

PROGRAMACIÓN GNU/LINUX

SISTEMAS OPERATIVOS EN TIEMPO REAL

Fernando Dorado.

fdorado.rueda@gmail.com

Tabla de contenido

1. SISTEMAS OPERATIVOS EN TIEMPO REAL	5
1.1 Introducción	5
1.2 Clasificación.....	5
1.3 Características STR	6
2. PROGRAMACIÓN PARA APLICACIONES STR	8
2.1 Introducción	8
2.2 Máquina convencional:.....	8
2.3 Máquina Operativa	10
2.4 Máquina Simbólica.....	16
3. CONCURRENCIA	18
3.1 Introducción	18
3.2 Concurrencia en POSIX.....	19
3.2.1 Creación de procesos.	19
3.2.2 Hilos ("threads").....	23
4. SEÑALES POSIX.....	27
4.1 Introducción	27
4.2 Programación de la máscara del proceso	28
4.2.1 Modificación Máscara.....	29
4.3 Programación para señales POSIX 1003.1a.....	30
4.4 Programación para señales POSIX 1003.1b	32
4.5 Generación de señales	35
4.6 Espera de señales.....	36
4.6.1 Espera en 1003.1a.....	36
4.6.2 Espera en 1003.1.b.....	36
4.7 Manejadores	38
4.8 Temporizador basado en señales.....	38
4.9 Señales en procesos multihilo.....	39
TEMA 5. TEMPORIZACIÓN	40
5.1 Introducción	40
5.2 Servicios de temporización	40
5.2 Temporizadores en ADA.....	41
5.3 Temporización en POSIX 1003.1a	43
5.4 Temporización en POSIX 1003.1b	43

TEMA 6. MEMORIA COMPARTIDA.....	48
6.1 Sincronización.	48
6.1.1 Espera activa (“Busy Wait”)	48
6.1.2 Semáforos	49
6.3 Memoria Compartida en POSIX 1003.1b	52
6.4 Mutex	56
6.5 Variables de condición	58
6.6 Regiones crítica condicionales	60
6.7 Monitores.....	61
6.8 Datos y Objetos protegidos (ADA)	61
7. MENSAJES	63
1. Introducción	63
2. Descripción a nivel de lenguaje: ADA	64
3. Paso de mensajes local en POSIX	65
4. Colas de mensaje.....	67
ANEXO.....	74
2. PUNTEROS	74

1. SISTEMAS OPERATIVOS EN TIEMPO REAL

1.1 Introducción

DEFINICIÓN

Los STR están interaccionando con el mundo real. Responden al paso del tiempo (activados por tiempo) o sucesos externos (activados por eventos). También es importante el tiempo en el que se producen los resultados.

En la práctica, las restricciones temporales son una de las especificaciones que debe cumplir el sistema, en caso de no cumplirla el sistema no puede funcionar correctamente y deberá ser rechazado.

Además, estos sistemas son muy fiables y tienen tolerancia ante fallos.

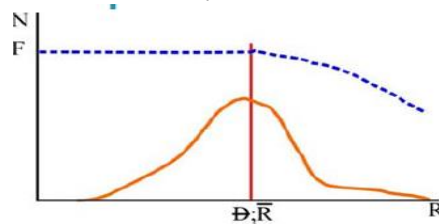
RESTRICCIONES TEMPORALES

- **Ciclo (T)**
- **Coste (C).** Tiempo de ejecución máximo si no existen interferencia de otras actividades.
- **Tiempo de respuesta (R).** Tiempo de ejecución máximo (peor) en condiciones reales. No puede ser inferior al coste.
- **Tiempo límite o “deadline” (D).** Máximo tiempo de respuesta admisible.
- **Restricción temporal típica.** $R \leq D$.

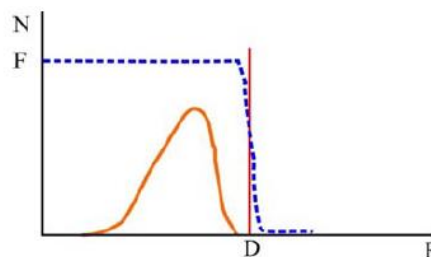
Las tareas de alta prioridad “roban” tiempo a las de baja prioridad (interferencia).

1.2 Clasificación.

1. **No de tiempo real.** No se definen plazos

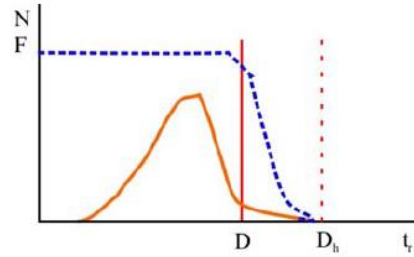


2. **Sistemas críticos.** El tiempo de respuesta debe ser menor que el máximo. De no ser así, será rechazado.



3. **Sistemas no críticos.** Se pueden tolerar retrasos dentro de ciertos límites. Subtipos:

- a. Restricciones firmes. Si se incumplen el resultado carece de valor



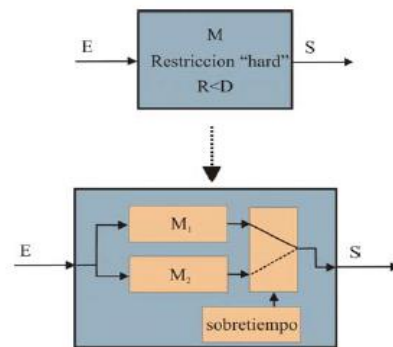
- b. Restricciones no firmes. Si se incumplen, el resultado es útil, pero no óptimo.

Un mismo sistema puede tener ambos tipos de restricciones en funciones distintas y algunas funciones pueden tener ambos tipos de restricciones.

COMBINACIÓN DE RESTRICCIONES

M1: Restricción para una solución óptima.

M2: Restricción para solución subóptima. Se usa en el caso del sobretiempo



SISTEMAS EMPOTRADOS:

No siempre son STR. Funcionan como un componente más del sistema ya que no tienen los recursos habituales en un computador. Tiene recursos y periféricos apropiados para su función.

1.3 Características STR

- Tiempo de respuesta predecible.
- Sistemas grandes y complejos.
 - Diseño basado en especificaciones.
 - Extensibilidad.
- Necesidad de soporte para cálculo numérico.
 - Lenguaje con soporte adecuado para entero y flotante.
- Alta fiabilidad y tolerancia a fallos.
 - Alta fiabilidad debido al diseño basado en especificaciones.
 - Supera fallos y circunstancias excepcionales
 - Redundancia estática. Fallos sistemáticos requieren diversidad de diseño.
 - Redundancia dinámica. Detección de fallos y ejecución de algoritmos de recuperación.

- Actividad concurrente de los componentes.
 - SO multitarea.
 - Utilizar servicios de comunicación, sincronización y planificación.
- Necesidad servicios de temporización.
 - Necesidad reloj de tiempo real.
- Interacción hardware.
 - Necesidad gestionar soporte físico a bajo nivel y codificarse a alto nivel.
- Eficiencia frente a predictibilidad.
 - El sistema debe ser predecible y puede que no sea necesaria la eficiencia para tal fin.

2.PROGRAMACIÓN PARA APLICACIONES STR

2.1 Introducción

COMPUTADOR:

Def: Máquina programable de propósito general que procesa datos. Puede verse como una superposición de máquinas virtuales, correspondiendo cada una de ellas a un nivel de detalle con su lenguaje y escritura.

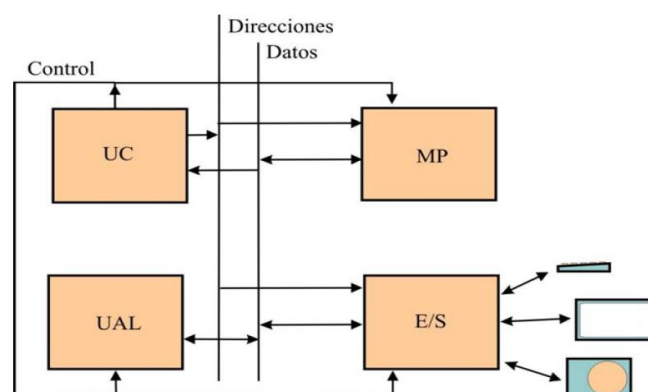
Niveles de máquina:

Máquina Simbólica
Máquina Operativa
Máquina Convencional
Micromáquina
Circuitos Lógicos
Circuitos Físicos
Dispositivos

2.2 Máquina convencional:

Lenguaje máquina (codificación binaria, programa almacenado en memoria, instrucciones muy simples).

Arquitectura Von Neumann: **MP:** Contiene datos o instrucciones, datos almacenados en palabras, cada una accesible por su dirección. **UC:** Lee e interpreta instrucciones. **UAL:** Circuitos lógicos para ejecutar instrucciones. **U E/S:** Comunicacino con el exterior.



Máquina operativa: Maquina convencional + SO. Lenguaje máquina + SO.

Máquina simbólica: Lenguaje de alto nivel y modelo de máquina que implica.

Registros:

Contienen datos. Acceso diferente a MP. Pueden ser:

- Aritméticos: Operandos y resultados intermedios
- Registros E/S. Intercambian datos con periféricos.
- Direccionamiento. Direcciones para MP.
 - Contador de programa (CP). Define la instrucción actual.
 - Puntero de pila (PP). Define la cima de la pila. Apunta a una palabra dentro de una zona de memoria reservada.

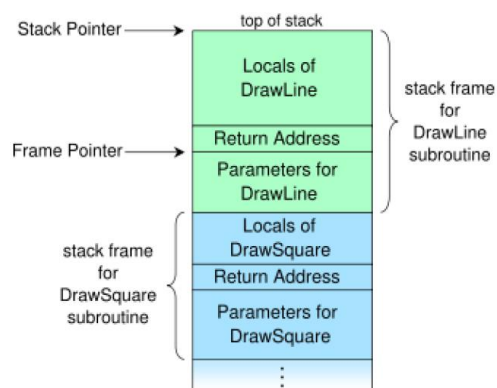
PILA DEL SISTEMA.

Una pila de llamadas es una estructura dinámica de datos LIFO (“Last In First Out”) que almacena información sobre las subrutinas activas de un programa de ordenador. La principal razón por la que se usa es para seguir el curso del punto al cual cada subrutina activa debe retornar el control cuando termine de ejecutarse, es decir, su principal función es almacenar la dirección de retorno. Aparte, también puede usarse para almacenar datos locales, pasar parámetros, etc. En general, hay una pila asociada a cada tarea o hilo de un proceso.

La pila del sistema se divide en registros de activación (“stack frames”) donde cada uno corresponde a una llamada a una subrutina activa, es decir, una subrutina que no ha terminado con un retorno a quién la llamó. El otro elemento fundamental es el puntero de pila (“stack pointer”) que apunta a la posición del último elemento que entró en la pila, es decir, el que debe ejecutarse ahora. En otras palabras, el valor del puntero de pila es el límite actual de la pila.

