C. a duplicar	**<unistd.h>**
COMUNICACIÓN DE PROCESOS (COLAS DE MENSAJES) – PRACTICA 6		
Almacena la **clave de la cola** en una variable de tipo **key_t.** Si queremos que los procesos usen la misma cola, el segundo parámetro debe de ser el mismo.	**ftok(a,b)** ➔ a = fichero existente, el que sea, entre comillas “” ➔ b = número cualquiera	**<stdio.h>** **<stdlib.h>** **<string.h>** **<sys/msg.h>** **<errno.h>**
Devuelve un entero que es el **identificador de la cola**. Este nos servirá para trabajar con la cola	**msgget(a, IPC_CREAT)** ➔ a = clave obtenida al hacer ftok	**<stdio.h>** **<stdlib.h>** **<string.h>** **<sys/msg.h>** **<errno.h>**
Escribe una estructura en la cola de mensajes. El cuarto parametro da igual si se pone 0 o IPC_NOWAIT	**msgsnd(a,b,c,d)** ➔ a = identificador de la cola (msgget) ➔ b = variable de tipo struct que se va a escribir ➔ c = sizeof(struct)-sizeof(long) ➔ d = Poner 0(espera) o IPC_NOWAIT(no espera)	**<stdio.h>** **<stdlib.h>** **<string.h>** **<sys/msg.h>** **<errno.h>**
Lee una estructura de la cola. La espera aquí sí que es importante tenerla en cuenta	**msgrcv(a,b,c,d,e)** ➔ a = identificador de la cola (msgget) ➔ b = variable de tipo struct en la que guardar la info ➔ c = sizeof(struct)-sizeof(long) ➔ d = tipo de estructura que queremos leer ➔ e = Poner 0(espera) o IPC_NOWAIT(no espera)	**<stdio.h>** **<stdlib.h>** **<string.h>** **<sys/msg.h>** **<errno.h>**
Borra la cola de mensajes a través de su identificador de cola	**mgctl(a,IPC_RMID,0)** ➔ a = identificador de la cola (msgget)	**<stdio.h>** **<stdlib.h>** **<string.h>** **<sys/msg.h>** **<errno.h>**

PRACTICA 2

Para usar programas con hilos hay que poner la librería **"pthread.h".**

Para **compilar** programas **".c"** con hilos hay que poner al final del cc **"-lpthread"**

EJ.: **cc nombre_fichero.c -o nombre_fichero -lpthread**

PRACTICA 3 (REDIRECCIONAMIENTO)

Tabla de canales original:

0 --> Teclado

1 --> Pantalla

2 --> Errores(Pantalla)

Cuando se hace un fork, la copia hereda la tabla de canales entera del proceso que llama al fork.

Cuando se hace un execl, el nuevo proceso solo copia las 3 primeras posiciones de la tabla de canales.

Si se quiere cambiar solo la tabla de canales de la copia, se debe de hacer dentro del "if(x==0)".

Para cambiar los canales:

·**close(pos)** ---> cierra la posicion "pos".

Si se cierra la pos 1, el proceso no podrá mostrar nada por pantalla

·**open("fichero", O_WRONLY | O_CREAT)** ---> abre un fichero y lo pone en la primera posicion libre que encuentre en la tabla de canales.

- **O_WRONLY** -----> abre el fichero solo para escritura
- **O_CREAT** ---------> si el fichero no existe lo crea

PRACTICA 4

La estructura **sigaction** tiene dos métodos importantes:

➔ **sa_flags** => siempre se iguala a 0.

➔ **sa_handler** => se iguala a la rutina que tiene que ejecutar al recibir la señal.

EJ.: **struct sigaction s1;**

s1.sa_flags = 0;

s1.sa_handler = rutina10;

PRÁCTICA 5

Para que otro proceso pueda escribir en la tubería tiene que hacer lo siguiente:

Estando dentro de la condición if(x==0) después de hacer el fork()

- Quitar la posición 2 de la T.C. usando el comando **close(2).**
- Duplicar la posición del array de dos posiciones que queremos que tenga el proceso hijo.
- Lo normal después de pasarle la posición al hijo, es cerrar esa misma posición en el padre.

EJ.: **int x;**

If(x == 0)

{

close(2);

dup(tubería[0]) //Duplicamos la lectura para el hijo

execl(“hijo”,”hijo”,NULL)

}