








-  **MAKEFILE:**
-  **MANEJO DE HILOS:**
  - Crear hilo
  - Esperar hilo
  - No esperar hilo
  - Finalizar hilo
-  **MANEJO DE PROCESOS:**
  - FORK y EXECL:
-  **SEÑALES:**
  - Prepararse para una recibir una señal
  - Enviar señal a un proceso
-  **FICHEROS:**
  - Abrir fichero
  - Escribir en fichero
  - Leer de fichero
  - Cerrar fichero
-  **FIFOS:**
  - Crear FIFO
  - Abrir FIFO
  - Escribir en la FIFO
  - Leer de la FIFO
  - Cerrar FIFO
  - Eliminar FIFO
-  **PIPES:**
  - Crear PIPE
  - Escribir en la PIPE
  - Leer de la PIPE
  - Cerrar PIPE
-  **COLAS:**
  - struct mensaje
  - Crear COLA
  - Abrir COLA
  - Escribir en la COLA
  - Leer de la COLA
  - Eliminar COLA



# MAKEFILE:

Junto a all se escriben los nombres de los ejecutables. Cada ejecutable depende de un fichero **.c** y sus dependencias **.h**. Si algo se modifica, se ejecuta el comando asociado en la línea inferior.

El fichero se llamara **Makefile** con la M en mayuscula.

```
all: programa1 programa2 // archivos resultantes de ejecutar los comandos

programa1: programa1.c graficos.h // archivos que si se modifican, haran que el
programa1 ejecute el comando que tiene
    cc programa1.c graficos.h -o programa1 // codigo a ejecutar

programa2: programa2.c
    cc programa2.c -o programa2 -lpthreads
```



# MANEJO DE HILOS:

```
#include <stdio.h>
#include <pthread.h>

void *hilo1(void) {
    printf("hola, soy el hilo 1 \n");
}

void *hilo2(int *numero) {
    printf("Hola, soy el hilo 2 y me pasan el numero %d\n",*numero);
}

void *hilo3(int *numero) {
    printf("Hola, soy el hilo 3 y me pasan el numero %d\n",*numero);
    *numero=*numero + 5;
    pthread_exit(numero);
}

main() {
    pthread_t h1,h2,h3;
    int parametro=7,*retorno;
    printf("Comenzamos el proceso de los hilos \n");
    pthread_create(&h1,NULL,(void *) &hilo1,NULL);
    pthread_create(&h2,NULL,(void *) &hilo2,&parametro);
    pthread_create(&h3,NULL,(void *) &hilo3,&parametro);
    printf("Finalizamos la creacion de los hilos\n");
    pthread_join(h1,NULL);
    pthread_join(h2,NULL);
```

```
pthread_join(h3,(void *) &retorno);  
printf("El hilo 3 devuelve %d\n",*retorno);  
printf("El valor de parametro tambien cambia, y vale %d\n",parametro);  
}
```

## Crear hilo

---

```
pthread_create(&h1,NULL,(void *) &hilo1,NULL);
```

## Esperar hilo

---

```
pthread_join(h2,NULL);  
pthread_join(h3,(void *) &retorno);
```

## No esperar hilo

---

```
pthread_detach(h1);
```

## Finalizar hilo

---

```
int *resultado;  
pthread_exit(resultado);
```



## MANEJO DE PROCESOS:

---

## FORK y EXECL:

---

```
int vpid = fork();
if (!vpid) {
    execl("exeFileName", "exeFileName", NULL);
}
```

```
int vpid = fork();
if (!vpid) {
    close(2); // cerramos la fila 2 de la tabla de canales, correspondiente con
la salida de error estandar
    dup(tuberia[1]); // abre el descriptor de fichero en la primera entrada
disponible de la tabla de canales
    // Todo esto se tiene que hacer aquí para pasar una tubería a un hijo
porque es justo después de
    // crear un nuevo proceso como este pero justo antes de que se reemplace
por otro ejecutable.
    // SI NO NECESITAS PASAR UNA TUBERÍA A UN HIJO, IGNORA ESTO.
    execl("exeFileName", "exeFileName", NULL);
}
```



## SEÑALES:

---

# Prepararse para una recibir una señal

---

Solo podemos usar la señal 10 y la 12 para sincronizar nuestras cosas.

En el siguiente ejemplo usamos **SIGALRM** (la señal 14) para que mientras no llegue, espere usando **pause()** y después de que llegue la alarma en 5 segundos, continúe el programa.

```
int seguir = 1;
void fin(int n) {
    seguir = 0;
}

int main() {
    signal(SIGALRM, fin); // SIGALRM = 14
    alarm(5);
    while (seguir) {
        pause();
    }
}
```

```
return 0;
}
```

## Enviar señal a un proceso

```
kill(pidProcesoB, 10);
```



## FICHEROS:

### Abrir fichero

```
int descriptor = open("nombreFichero", O_CREAT | O_WRONLY | O_APPEND);
```

### Escribir en fichero

```
int descriptor = open("fichero.txt", O_CREAT | O_WRONLY | O_APPEND, 0600);
char character = 's'; // esto equivale a un byte
int num_bytes = write(descriptor, &character, sizeof(character));
```

### Leer de fichero

```
int descriptor = open("fichero.txt", O_RDONLY);
char character;
if (read(descriptor, &character, sizeof(character)) == -1){
    printf("Error al leer del fichero");
    exit(-1);
}
```

# Cerrar fichero

---

```
close(descriptor);
```



## FIFOS:

Las funciones que se usan para tratar FIFOS, devuelven un -1 en caso de error. En las funciones read() y write() se devuelve el numero de bytes leidos o escritos, -1 en caso de error.