Cuando un programa llama a una subrutina, el programa que llama empuja (“push”) la dirección de retorno, es decir: si antes el puntero de pila apuntaba a la posición 0, ahora apunta a la posición 1 que es donde está la subrutina llamada. Cuando la subrutina finaliza, retira (“pop”) la dirección de retorno de la pila y transfiere el control a esa dirección, volviendo hacia atrás, por lo que el puntero de pila apunta de nuevo a la posición 0.

En este ejemplo, la subrutina “dibuja cuadrado” llama a la subrutina “dibuja línea” que es la subrutina activa. Cuando se la llama, en la pila se guardan los parámetros de dicha subrutina y la dirección de retorno de la subrutina que la invocó, es decir, a donde debe apuntar el puntero de pila una vez que se retire “dibuja línea”. De este modo, al retirar “Locals of draw line” se retorna a “locals of Draw Square”



CONCURRENCIA Y REGISTROS.

Def: Ejecutar varias tareas en paralelo. Con un solo procesador es posible simularse:

- Dedicando memoria separada a tareas diferentes.
- Multiplexando el acceso a la máquina convencional:
 - Conservar una copia de los registros relevantes (estado de la máquina) para las tareas que no se están ejecutando.
 - Cambio de contexto. Salvamos el estado actual y recuperamos el estado de la tarea que pasa a utilizar la máquina

Especialmente evidente para el contador de programa y el puntero de pila. Sin pilas separadas no hay secuencias de llamadas a función separadas.

2.3 Máquina Operativa

SISTEMA OPERATIVO.

Puede verse como una máquina virtual “sobre” la máquina convencional. Se encuentra dentro del nivel de máquina operativa. Realiza funciones complejas que el “hardware” real de la máquina no puede hacer, ofreciendo soluciones de altonivel a problemas complejos, opciones de multitareas.

Además, **adminstras los recursos de la computadora**, tiempo de acceso. Multiplexa los recursos en el tiempo

Implementación habitual:

- Parte crítica (núcleo) asociada a un nivel de funcionamiento “privilegiado” del hardware.
- Llamadas implementadas mediante instrucciones de máquinas especiales
- Instrucciones cambian el contexto y el nivel de privilegio controladamente.

Opciones de implementación:

- Construir un sistema de propósito específico.
- Usar lenguaje de alto nivel y su sistema de soporte en tiempo de ejecución (ADA)
- Servicios de SO + lenguaje convencional.

Deben proporcionar: Servicios para concurrencia (multitarea), planificación, temporización, comunicación y sincronización.

SO & SRT:

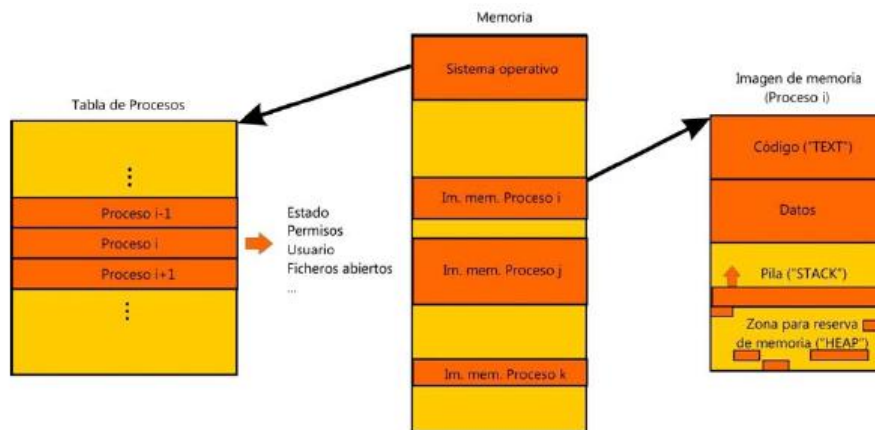
- Objetivos de diseño: Eficiencia del uso del computador .Reparto equilibrado de la máquina entra apps y usuario
- Objetivos de los sistemas informáticos de tiempo real:
 - Cumplir especificaciones temporales.
 - Mayor prioridad a tareas urgentes.
 - Las tareas no se tratan de manera equitativa.
 - La eficiencia no es el fin principal.

PROCESO:

Es una unidad de concurrencia, no es un programa, es un programa en ejecución. El mismo programa puede utilizarse en varios procesos.

Componentes:

- Imagen de memoria:
 - Creada a partir del programa ejecutable
 - Contiene instrucciones (programa) y datos.
- Entrada en la tabla de procesos:
 - Estado de ejecución
 - Recursos utilizados
 - Datos administrativos



INTERCAMBIO DE MEMORIA:

Las imágenes de memoria no siempre están en la MP. Se vuelca en disco imágenes de procesos inactivos, permitiendo así mantener más procesos de los que caben en MP.

INTERFAZ DE USUARIO:

Intérprete de comandos "Shell" en modo texto. Interfaz gráfica de usuario. Hay que tener en cuenta que son el aspecto externo del s.o, no debiendo identificarse con él. Normalmente son procesos que utilizan los mismos servicios que el usuario. Puede haber varias opciones para escoger.

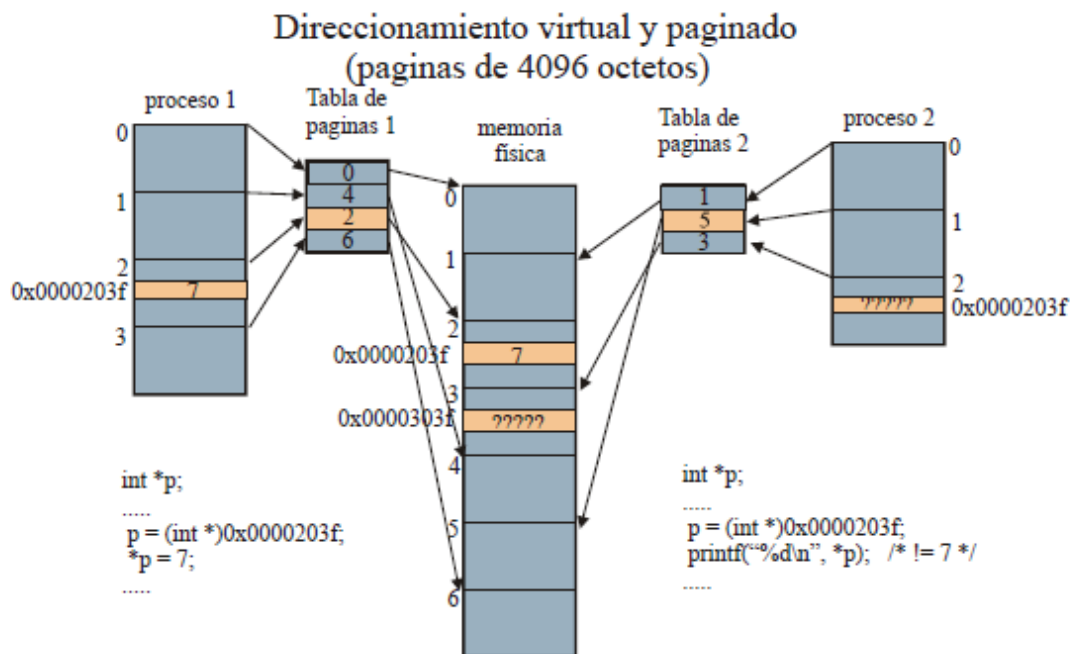
DIRECCIONAMIENTO VIRTUAL Y PAGINADO

El direccionamiento virtual consiste en asignarle a cada proceso un espacio de direccionamiento propio, siendo necesario que se traduzcan después las instrucciones en dicho espacio en direcciones reales. Cuando hacemos un direccionamiento paginado, lo que hacemos es dividir la memoria disponible en páginas de una longitud igual a una cantidad que debe ser potencia de 2.

Para traducir en el direccionamiento virtual y paginado, se utilizan las tablas de página. Cada tabla de página tiene una entrada para cada página virtual, y almacena los números de páginas reales. Cuando se invoca una instrucción en una memoria virtual, esta se traduce en la tabla de páginas para hacérsela llegar a la memoria real:

Los problemas que soluciona la memoria virtual y el direccionamiento paginado son:

1. La gestión de memoria, es decir, buscar sitio para un proceso.
2. La protección de datos, es decir, crear “barreras” entre procesos.
3. La reubicación, es decir, evitar que un proceso cambie de dirección.



SOFTWARE BAJO NIVEL

- Arquitectura E/S.
 - Espacio de direcciones propio, con instrucciones especializadas.
 - Espacio de direcciones compartido con memoria.
- Dispositivo físico. Capaz de controlar uno o varios dispositivos E/S
- Comunicación con la CPU.
 - Registros. Transmiten información: Control, Estado, Datos
 - Interrupciones. Sincronización mediante una señal dedicada.
- Necesidad de sincronización. Acciones E/S no son inmediatas.

- Métodos sincronización.
 - Espera activa o polling. CPU muestrea repetidamente el estado de controlador hasta detectar la condición. Poco eficiente. Debe incluir retraso para ser admisible.
 - Interrupciones. Controlador envía interrupción a la CPE. En respuesta se ejecuta una acción. Más eficiente pero difícil.
 - Acceso a memoria (DMA). Controlador almacena los datos en memoria. Requiere interrupciones de sincronización. Lo más eficiente.

Tratamiento de interrupciones:

- Una interrupción activa una rutina de servicio. Salvamos el estado del procesador y recuperamos luego.
 - Identificación de la causa.
 - Vector de interrupción.
 - Es posible conocer el número de interrupción mediante un registro de estado.
 - Consulta: Necesario cada dispositivo para identificar la causa.
 - Procedimiento mixto:
 - Varias causas por señal de interrupción. Un vector por señal, y consulta para identificar la causa.
 - Varias causas por dispositivos: Vector que identifica el dispositivo y consulta para identificar la causa.
 - Prioridad. Diferentes niveles de prioridad
 - Máscaras de interrupción: Una interrupción puede ignorarse temporalmente.
- Niveles:
- Habilitación global de interrupciones (CPU)
 - Máscara de señales individuales (CPU/Contr.)
 - Máscaras de causas individuales (Bits de control de dispositivos).
- SOFTWARE A BAJO NIVEL EN C**
- Acceso a registros fácil si el espacio de direcciones es compartido con memoria.
 - Manejo de interrupción. Necesario resolver sincronización.
 - Inclusión de código máquina. (ASM)

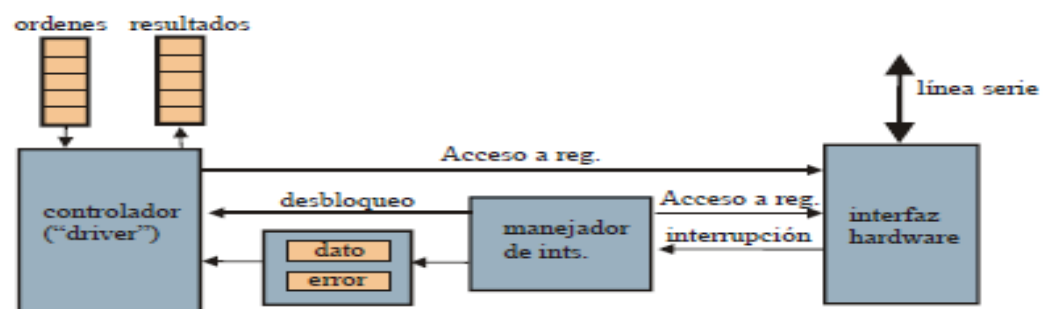
SOFTWARE DE E/S

- Funciones del software E/S
 - Interfaz independiente del dispositivo
 - Tratamiento de errores
 - Sincronización con actividades E/S
 - Gestión de “buffers” intermedios
- Niveles del software E/S
 - **Nivel Usuario**
 - Bibliotecas de E/S y procesos del sistema
 - Funciones:
 - Ofrecer interfaz requerido por el lenguaje de programación
 - Sincronización con operación de E/S
 - Formateo de datos.
 - **Nivel Kernel**
 - Operaciones comunes y una interfaz única.
 - Funciones:
 - Correspondencia entre nombres y dispositivos
 - Gestión de buffers para tratamiento de datos
 - Tratamiento errores
 - Gestión de acceso a dispositivos
 - Ocultar diferencias entre dispositivos.
 - **Controladores de dispositivos: “Drivers”**
 - Específico para cada dispositivo
 - Ofrecen al SO una determinada interfaz
 - Funciones:
 - Desarrollar peticiones en forma de secuencia de comandos
 - Se bloquea a la espera de interrupciones
 - Tratamiento de errores
 - Dificultades: Peculiaridades interacciones con el SO. Recursos y condiciones de funcionamiento pueden cambiar.

- **Manejadores de interrupción**

- Pasos necesarios:

- Guardar el contexto del procesador actual y preparar el contexto del manejador (pila, tabla de páginas...).
 - Operaciones ligadas al hardware. (reconocer interrupción, habilitar...)
 - Procedimiento de servicios de interrupciones
 - Escoger nuevo proceso y prepara nuevo contexto
 - Pasar el control a nuevo proceso



2.4 Máquina Simbólica

Máquina virtual sobre la maquina operativa que “entiende” lenguajes de alto nivel.

Opciones de implementación:

- Interpretación. Un intérprete lee el código fuente como datos y lo ejecuta
 - El intérprete siempre se está ejecutando cuando se ejecuta el programa.
 - Muy ineficiente
- Compilación. Un compilador traduce el código fuente a código máquina.
 - Una vez compilado, el código ejecutable puede ejecutarse tantas veces como sea necesario.
 - Tipos:
 - Compilación Separada.
 - Necesaria para modularidad. Construir el programa a partir de varios módulos que se compilan separadamente.
 - Fases
 - Compilación a objetos. (Módulos que han cambiado)
 - Traduce código fuente para generar código objeto reubicable.
 - Referencias externas indefinidas.
 - Código objeto puede adaptarse más tarde a direcciones arbitrarias.
 - Edición de enlace (todos los módulos)
 - Asignación definitiva de direcciones.
 - Reubicación del código de todos los módulos
 - Resolución de referencias externas.
 - Conceptos lenguaje C. A programar!

- Punteros:

```
#include <stdio.h>
int main()
{
    int a = 10, *p;    /* "a" como Entero y "p" como puntero */
    p = &a;    /* Asigna la dirección de "a" a "p" */
    printf("\nEl valor de A : %d", a);    /* Contenido de "a" */
    printf("\nLa dirección de A : 0x%x", &a); /* Dirección de "a" */
    printf("\nEl valor de P : 0x%x", p);    /* Contenido de "p" */
    printf("\nLa dirección de P : 0x%x", &p); /* Dirección de "p" */
    printf("\nEl contenido de P es : %d", *p); /* Puntero de &a */
    return 0;
}
```


- Cadena de caracteres:
 - **Strlen**: Devuelve numero de caracteres de la cadena
 - **Strcopy** : Copia la cadena. Devuelve un puntero a la cadena destino
 - **Strcat** : Añade la cadena origen al final de la cadena destino.
 - **Strcmp**: Compara cadena. Devuelve 0 si son iguales.
 - **Scanf** : lee con formato de entrada estándar
 - **Fscanf** : igual, pero lee de un archivo.
 - **Sscanf** : Lee de una cadena de caracteres (útil convertir argumentos)
- Ficheros : Diapositivas Tema2

NORMAS POSIX PARA STR

- Definidas por IEEE.
- Idea: Hacer posible la portabilidad entre diferentes SO
- Implementadas como una biblioteca.
- Especifican sobre todo requerimientos funcionales, no de tiempos de respuesta.
- Soportar estas normas no implica que el sistema operativo sea apropiado para tiempo real.

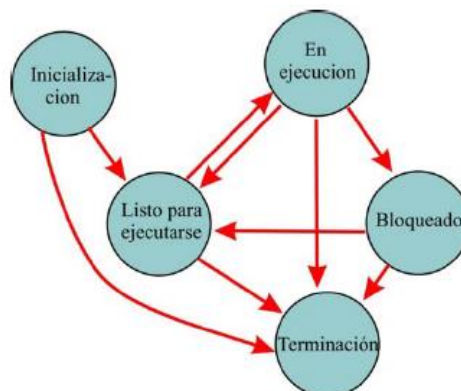
3. CONCURRENCIA

3.1 Introducción

Al tener más tareas que procesadores tenemos que multiplexar el tiempo de acceso. Soluciones:

- **Programa único. Ejecutivo cíclico.**
 - Programador reparte el tiempo de CPU entre las tareas
 - Las tareas son funciones invocadas por un programa único
- **Dos niveles de prioridad:**
 - Alta prioridad ("foreground"): Tareas críticas
 - Baja prioridad ("background"): Tareas no críticas
 - Programador reparte el tiempo de CPU en cada nivel
- **Sistema multitarea. Programación concurrente**
 - Tareas definidas separadamente en programas independientes.
 - Múltiples niveles de prioridad. Tiempo se asigna indirectamente mediante prioridad y algoritmo de planificación.
 - Ventajas: Modularidad, facilidad de modificación, escalabilidad, portabilidad, protección.
 - Inconvenientes: Menos eficientes, necesitamos servicios comunicación y sincronización.

En un esquema de programación concurrente las diversas tareas se describen por programas separados, y el tiempo del procesador se reparte indirectamente asignado prioridades. El método de planificación empleado decide en cada instante a que tarea le corresponde ejecutarse teniendo en cuenta su prioridad. Cada tarea posee el diagrama de estados que se muestra a continuación.



Inicialización: La tarea espera a disponer de los recursos necesarios para su funcionamiento.

En cada instante existirá un número de tareas **listas para ejecutarse**, que piden acceso al computador.

Solo una tarea podrá estar en **ejecución**.

Las tareas que esperan algún evento permanecerán en estado **bloqueado** hasta que dicho evento ocurra.

En el estado de **terminación** la tarea espera a que se realice alguna operación final, como por ejemplo la entrega de resultado.

3.2 Concurrency en POSIX

2 opciones, procesos e hilos:

- **Procesos:** Una unidad de actividad que se caracteriza por la ejecución de una secuencia de instrucciones, un estado actual, y un conjunto de recursos del sistema asociado.
 - Tienen espacios de direccionamiento virtual separados y su propia tabla de página.
 - Tienen una entrada completa en la tabla de procesos.
 - Se asegura protección.
 - No es inmediato compartir memoria.
- **Hilos:** Son parte de un proceso, comparten espacio de direcciones, pero se crea una pila nueva y solo los datos globales son accesibles desde todos, porque el resto de datos son locales. Es menor costoso crear hilos que procesos nuevos, pero no tenemos la protección entre hilos.

3.2.1 Creación de procesos.

CREACIÓN DE UN PROCESO HIJO A PARTIR DE UN PROCESO PADRE

1. Añadimos dos librerías:

#include<sys/types.h> → Incluye el tipo pid_t

#include<unistd.h> → Incluye la función fork y la función execl

2. Llamada a fork:

- a. Creamos un identificador de nuestro proceso hijo:

pid_t hijo; //pid_t es un tipo de variable que identifica al proceso con un numero

- b. Llamamos a fork:

hijo = fork(void);

La función fork tiene la siguiente estructura:

pid_t fork (void)

Crea el proceso hijo a partir del proceso padre

Proseguimos ahora con la **activación del proceso deseado**:

3. Creamos la división padre/hijo:

```
if (! hijo){código del hijo}
```

```
Else{código del padre}
```

4. Para que el hijo sea el que corra el proceso deseado, usamos:

```
execl("./nombredelejecutable","argumento1","argumento2", ..., NULL);
```

Esto se incluye en el código del hijo, claro está.

La función execl tiene la siguiente estructura:

```
pid_t execl (const char*ejecutable, const char*arg0, ... , NULL)
```

Siendo el primer valor que se le pasa un puntero al ejecutable que se va a ejecutar, en general mediante el método `./nombredelejecutable`

El resto de parámetros son los argumentos que se le pasan a dicho ejecutable. El último argumento que se le pasa a execl SIEMPRE será NULL.

Explicación: El primer paso, la función fork crea una copia del proceso padre. Si esta función falla, devuelve -1. Si no, devuelve dos valores: en el proceso padre devuelve el valor del PID (identificador) de proceso hijo; en el proceso hijo, devuelve el valor 0. Estos valores los asignamos al identificador que hemos creado previamente.

Esto explica cómo se estructura el punto 3: el if depende del valor del identificador. Si estamos el proceso hijo su valor será cero, por ello usamos |hijo|. Cualquier otro valor nos llevaría al código del proceso padre.

- Sin exec:

```
(...)  
pid_t hijo;  
hijo = fork();  
if(hijo == 0) { /* Estoy en el hijo */  
    funcion_hijo(...); /* padre e hijo comparten ejecutable */  
    exit(0); /* exit termina el proceso hijo */  
}  
/* Aquí sigue el padre y nunca entra el hijo */  
(...)
```

- Con exec:

```
(...)  
pid_t hijo;  
hijo = fork();  
if(hijo == 0) { /* Estoy en el hijo */  
    execl("./hijo", "a", "b", NULL); /* hijo con ejecutable "./hijo" */  
    exit(0); /* (por si fracasa execl) */  
}  
/* Aquí sigue el padre y nunca entra el hijo */  
(...)
```

TERMINACIÓN Y ESPERA DE PROCESOS

TERMINACIÓN

- Por iniciativa propia:
 - **return** desde **main**
 - **exit** o **_exit** desde cualquier función
- Proceso abortado: Normalmente la llegada de una señal que no se trata.

ESPERA

- La espera incluye la recogida del resultado
- Solo el padre puede esperar hacerlo
- Llamadas:
 - **wait**: Espera a cualquier hijo
 - **waitpid**: Puede esperar a un hijo concreto o no esperar.
- El entero recibido no es exactamente el resultado. Usar macros para recuperarlo.

Lo primero que tenemos que hacer es añadir dos librerías más:

#include <stdlib.h> -> Incluye la función exit.