## Crear FIFO

---

```
if (mkfifo("nombreFifo", 0660)==-1){
    printf("Error al crear la FIFO");
    exit(-1);
}
```

## Abrir FIFO

---

```
int fifoAC = open("fifoAC", O_RDWR); // O_WRONLY - O_RDONLY
if (fifoAC==-1) {
    perror("Error de open fifo");
    exit(-1);
}
```

## Escribir en la FIFO

---

```
int magicNumber = 43;
if (write("nombreFifo",&magicNumber,sizeof(magicNumber)==-1){
    printf("Error al cerrar la FIFO");
}
```

```
        exit(-1);  
    }
```

## Leer de la FIFO

---

```
if (unlink("nombreFifo")==-1){  
    printf("Error al leer de la FIFO");  
    exit(-1);  
}
```

## Cerrar FIFO

---

```
if (close(fifoAC)==-1){  
    printf("Error al cerrar la FIFO");  
    exit(-1);  
}
```

## Eliminar FIFO

---

```
if (unlink("nombreFifo")==-1){  
    printf("Error al eliminar la FIFO");  
    exit(-1);  
}
```



## PIPES:

---

## Crear PIPE

---

```
int tuberia[2]; // 0 es lectura y 1 es escritura  
pipe(tuberia);
```

# Escribir en la PIPE

```
int numeroEscrito = 5;
if (write(tuberia[1],&numeroEscrito,sizeof(numeroEscrito))== -1){
    printf("Error al escribir en la PIPE");
    exit(-1);
}
```

# Leer de la PIPE

```
int numeroLeido;
if (read(tuberia[0],&numeroLeido,sizeof(numeroLeido))== -1){
    printf("Error al leer de la PIPE");
    exit(-1);
}
```

# Cerrar PIPE

```
close(tuberia[0]);    // extremo de lectura
close(tuberia[1]);    // extremo de escritura
```



## COLAS:

## struct mensaje

Con las colas necesitamos usar un struct con los campos deseados. El struct acaba en ; despues de la llave de cierre. Comienza con un long que indica con un int el tipo de Mensaje. (Tipo: 1, 2, 3,...)



```
struct Mensaje{
    long tipo;
    int numero;
    char texto[4];
};

mensaje.tipo = 1;
mensaje.numero = atoi("Ascii_TO_Integer,texto_a_su_valor_numerico_en_ASCII");
strcpy(mensaje.texto,"hola");
```

## Crear COLA

---

```
// El fichero y el numero pueden
// ser cualquiera. Siempre usar el
// mismo en todos los ejecutables
// que usen la misma cola.

key_t clave = ftok("./Makefile",1);
if(if clave == key_t-1) perror("error al crear la clave de la cola");
```

## Abrir COLA

---

```
int fdcola = msgget(clave,0600|IPC_CREAT);
```

## Escribir en la COLA

---

```
// Devuelve -1 en caso de error.
// En caso de que ocurra un error y no se pueda escribir en la cola hay 2 opciones:
// Usar IPC_NOWAIT o 0.
// 0 bloqueará el proceso hasta que pueda escribir y luego continuara
// IPC_NOWAIT como su nombre indica, no espera y continuara, devuelve -1 si falla y
// no espera

if (msgsnd(fdcola,&mensaje, sizeof(mensaje)-sizeof(long),IPC_NOWAIT) == -1){
    perror("B: no se puede enviar a la cola de mensajes");
}
```

# Leer de la COLA

---

```
// Devuelve -1 en caso de error
// Puede hacerse que se espere o que siga la ejecucion usando 0 o IPC_NOWAIT
respectivamente.
// El 1 es el long tipo que queremos leer exclusivamente, podria ser 2, 3, 4...

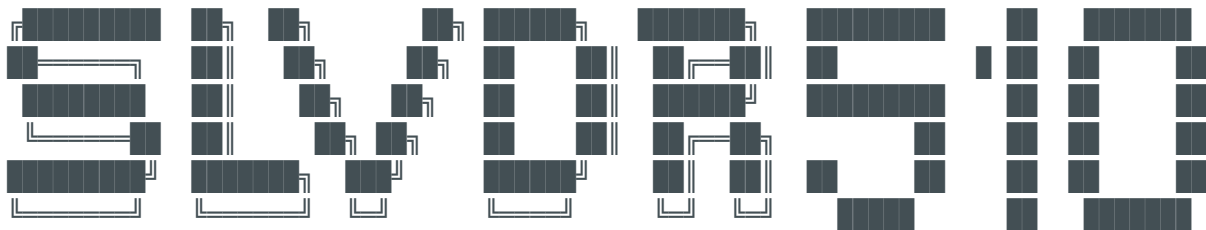
msgrcv(fdcola,&mensaje,sizeof(mensaje)-sizeof(long),1,IPC_NOWAIT);
msgrcv(fdcola,&mensaje,sizeof(mensaje)-sizeof(long),1,0);
```

## Eliminar COLA

---

Aquí siempre vamos a buscar ayudarnos de las sugerencias de vscode. Ya que las cosas raras te las indica ahí y con un poco de memoria lo recuerdas.

```
msgctl(fdcola,IPC_RMID,NULL);
```



[github.com/slvdr510](https://github.com/slvdr510)