#include <sys/wait.h> -> Incluye wait/waitpid

La forma normal de finalizar un proceso es usar la función exit -> **exit(1)**;

La función exit es de la forma:

void exit (int status)

int status es un valor al que tienen acceso diversos comandos

Como ya sabemos, para saber si un proceso hijo a terminado o no, usamos **wait / waitpid**. Son muy útiles porque liberan completamente los recursos del ordenador que el proceso hijo estaba usando. **Waitpid** es más flexible. En caso de usar alguno de ambos, tenemos que inicializar una variable int donde se guardará el valor que devuelve exit;

WAIT vs **WAITPID**

```
(...)  
pid_t resw, hijo; /* resultado de wait y pid del hijo */  
int res; /* resultado del hijo mas info. adicional */  
hijo = fork();  
if(hijo == 0) {  
    funcion_hijo(...);  
}  
(el padre sigue ejecutándose)  
resw = wait(&res); /* Espera (de cualquier hijo, pero sólo hay uno) */  
if(resw != -1) imprime_res(res, resw); /* Ya veremos qué contiene */  
else printf("Error en wait: %s\n", strerror(errno));  
(...)
```

```
(...)  
pid_t resw, hijo; /* resultado de waitpid pid del hijo */  
int res; /* resultado del hijo mas info. adicional */  
hijo = fork();  
if(hijo == 0) {  
    funcion_hijo(...);  
}  
(el padre sigue ejecutándose)  
do {  
    salir = 1;  
    resw = waitpid(hijo, &res, WNOHANG); /* Mira si el hijo ha acabado */  
    if(resw > 0) imprime_res(res, resw); /* Ya veremos qué contiene */  
    else if(resw == 0) {  
        salir = 0;  
        hacer_algo(); /* El padre aprovecha el tiempo... */  
    } else printf("Error en waitpid: %s\n", strerror(errno));  
} while(salir == 0);
```

Función `imprime_res`:

```
void imprime_res(int result, pid_t id) {
    if(WIFEXITED(result)) {
        printf("%ld acabó normalmente con estado %d\n",
            (long)id, WEXITSTATUS(result));
    }
    else {
        if(WIFSIGNALED(estados)) {
            printf("%ld muerto por señal\n", (long)id);
        } /* if */
        else {
            /* Otras posibilidades */
        } /* else */
    } /* if */
}
```

IDENTIFICACIÓN DE PROCESOS:

1. **Getpid:** Esta función devuelve el identificador del proceso que se está ejecutando.

`pid_t getpid()`

2. **Getppid:** Esta función devuelve el identificador del proceso padre del proceso que se está ejecutando.

`pid_t getpid()`

Ej: `printf("El proceso actual es el número %d", (int)getpid());`

Usos del valor de la variable de estado de exit y opciones de `waitpid`

Nosotros usaremos tres macros con la variable "estado" que almacena el valor que devuelve exit. Estas son:

- **WIFEXITED(estados):** este comando da un número, que será distinto de 0 siempre que se saliera del proceso a través de exit de forma normal.
- **WIFSIGNALED(estados):** si su valor es 1, indica que se salió por culpa de una señal, aunque otros números indican más posibilidades.
- **WEXITSTATUS(estados):** recupera los 8 bits menos significativos del valor que se pasó a exit, es decir, si escribimos `exit(2)`, nos daría 2.

Las opciones de `waitpid` son el último argumento de la función, que normalmente escribiremos como 0, aunque podríamos poner:

- **WNOHANG:** evita que el padre espere a ningún hijo. En este caso, se devuelve 0 si existen hijos activos, y si no devolverá -1. Permite ejecutar otra función al padre en vez de bloquearse.
- **WUNTRACED:** el padre recibe información adicional del hijo si se alguna de las siguientes señales: `SIGTTIN`, `SIGTTOU`, `SIGSSTP` o `SIGTSTOP`.

3.2.2 Hilos ("threads")

CONCEPTO

- Todo hilo está asociado a un proceso.
 - Comparten espacio de direcciones
 - Variables globales accesibles desde todos ellos
 - Recursos compartidos
 - No se duplican todos los datos de la tabla de procesos (solo imprescindible)
 - No se duplican la imagen de memoria. Se crea una **pila nueva**
- Ventajas
 - Creación menos costosa que proceso
 - Conmutación menos costosa
 - Facilidad para compartir memoria
- Desventaja: No existe protección entre hilos

CREACIÓN DE UN PROCESO HIJO A PARTIR DE UN PROCESO PADRE:

Añadimos la librería:

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

1. Creamos punteros a la rutina de arranque

Por cada hilo que vayamos a crear, tenemos que crear un puntero antes de definir el hilo principal:

```
void *hilo(void *arg);
```

En vez de "hilo", podemos escribir lo que queramos. Después ya se crea el hilo principal:

```
void main(void)
```

```
{
```

```
...
```

```
}
```

2. Creamos el identificador del hilo, los atributos y el argumento de rutina

Son imprescindibles el identificador del hilo y el argumento para la rutina, los atributos, sin embargo no, porque siempre podemos dejar los que vienen por defecto:

```
pthread_t thr1;  
pthread_attr_t attr1;  
int i;
```

3. Crear el hilo en sí

```
pthread_create(&thr1, NULL, hilo, (void *)&i);
```

La función `pthread_create` es de la forma:


```
int      pthread_create (pthread_t *thread, const pthread_attr_t *attr,  
                        void *(*start_routine)(void*), void *arg)
```

El entero que devuelve es distinto de 0 si ha habido algún problema. De esta manera podemos detectar errores en la creación del hilo.

El primer argumento es el identificador del hilo, el segundo es la variable que contienen los argumentos del hilo, (NULL, para ver los que vienen por defecto). El tercer argumento es el que indica que hilo va a crearse. El cuarto argumento es un puntero genérico que recibe el hilo nuevo.

4. Definir el hilo.

Un hilo se define de forma parecida a las funciones, después de cerrar el hilo principal main.

```
#include<pthread.h>  
void *hilo(void *);  
void main(void)  
{  
    pthread_t thr1;  
    pthread_attr_t attr1;  
    int i;  
    pthread_create(&thr1, NULL, hilo, (void *)&i);  
}  
void *hilo(void *pi)  Definición en sí del hilo que se crea  
{  
    <código del hilo>  
}
```


TERMINACIÓN Y ESPERA DE HILOS

- **TERMINACIÓN CON RETORNO**

1. Usando función return en cualquier momento del código

```
return (void *)p;
```

2. Usando **pthread_exit**:

```
Pthread_exit((void *)p);
```

Ambos usan (void *) porque recordemos que los hilos devuelven ese tipo de puntero. Una vez acabado el hilo de esta manera, puede recogerse dicho valor usando:

```
pthread_join (pthread_t *thread, void **valor)
```

El primer argumento de entrada es el identificador del hilo al que se recoge la información, y el segundo es la variable donde se almacenará dicho valor (si no se desea almacenar, escribimos NULL).

- **TERMINACIÓN SIN RETORNO**

Una vez que se llega al final de la rutina del hilo, este termina. Para evitar que queden “restos” de los hilos en el sistema, es conveniente invocar a:

```
int pthread_detach (pthread_t thread)
```

Lo único que hace es que, al acabar el hilo, todos los recursos que utilizaba queden libres. El argumento que recibe es el identificador del hilo del que se desean liberar los recursos al acabar.

- **TERMINACIÓN DESDE OTRO HILO**

Hay que usar la función:

```
int pthread_cancel (pthread_t thread)
```

Cuyo único argumento de entrada es el identificador del hilo que se desea terminar. No siempre es posible cancelar el hilo, pues está ligado a dos conceptos relativos a la cancelabilidad del hilo:

- **Estado:** Puede tomar dos valores, “disabled” y “enabled”. En el primero no se puede cancelar, en el segundo se produce en función del tipo de cancelabilidad.
- **Tipo de cancelabilidad:** Define si la cancelación se produce de forma “síncrona” o “asíncrona”. En el primero solo se puede cancelar en determinados puntos de código llamados “puntos de cancelación”, en el segundo se puede cancelar en cualquier instante.

Por defecto, todos los hilos comienzan con estado “enabled” y tipo “síncrono”. Para modificarlo usamos dos funciones:

1. **Para cambiar el estado:** Usamos:

int pthread_setcancelstate (int state, int *oldstate).

El primer argumento puede ser "PTHREAD_CANCEL_ENABLE" (estado enabled) o "PTHREAD_CANCEL_DISABLE" (estado disabled). El entero que devuelve la función es el estado anterior, a menos que se use NULL en el segundo argumento de entrada.

2. **Para cambiar el tipo:**

int pthread_setcanceltype (int type, int *oldtype)

El primer argumento puede ser "PTHREAD_CANCEL_DEFERRED" (tipo síncrono) o "PTHREAD_CANCEL_ASYNCRONOUS" (tipo asíncrono). El entero que devuelve la función es el estado anterior, a menos que se use NULL en el segundo argumento de entrada.

Para crear un punto de cancelación donde las peticiones de cancelación de hilos síncronos se cumplan, se usa la función:

void pthread_testcancel (void)

Simplemente invocándola se crea dicho punto, y en ese momento se comprueba si hay alguna petición de cancelación pendiente, y si es así, se resuelve.

COMPATIBILIDAD Y DETECCIÓN DE ERRORES

Errno ya no es una var global única. Es un macro que ofrece resultados diferentes para cada hilo.

Las llamadas **pthread_xx** no tienen por qué usar **errno**; Devuelve el código de error como resultado.

Llamadas seguras en hilos: "**thread-safe**":

- Solo las llamadas "thread-safe" pueden invocarse concurrentemente desde hilos distintos
- Llamadas no seguras:
 - Habitualmente usan variables globales y podemos tener problemas
 - Pueden utilizarse si no se invocan de manera concurrente

4. SEÑALES POSIX

4.1 Introducción

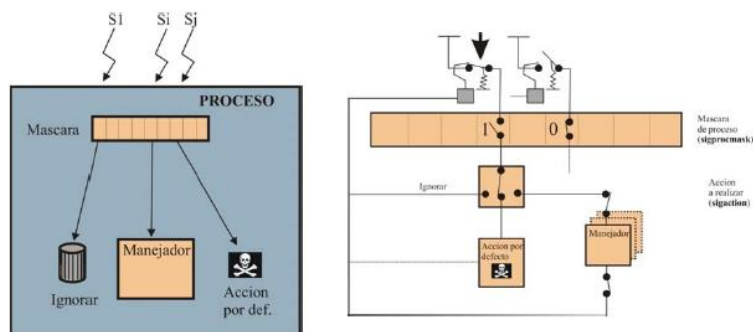
Las señales son un medio de comunicación entre procesos e hilos. Básicamente se utilizan para notificar eventos excepcionales, es decir, asíncronos, pues no sabemos cuándo pueden ocurrir. Todo su código está incluido en la librería <signal.h>, es decir, que a la hora de usar señales siempre empezaremos con:

#include<signal.h>

Cuando se genera una señal, se encuentra pendiente hasta que le “llega” a un proceso, que actuará en consecuencia. Sin embargo, la llegada de señales depende de la máscara del proceso. La máscara del proceso puede entenderse como un número formado por bits, uno para cada señal, y en función de si el bit está a 0 ó a 1, la señal estará o no bloqueada.

Cuando una señal está bloqueada, esa señal no llega al proceso. Si por el contrario la señal no está bloqueada, una vez que se genera, pueden ocurrir varias cosas:

- El proceso es **terminado**.
- La señal es **ignorada**.
- El proceso se **detiene**.
- El proceso **continúa** si había sido detenido.
- Una vez que llega la señal, la recibe un **manejador** (“handler”), una especie de función.



Las señales tienen prioridad desde **SIGRTMIN** a **SIGRTMAX**. Las señales se encolan hasta un máximo de **SIGQUEUE_MAX**.

Algunas señales **POSIX 1003.1a**

- **SIGFPE** : Errores de cálculo
- **SIGKILL** : Terminación incondicional del proceso “kill -q<pid>”
- **SIGTERM** : Terminación del proceso “kill<pid>” – Línea de comandos
- **SIGINT** : Señal de interrupción recibida por el terminal (teclado) “ctrl + c”
- **SIGQUIT** : Señal de terminación recibida por el terminal
- **SIGUSR1, SIGUSR2** : De propósito general, reservado para el usuario.
- **SIGALARM** : Funciones de tiempos. Temporizador

Siempre que vayamos a usar señales, hay que hacer dos cosas:

1. Programar la máscara del proceso. Paso indistinto del tipo de señal.
2. Especificar qué se hace cuando “llega” la señal. Diferente en 1003.1a y 1003.1b

4.2 Programación de la máscara del proceso

Para programar la máscara del proceso haremos uso de “conjunto de señales” (también son máscaras) que están definidas por el tipo de variable **sigset_t**. En estos conjuntos añadiremos o eliminaremos señales, que nos servirán para definir la máscara del proceso:

- Para añadir todas las señales del conjunto:

int sigfillset (sigset_t *pset)

El argumento de entrada es un puntero al conjunto al que queremos añadir todas las señales.

Ejemplo:

```
sigset_t conjunto;  
sigfillset(&conjunto);
```

- Para eliminar todas las señales del conjunto:

int sigemptyset (sigset_t *pset). Idéntico al funcionamiento anterior

- Para añadir una señal concreta

int sigaddset (sigset_t *pset, int sig)

El primer argumento es de nuevo un puntero al conjunto al que vamos a añadirle la señal, mientras que el segundo es el nombre de la señal que queremos añadir (es una macro). Por ejemplo, para bloquear SIGALRM.

```
sigset_t conjunto;  
sigaddset(&conjunto,SIGALRM);
```

- Para eliminar una señal concreta

int sigdelset (sigset_t *pset, int sig)

4.2.1 Modificación Máscara

Todas estas funciones devuelven 0 cuando todo va bien, y un -1 si hay algún error.

Una vez definido los conjuntos de señales con las que queremos modificar en nuestra máscara del proceso, usaremos la siguiente función para modificarla:

Int sigprocmask (int how, const sigset_t *set, sigset_t *oset)

- **Primer argumento:** Es el que modifica en sí la máscara. Podemos escribir tres órdenes
 - **SIG_BLOCK** : Las señales bloqueadas en la máscara de señales son las que ya tiene bloqueadas más las que están incluidas en el conjunto que se coloca en el segundo argumento de sigprocmask.
 - **SIG_UNBLOCK** : Se desbloquean las señales que están incluidas en el conjunto que se coloca en el segundo argumento de sigprocmask.
 - **SIG_SETMASK** : Se borra la máscara actual y la nueva máscara solo tiene bloqueadas las señales incluidas en el conjunto que se coloca en el segundo argumento de sigprocmask.
- **Segundo argumento:** Es el conjunto cuyas señales se toman como referencia para las órdenes del primer argumento. Si ponemos NULL, no se modifica la máscara.
- **Tercer argumento:** En este conjunto se guarda la máscara del proceso actual antes de la modificación. Si se escribe NULL, no se guarda.

Esta función devuelve 0 si todo va bien y -1 para algún error.

Ejemplo: Desbloqueo de la señal SIGALARM:

```
sigset_t conjunto;  
sigemptyset(&conjunto);  
sigaddset(&conjunto, SIGALRM);  
sigprocmask(SIG_UNBLOCK, &conjunto, NULL);
```

Las señales **SIGKILL** y **SIGSTOP** son las únicas que NO pueden ser bloqueadas.

Como hemos dicho, la definición de la máscara es común en las señales 1003.1ª y 1003.1b, pero la programación de lo que se hace una vez llegan las señales no. Por eso vamos a explicarlo de forma separada.

4.3 Programación para señales POSIX 1003.1a

Lo primero que hay que hacer es definir una variable estructura "sigaction", que será utilizada en una función también llamada "sigaction". Esta estructura está definida de la siguiente forma:

```
struct sigaction {  
    void( *sa_handler) ();  
    sigset_t sa_mask;  
    int sa_flags;  
};
```

- Primer elemento: Hay tres posibles opciones:
 1. **SIG_DFL** : Se ejecuta la acción por defecto para la señal asociada (las señales definidas tienen una acción asociada).
 2. **SIG_IGN** : Se ignora la señal que ha llegado.
 3. **El nombre de una función manejadora previamente definida.**
- Segundo elemento : Es un conjunto que se modifica como tal, y que en caso de usar un manejador se utiliza como máscara del proceso.
- Tercer elemento : En 1003.1a, simplemente ponemos un 0.

Una vez hemos definida nuestra estructura **sigaction**, tenemos que utilizar la función **sigaction** para asociarle una señal.

Int sigaction (int sig, struct sigaction *act, struct sigaction *oact);

El primer argumento es tan solo el nombre de la señal que va a disparar la acción definida en la estructura sigaction que se coloca en el segunda argumento. El último argumento es simplemente la acción que estaba previamente programada, almacenada también en una estructura sigaction. Puede ponerse NULL.

Para comprender bien todo esto, vemos el ejemplo, en el que supondremos que hay previamente un manejador definido como "manejador1":

```
struct sigaction ejemplo;  
ejemplo.sa_handler=manejador1;  
ejemplo.sa_flags=0;  
sigemptyset(&ejemplo.sa_mask);  
sigaction(SIGALRM,&ejemplo,NULL);
```

Cuando la señal SIGALARM se activa, al estar asociada al manejador, este se disparará y todo lo que esté definido en él se ejecutará.

TRATAMIENTO DE SEÑALES:

- **Tratamiento asíncrono:**
 - Mecanismo previsto originalmente
 - Se programa un manejador como acción a realizar.
 - Al recibir la señal, el manejador se ejecuta, interrumpiendo el proceso temporalmente.
 - Posibles problemas con llamadas no “async-safe”
 - Interrumpe las esperas (error EINTR)

Señales POSIX 1003.1a: Tratamiento asíncrono

Ejemplo: Manejador para SIGTERM y SIGINT

```
(...) /* Desenmascarar ambas señales */
#include <signal.h> /* Antes hay que hacer un conjunto */
(...) sigemptyset(&sen);
void manej(int s) { .... } sigaddset(&sen, SIGTERM);
(...) sigaddset(&sen, SIGINT);
struct sigaction acc; sigprocmask(SIG_UNBLOCK, &sen, NULL);
sigset_t sen; /* A partir de ahora se recibirán señales y
entrará manej asincrónamente */
(...)

/* Programar manejador */
acc.sa_flags = 0;
acc.sa_handler = manej;
sigemptyset(&acc.sa_mask);
sigaction(SIGTERM, &acc, NULL);
sigaction(SIGINT, &acc, NULL);
```

- **Tratamiento síncrono:**
 - El proceso espera a la llegada de la señal.
 - No estaba previsto: Puede hacerse utilizando la interrupción de esperas por error EINTR.
 - Siempre ha de ejecutarse el manejador

Señales POSIX 1003.1a: Tratamiento síncrono

Ejemplo: Esperar SIGUSR1

```
(...) /* Desenmascarar SIGUSR1 */
#include <signal.h> sigemptyset(&sen);
(...) sigaddset(&sen, SIGUSR1);
void manej(int s) { .... } sigprocmask(SIG_UNBLOCK, &sen, NULL);
(...) /* Es necesario que entre manej para romper
la espera */
struct sigaction acc; (...)
sigset_t sen; /* Esperar SIGUSR1 */
int res; res = sleep(3600);
if(res != 0) {
    printf("Ha Llegado SIGUSR1\n");
    hacer_cuando_SIGUSR1(...);
} else {
    printf("Error: No llega SIGUSR1!!\n");
    hacer_trat_error(...);
}
...
```

- **Enviar señales**
 - Int kill (pid_t destino, int señal)

Ejemplo: Sobretiempo con señales

```
#include <unistd.h>
#include <signal.h>
int timeout; /* Flag de sobretiempo */
void maneja(int sign) {
    timeout = 1;
}
void main(void) {
    int fin; /* Flag de fin */
    struct sigaction action;
    sigset_t set;

    /* Programación de la acción a realizar */
    action.flags = 0;
    action.sa_handler = maneja;
    sigemptyset(&action.sa_mask);
    sigaction(SIGALRM, &action, NULL);

    /* Desbloqueo de SIGALRM */
    sigemptyset(&set); sigaddset(&set,
    SIGALRM);

    sigprocmask(SIG_UNBLOCK, &set, NULL);

    /* Después se programa el "despertador" */
    timeout = 0;
    alarm(20);
    fin = 0;

    /* Algoritmo con sobretiempo */
    while(!fin && !timeout) {
        fin = funcion_iteracion(...);
    }

    /* Analizar lo que ha sucedido */
    if(fin) alarm(0); /* Ya no debe saltar */
    if(timeout) {
        printf("Alarma! Sobretiempo!\n");
        < algoritmo alternativo >
    }
}
```

4.4 Programación para señales POSIX 1003.1b

La estructura sigaction se define ahora del siguiente modo:

```
struct sigaction {
    void( *sa_handler) ();
    sigset_t sa_mask;
    int sa_flags;
    void(*sa_sigaction) (int signo, siginfo_t *datos,void *extra);
};
```

- Primer elemento: NO se usará.
- Segundo elemento: Igual que 1003.1a
- Tercer elemento: hay que ponerla siempre a "SA_SIGINFO".
- Cuarto elemento: se utiliza exactamente igual que **sa_handler** en 1003.1a, la diferencia es que una vez se asigna un manejador automáticamente almacena el número de la señal en "signo", un puntero a una estructura tipo **siginfo_t** (que ahora definiremos) llamado "datos" en el que se almacenan datos adicionales, y por último un puntero a vacío llamado "extra" cuyo uso no está definido.

La estructura **siginfo_t** esta definida de la forma:

```
typedef struct {
    int si_signo;
    int si_code;
    union signal si_value;
} siginfo_t;
```

El tipo "union signal" se define así:

```
union signal {
    int sival_int;
    void * sival_ptr;
};
```

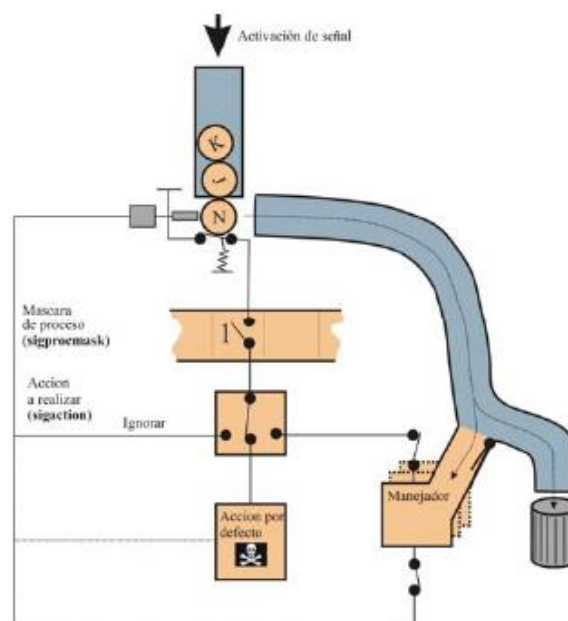

¿Qué es lo que ocurre?

1. El número de la señal que recibe el manejador ahora se encuentra en el entero “**signo**” y en el valor apuntado por “**datos->si_signo**”.
2. El puntero de la variable “**si_value**” no suele usarse. Ambos valores se le pasan al manejador a través de la función “**sigqueue**”.
3. El código de “**datos->si_code**” indica el motivo de la activación del manejador, que puede ser distinto de la activación de “**kill**”:
 - **SI_QUEUE**: Señal que genera la función sigqueue.
 - **SI_TIMER**: Señal que genera timer.
 - **SI_ASYNCIO**: Señal que genera una entrada/salida por pantalla asíncrona.
 - **SI_MESGQ**: Señal que genera la llegada de un mensaje a una cola vacía.
 - **SI_USER**: señal que genera un kill u otras llamadas similares.

```
(...)  
sigset_t s; siginfo_t info;  
/* Bloquear señal */  
sigemptyset(&s);  
sigaddset(&s, SIGRTMIN);  
sigprocmask(SIG_BLOCK, &s, NULL);  
(...)  
/*Esperar señal; suponemos SA_SIGINFO  
activado */  
/* Si se usó kill para enviar el dato  
no es válido */
```

```
sigwaitinfo(&s, &info);  
if(info.si_code == SI_QUEUE)  
    printf("Dato: %d\n", info.si_value.sival_int);  
else  
    if(info.si_code == SI_USER)  
        printf("No hay dato; enviada por kill\n");  
    else printf("Causa no esperada\n");  
(...)
```

Señales POSIX 1003.1b



Señales POSIX 1003.1b: Tratamiento asíncrono

Ejemplo: Manejador que recibe datos

```
(...)
#include <signal.h>
(...)
Int n = 0;
void mtr(int s, siginfo_t *e, void *p) {
    n = n + e->si_value.sival_int;
}
(...)
struct sigaction acc;
sigset_t sen;

/* Programar manejador */
acc.sa_flags = SA_SIGINFO;
acc.sa_sigaction = mtr;
sigemptyset(&acc.sa_mask);
sigaction(SIGRTMIN, &acc, NULL);

/* Desenmascarar SIGRTMIN */
sigemptyset(&sen);
sigaddset(&sen, SIGRTMIN);
sigprocmask(SIG_UNBLOCK, &sen, NULL);

/* A partir de ahora mtr entrará
asíncronamente y sumará en n los datos
asociados a la señal */
(...)
```

Tratamiento síncrono sin manejador:

- Llamadas **sigwaitinfo** y **sigtimedwait**
- Hay que enmascarar las señales que se esperan
- Pueden desbloquearse por error EINTR por señales que no se separan.

Señales POSIX 1003.1b: Tratamiento síncrono

Ejemplo: Recepción de datos con **sigwaitinfo**

```
(...)
#include <signal.h>
(...)
Int n = 0;
void mtr(int s, siginfo_t *e, void *p) {}
(...)
struct sigaction acc;
sigset_t sen; siginfo_t info;

/* Programar manejador */
acc.sa_flags = SA_SIGINFO;
acc.sa_sigaction = mtr;
sigemptyset(&acc.sa_mask);
sigaction(SIGRTMIN, &acc, NULL);

/* Enmascarar SIGRTMIN */
/* Si no, no hay garantía de que funcione */
sigemptyset(&sen);
sigaddset(&sen, SIGRTMIN);
sigprocmask(SIG_BLOCK, &sen, NULL);

/* Bucle para sumar los datos */
/* mtr nunca entra */
while(1) {
    sigwaitinfo(&sen, &info);
    n = n + info.si_value.sival_int;
}
(...)
```

4.5 Generación de señales

Vamos a explicar primero lo que vale para 1003.1a y para 1003.1b, y después solo para 1003.1b

FUNCIÓN KILL

```
int kill (pid_t pid, int sig);
```

El primer argumento es el identificador del proceso en el que vamos a mandar la señal, aunque si escribimos 0 se envía a todos los procesos a la vez. El segundo parámetro es la señal que se va a enviar. Como siempre, devuelve 0 si va bien y -1 si hay algún error.

FUNCIÓN SIGQUEUE

Permite no solo mandar una señal a un proceso sino también mandar datos adicionales. Además, las señales se encolan. Solo se puede usar en 1003.1b. Su forma es:

```
int sigqueue (pid_t pid,int sig,const union sigval val)
```

Las dos primeras entradas funcionan igual que en kill. La última es una variable tipo “**union sigval**” que habremos definido previamente y que se le pasará al manejador, por lo tanto, será accesible su variable union sigval. De nuevo devuelve 0 cuando funciona y -1 si no.

Ejemplos:

```
pid_t pid;  
union sigval valor;  
pid=getpid();  
valor.sival_int=2;  
  
kill(pid,SIGALRM);  
sigqueue(pid,SIGALRM,valor);
```

En ambos casos mandamos la señal SIGALRM al proceso pid, pero en el Segundo además, mandamos el dato adicional de “valor”.

4.6 Espera de señales

4.6.1 Espera en 1003.1a

Se usa la función `sleep`, que espera el tiempo que se le pasa a menos que llegue una señal. Es de la forma:

```
unsigned int sleep (unsigned int seg)
```

Si acaba el tiempo de espera devuelve un 0, y si llega una señal devuelve el tiempo que le quedaba por esperar todavía.

También se puede usar la función `sigsuspend`:

```
int sigsuspend (const sigset_t *nueva_mascara)
```

Recibe un conjunto de señales, y lo que hace es bloquear el programa hasta que llega alguna señal que no esté incluida en ese conjunto de señales, en cuyo caso retorna el valor -1, se ejecuta el manejador de esa señal si lo hay, y entonces el programa sigue:

Ejemplo:

```
sigset_t conjunto;  
sigfillset(&conjunto);  
sigdelset(&conjunto, SIGALRM);  
sleep(30);  
sigsuspend(&conjunto);
```

Primero esperaría 30seg por `sleep`, a menos que llegue una señal, y después esperaría hasta que se enviara la señal `SIGALARM`.

4.6.2 Espera en 1003.1.b

Se usa sobre todo la función `sigwaitinfo`, de la forma:

```
int sigwaitinfo (const sigset_t *estas_sg, siginfo_t *info)
```

En este caso, el conjunto de señales del primer argumento debe estar bloqueada previamente con `sigprocmask`. Al recibirlas, `sigwaitinfo` las desbloquea y bloquea el resto, continuando el proceso solo cuando llegue una de esas señales que estaban bloqueadas. Cuando la señal llega, `sigwaitinfo` devuelve el número de esa señal, y además guarda el puntero `siginfo_t` la información adicional de esa señal que se hubiera mandado con `sigqueue`.

Ejemplo: Imaginemos que la señal es la que se mandaba en el ejemplo de sigqueue:

```
siginfo_t info;
sigset_t conjunto;
sigemptyset(&conjunto);
sigaddset(&conjunto, SIGALRM);
sigprocmask(SIG_BLOCK, &conjunto, NULL);

sigwaitinfo(&conjunto, &info);
```

Cuando se mande la señal **SIGALARM**, entonces dejará de esperarse, y en la variable “info” se guardará el valor 2 que se mandó, concretamente en **info.si_value.sival_int**.

También podemos usar la función **sigtimedwait**:

```
Int sigtimedwait (const sigset_t *estas_sg, siginfo_t *infor, const struct timespec *timeout)
```

Funciona igual que **sigwaitinfo**, solo que puede añadirse un tiempo límite de espera con una estructura tipo **timespec**, que es de la forma:

```
struct timespec {
    time_t tv_sec;
    long tv_nsec;
};
```

El primer elemento se guardan los segundos y en el segundo los nanosegundos. El máximo de ambos es 10^9 . Si ambos son cero, lo único que hace la función es comprobar si hay señales pendientes. Del ejemplo anterior.

```
Struct timespec t; t.tv_sec=40;

Sigtimedwait(&conjunto, &info, &t);
```

4.7 Manejadores

Los manejadores son funciones que se activan cuando la señal asociada a ellos con la función **sigaction** se activa. La forma de definirlos es muy sencilla, ya que simplemente se escribe su código entero antes del hilo main. Solo hay que distinguir entre los manejadores para señales definidas en 1003.1a y los que son para 1003.1b

MANEJADORES PARA 1003.1a

```
void nombremanejador (int signo)
{
<código del manejador>
}
```

MANEJADORES PARA 1003.1b

```
void nombremanejador (int signo, siginfo_t *datos, void *extra)
{
<código del manejador>
}
```

4.8 Temporizador basado en señales

Se usa la llamada:

Int alarm (unsigned int secs)

Cuenta los segundos que recibe, y cuando acaba active la señal **SIGALARM**. Devuelve 0 si va bien y -1 si hay error, y además los segundos que le quedaban por contar. Para desactivar la señal o detener el contador lo único que hay que hacer es introducir un 0 en el tiempo de espera.

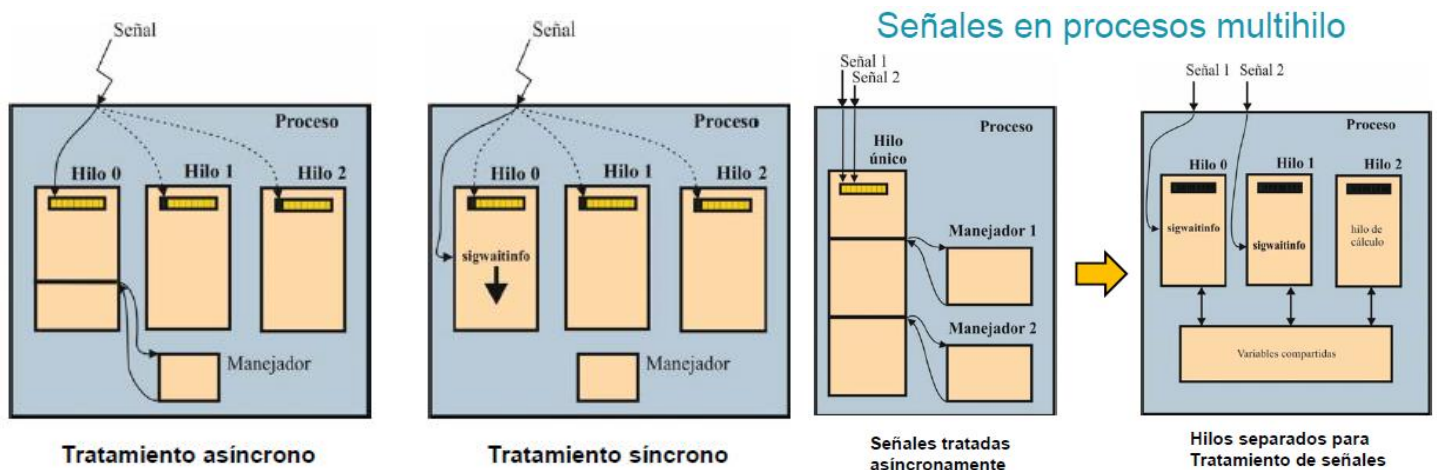
```
alarm(100);
sigwaitinfo(&conjunto,&info);
alarm(0);
```

Activa el temporizador para 100seg, pero si la señal que espera sigwaitinfo llega antes, pasa a la siguiente línea del código que desactiva el temporizador, luego no se activaría **SIGALARM**. En caso que tardara más de 100seg, **SIGALARM** se activaría

4.9 Señales en procesos multihilo

Selección del hilo destinatario:

- Tres casos:
 - Señal generada síncronamente por error: El hilo que la provoca
 - Señal enviada a un hilo con `pthread_kill`: Destinatario perfectamente definido
 - Señal enviada al proceso completo: Caso a discutir
- Cada hilo tiene una máscara propia que se modifica con `pthread_sigmask`:
 - Tratamiento asíncrono (manejador)
 - Manejador común para todos los hilos
 - Si la señal está enmascarada en todos menos uno, el manejador se ejecuta en el contexto de ese hilo
 - En caso contrario, no se puede predecir el destinatario
 - Tratamiento síncrono (`sigwaitinfo` y otras)
 - Señal enmascarada en todos los hilos
 - Solo un hilo esperando a la vez de la señal



VENTAJAS E INCONVENIENTE SEÑALES

Ventajas:

- Son asíncronas
- Simples de programar
- Orientadas a un proceso en particular
- No suelen tener restricciones

Inconvenientes

- Tratamiento poco eficiente
- Naturaleza asíncrona (salvo 1003.1b)
- Poco ancho de banda para transmitir datos

TEMA 5. TEMPORIZACIÓN

5.1 Introducción

Los **temporizadores** sirven para generar señales tras un cierto tiempo establecido, ya sea una única vez (disparo único) o repetidas veces. Igual que ocurría con las señales, hay una versión básica definida en 1003.1a y otra para tiempo real definida en 1003.1b, bastante más avanzada. En ambos casos, la librería que hay que incluir es la siguiente:

```
#include<time.h>
```

Los **temporizadores** no se acumulan, es decir, una vez que disparan su señal, si este no es atendida y se dispara otro temporizador la señal anterior se pierde. Entre las ventajas de los temporizadores en 1003.1b se encuentra el hecho de que, aunque no se traten estas señales, sí se contabiliza el número de ellas que se pierde ("overrun"). Además, aumenta el rango de precisión de tiempo de 1 segundo a 1 nanosegundo, y por último, permiten disparar cualquier señal, no sólo SIGALRM.

5.2 Servicios de temporización

Según causa de desbloqueo:

- **Tareas activadas por eventos:** Esperan cambios en la señal de entrada.
 - Soluciones:
 - Muestreo periódico (polling) : Simple y poco eficiente.
 - Respuesta a interrupciones. Más eficiente.
- **Tareas activadas por tiempo.** Necesitamos disponer de algún servicio que proporcione acceso a referencias de tiempo.
 - **Servicios de temporización:**
 - Acciones periódicas: Bucle de control
 - Aviso aislado
 - Retraso/Espera: Tarea bloqueada en el intervalo de t programado.
 - Sobretiempo ("timeout"). Tmax para realizar una actividad, si pasa este tiempo, hay que interrumpir la acción.
 - Referencias de t:
 - Relativas: Respecto al instante que se invoca
 - Absoluta: Evento a una hora y fecha determinada

Soluciones: Para los servicios de temporización.

1. **Temporizador por programa** ("Ballast codingW") : Solo para sistemas muy simples.
Expresa especificaciones de t como esperas e implementarlas en un código. Puede haber ciertas granularidades del reloj, latencia de interrupciones y expulsión de tareas que me derive en un error local acumulativo.
 2. **Reloj de tiempo real y temporizadores.** Incluye una referencia de tiempos que sirve como referencia absoluta y permite medir intervalos de t .
- **Temporizadores:**
 1. Repetitivos: Generan eventos de t repetidamente con T fijo
 2. Disparo único: Único evento aislado en un instante de t , con referencia relativa o absoluta.

Se puede construir un temporizador repetitivo a partir de uno de disparo único con referencia relativa, pero tenemos peor calidad de temporización y errores locales acumulativos por cada interacción. Esto no sucede con un temporizador periódico o disparo único con referencia absoluta.

5.2 Temporizadores en ADA

Temporización en ADA

- **Package Calendar** (obligatorio):
 - **Time**: Tipo que almacena un instante de tiempo.
 - **Duration**: Tipo que almacena un resultado de cálculos de tiempo
 - Número real en **punto fijo**. Indica segundos.
 - **Granularidad** de 20 ms como máximo
 - **Extensión** de -86400.0 a 86400.0 como mínimo
 - La resta de dos **Time** puede generar un resultado **Duration**
 - **Clock**: función que permite tener acceso al tiempo actual
- "Package" **Real_time**: opcional, con mayor resolución del reloj

- Intervalo de tiempo invertido en un conjunto de sentencias:

```

declare
  t1, t2: Time;
  intervalo: Duration;
begin
  t1 := Clock;
  <más sentencias>
  t2 := Clock;
  intervalo := t2 - t1;
end;

```

- Retraso de tiempo relativo (sentencia **delay**):

```

<sentencias1>
delay 10.0;
<sentencias2>

```

- Retraso de tiempo absoluto (sentencia **delay until**):

```

Comienzo := Clock;
<Sentencias1>
delay until Comienzo + 10.0;
<Sentencias2>

```

- Temporización periódica:

- Los retrasos **delay** introducen incertidumbres y deriva **acumulativa**.
- Esto no sucede con una referencia absoluta: **delay until**

```

declare
  Proximo: Time;
  Intervalo: constant Duration := 7.0;
begin
  Proximo := Clock + Intervalo;
  loop
    <sentencias>
    delay until Proximo;
    Proximo := Proximo + Intervalo;
  end loop;
end;

```

- Sobretiempo con la sentencia **select**:

```

select
  accept entrada1(l: Integer) do
    <sentencias>
  end entrada1;
or
  delay 5.0;
  <sentencias alternativas>
end select;

```

- Sobretiempo para abortar una actividad ya comenzada:

```

select
  delay 0.5;
  <sentencias A>
then abort
  <sentencias B>
end select;

```

5.3 Temporización en POSIX 1003.1a

La función `time` es de la siguiente forma:

```
time_t time (time_t *tiempo)
```

La función devuelve un el número de segundos transcurridos desde una referencia fija, que es el 1 de enero de 1970 a las 0:00 horas. En el contenido apuntado por el puntero también se almacena, pero puede ponerse `NULL`.

El tipo `time_t` es un entero `long`, definido en la librería `time.h`. Puede usarse en cualquier caso un entero `long` definido como tal, esto es, (`long int`). Para referenciar el dato por pantalla o almacenarlo, se usa “%ld”. El molde de `long int` es (`long`).

El resto de llamadas de temporización de 1003.1a son `sleep` y `alarm` que ya han sido explicadas. No es aconsejable usarlas porque su rango de precisión es 1 segundo. Además, ninguna de las dos puede actuar de forma repetitiva, son de disparo único, y ambas usan la señal `SIGALRM` luego no pueden utilizarse a la vez sin riesgo de fallos.

5.4 Temporización en POSIX 1003.1b

Mejoras:

- Mayor resolución (hasta ns).
- Temporizadores con diversos modos.
- Asignación flexible de señales a eventos.
- Detección de “overruns”.

Para definir los tiempos utilizaremos siempre la estructura “`timespec`”, que ya hemos definido, pero la repetimos por comodidad:

```
struct timespec {  
    time_t tv_sec;  
    long tv_nsec;  
};
```

Recordamos que tanto en el campo de los segundos (`tv_sec`) como el de los nanosegundos (`tv_nsec`) el máximo permitido es `10e9`.

Los **pasos para programar** un temporizador son los siguientes:

PRIMER PASO: DEFINICIÓN DE LAS VARIABLES A UTILIZAR

Tendremos que crear una variable estructura **"timespec"** para definir el tiempo del temporizador. Otra variable estructura **"itimerspec"** donde se indicarán los tiempos de espera, una última variable estructura **"sigevent"** donde especificaremos la señal que envía el temporizador y los datos adicionales de ésta, y otra variable **"timer_t"** que será el identificador de nuestro temporizador. Vamos a explicarlas:

- **Itimerspec**

```
struct itimerspec {
    struct timespec it_value;
    struct timespec it_interval;
};
```

En **"it_value"** ponemos el tiempo que se espera para disparar la señal desde que iniciamos el temporizador; en caso de que valga 0, el temporizador se desactiva. En **"it_interval"** ponemos el tiempo que espera entre una repetición y otra; en caso de que valga 0, el temporizador será de un solo disparo.

Por ejemplo:

```
struct timespec val={10,0}; /* Para contar 10 segundos
struct timespec inter={5,500000000L}; /* Para contar 5'5 segundos
struct itimerspec ejemplo;
```

```
ejemplo.it_value = val; /*La primera vez,esperaré 10 segundos para
disparar
ejemplo.it_interval =inter; /* Después, cada 5'5 segundos dispararé
```

- **Sigevent**

```
struct sigevent {
    ...
    int sigev_notify;
    ...
    int sigev_signo;
    ...
    union signal sigev_value;
};
```

Esta estructura tiene muchos más campos (de ahí los "...") pero a nosotros sólo nos interesan esos. En **"sigev_notify"** definimos cómo vamos a notificar el evento, en nuestro caso escribimos **"SIGEV_SIGNAL"**, que es lo que se escribe cuando se notifica mediante una señal, o si no queremos que se notifique de ninguna manera, escribimos **"SIGEV_NONE"**. En el campo **"sigev_signo"** metemos la señal que queremos que dispare el temporizador posteriormente. Por último, en **"sigev_value"** metemos la información adicional que mandamos con la señal.

Por ejemplo:

```
struct sigevent exp;
exp.sigev_notify = SIGEV_SIGNAL; /* Uso como notificación una señal
exp.sigev_signo = SIGRTMIN; /* La señal SIGRTMIN, en este caso
exp.sigev_value.sival_int = 8; /* Mandaré el número 8 con la señal
```

SEGUNDO PASO: CREO EL TEMPORIZADOR

Se usa una función que crea el temporizador, pero no lo inicializa:

```
int timer_create (clockid_t reloj, struct sigevent *aviso, timer_t
*tempo)
```

El primer argumento se introduce el reloj que se va a utilizar. Si bien se pueden definir relojes, nosotros SIEMPRE utilizaremos para cualquier función que reciba un “clockid_t reloj” la macro “**CLOCK_REALTIME**”. El segundo argumento será un puntero a la estructura “**sigevent**” que hayamos definido, con la señal que queremos que active el temporizador y los valores adicionales que mande. El último argumento es un puntero al identificador de nuestro temporizador, la variable “**timer_t**” que creamos antes, que será lo que nos permitirá referenciar el temporizador a otras funciones. El valor de retorno, como siempre, será 0 si todo va bien, y -1 si hay algún error.

Por ejemplo, siguiendo con los descritos en el primer paso:

```
timer_t tempor; /* Identificador de nuestro temporizador
timer_create(CLOCK_REALTIME,&exp,&tempor); /* Creamos el temporizador
```

TERCER PASO: ARRANCAR EL TEMPORIZADOR

```
int timer_settime (timer_t tempo, int flags, const struct itimerspec
*spec, struct itimerspec *spec_ant)
```

El primer argumento de esta función es el identificador de nuestro temporizador. El segundo argumento indica cómo se interpreta el valor del tercer argumento: nosotros siempre escribiremos “0”, que significa que espera lo que indica el “it_value” del tercer argumento para disparar por primera vez; si por el contrario escribimos “**TIMER_ABSTIME**”, lo que hace es que coge el valor de “it_value”, que son segundos, y le suma esos segundos a la fecha de referencia (las 0:00 horas del 1 de enero de 1970) y compara el resultado con el reloj de tiempo real, disparando por primera vez si la fecha ha sido superada, y esperando en caso de que no. En cualquiera de los dos casos, en el tercer argumento se coloca el puntero al “itimerspec” que creamos antes, que indica tanto cuando se dispara por primera vez como el intervalo entre los disparos, si los hay. El último argumento es un puntero a otro “itimerspec” para guardar el valor antiguo, puede dejarse a NULL si no se quiere guardar. Como siempre, la función retorna 0 si todo ha ido bien y -1 si ha habido algún error.

Para no tener que calcular los segundos desde una determinada fecha, usamos la estructura "tm" para definir nuestra fecha y la función "mktime" que la convierte en segundos, ambas definidas en la librería time.h:

```
struct tm {
    int tm_hour; /* Horas de 0 a 23
    int tm_min; /* Minutos de 0 a 59
    int tm_sec; /* Segundos de 0 a 59
    int tm_mon; /* Mes de 0 a 11
    int tm_mday; /*Día del mes de 1 a 31
    int tm_year; /* El año, siendo 0 el año 1900
};
```

```
time_t mktime (struct tm *fecha)
```

La función recibe el puntero de la estructura "Tm" y devuelve la fecha convertida en segundos desde la referencia (0:00 horas del 1 de enero de 1970).

Vamos a continuar primero el ejemplo anterior para inicializar el temporizador, y después escribiremos un ejemplo con struct tm y mktime. La continuación sería:

```
timer_settime(&tempor,0,&ejemplo,NULL); /* Inicializamos "tempor" con lo
indicado en la estructura "ejemplo"
```

Para que disparara por primera vez el 03-07-18 a las 20:30 horas, por ejemplo, tendríamos que modificar "ejemplo" de la siguiente forma:

```
time_t tiempo;
struct tm fecha = {20,30,0,6,3,1018};
tiempo=mktime(&fecha);
ejemplo.it_value = tiempo;
timer_settime(&tempor,"TIMER_ABSTIME",&ejemplo,NULL);
```

Además, hay dos funciones que hay que conocer porque nos dan información sobre un determinado temporizador en funcionamiento. Estas son:

- **timer_gettime:** nos permite saber cuánto falta para el próximo disparo. Es de la forma:

```
int timer_gettime (timer_t tempo, struct itimerspec *queda)
```

El primer argumento es el identificador del temporizador del que queremos saber cuánto tiempo falta para que dispare. El segundo argumento es un puntero a una estructura "itimerspec", donde en el valor "it_value" guarda el tiempo que resta para disparar, y en el valor "it_interval" guarda el intervalo de disparo, que puede ser diferente del que se programó porque el ordenador lo redondea a un número entero de periodos de su reloj. Como siempre, devuelve 0 si va bien y -1 si no.

- **timer_getoverrun:** permite saber cuántas veces se ha disparado el temporizador sin que su señal se trate. Es de la forma:

```
int timer_getoverrun(timer_t tempo)
```

Recibe el identificador del temporizador que vamos a comprobar, y devuelve el número de veces que se ha disparado. Si se supera el límite, lo que devuelve es "DELAYTIMER_MAX".

FUNCIONES QUE LLAMAN AL RELOJ DE REFERENCIA EN 1003.1b

Las siguientes funciones nos permiten trabajar con el reloj de tiempo real:

- *Conocer el tiempo actual:* para ello se usa la función

```
int clock_gettime (clockid_t reloj, struct timespec *tiempo)
```

El primer argumento como hemos dicho será "CLOCK_REALTIME". El segundo argumento es un puntero a la estructura "timespec" donde se guardará el tiempo actual. De nuevo, devuelve 0 si va bien y -1 si no.

- *Conocer la resolución/precisión del reloj:* para ello se usa la función

```
int clock_getres (clockid_t reloj, struct timespec *resol)
```

El primer argumento como hemos dicho será "CLOCK_REALTIME". El segundo argumento es un puntero a la estructura "timespec" donde se guardará la precisión del reloj. De nuevo, devuelve 0 si va bien y -1 si no.

- La función "clock_settime" permite modificar el reloj, pero no la usaremos.

TEMA 6. MEMORIA COMPARTIDA

6.1 Sincronización.

La comunicación por variables compartidas no incluye sincronización.

Sincronización:

- Sincronización por condiciones: Esperar determinado acontecimiento que sucede en otra tarea.
- Sincronización por exclusión mutua: Esperar a que los datos compartidos estén disponibles (exclusión mutua).
 - Necesario para que las operaciones sobre datos compartidos sean atómicas (indivisibles).
 - Asegura la coherencia de los datos compartidos.
 - Código en exclusión mutua: Sección crítica

6.1.1 Espera activa ("Busy Wait")

- **Inconveniente básico**: El proceso que espera realiza continuamente una actividad que no es útil.
- **Sincronización con condiciones**:

T1: (consumidor)
Hacer mientras flag = 0
Fin hacer
<realizar operaciones>

T2: (productor)
<realizar operaciones>
flag = 1

- **Exclusión mútua: No pueden evitarse fácilmente los fallos**
 - **Primer caso**: si los dos detectan el flag contrario a 1, **nunca accederán a los datos ("livelock")**:

T1:
flag1 = 1
/* La tarea 1 quiere entrar */
Mientras flag2 = 1 /* Primero espera a T2 */
Fin hacer
<sección crítica>
flag1 = 0 /* P1 ha acabado */
<resto de operaciones>

T2:
flag2 = 1
Mientras flag1 = 1
Fin hacer
<sección crítica>
flag2 = 0 /* T2 ha acabado */
<resto de operaciones>

- **Segundo caso**: si los dos detectan el flag contrario a 0, **no hay exclusión mútua**:

T1:
Mientras flag2 = 1 /* Primero espera a T2 */
Fin hacer
flag1 = 1 /* La tarea 1 toma posesión */
<sección crítica>
flag1 = 0 /* T1 ha acabado */
<resto de operaciones>

T2:
Mientras flag1 = 1
Fin Hacer
flag2 = 1
<sección crítica>
flag2 = 0
<resto de operaciones>

6.1.2 Semáforos

Un semáforo es un elemento que da acceso a un recurso de la memoria a un proceso y se lo niega al resto hasta que el anterior termine de utilizarlo. Es pues una forma de comunicación y sincronización entre procesos.

Su funcionamiento es el siguiente: un semáforo se inicializa con un valor entero no negativo, y siempre que dicho valor sea no negativo, los procesos pueden acceder al recurso que controla el semáforo. Siempre que un proceso vaya a acceder a un recurso, tiene que restarle una unidad al semáforo, y cuando acabe de usarlo, sumarle una unidad. Veamos un ejemplo:

Vamos a suponer que inicializamos un semáforo “SEM” con valor inicial “1” que controla el acceso al recurso “A”. El proceso “PACO” quiere acceder al recurso “A”, así que lo primero que hace es restarle una unidad al semáforo “SEM”, con lo cual el valor del entero del semáforo es “0”. Al ser un valor no negativo, “PACO” puede utilizar el recurso “A”. Acto seguido el proceso “PEPE” quiere acceder al recurso “A”, y para ello resta una unidad al semáforo “SEM”, y el valor del entero sería ahora “-1”, y como es un valor negativo, el proceso “PEPE” no accede al recurso “A” y se queda bloqueado en una cola de espera. Cuando el proceso “PACO” acabe de usar el recurso “A”, le sumará “1” al semáforo “SEM” y entonces el valor del entero será “0” y al ser no negativo, el proceso “PEPE” podrá acceder al recurso “A”, y al acabar, sumará también “1” al semáforo “SEM” dejando todo como estaba al principio.

Es importante aclarar que si el entero del semáforo llega un valor “-2”, por ejemplo, en cuanto otro proceso sume “1” al mismo entregará alguno de los procesos que estaban esperando, aunque el valor del entero sea negativo.

Existen semáforos “binarios”, que sólo pueden tomar los valores 0 y 1, pero que no los vamos a usar.

- Sincronización con una condición (sem_cond inicializado a 0):

T1: /* Espera la condición
generada por T2 */

<.....>

wait(sem_cond)

<realizar operaciones>

<.....>

T2: /* Genera la condición
que espera T1 */

<.....>

<realizar operaciones>

signal(sem_cond)

<.....>

- Exclusión mútua (sem_excl inicializado a 1):

T1:

<....>

wait(sem_excl) /* T1 espera a que T2 salga
de su sección crítica */

<.sección crítica>

signal(sem_excl) /* T1 permite acceso a T2 */

<.....>

T2:

<....>

wait(sem_excl) /* T2 espera a que T1 salga
de su sección crítica */

<.sección crítica>

signal(sem_excl) /* T2 permite acceso a T1 */

<.....>

Semáforos: Problemas de bloqueo

- Bloqueo mutuo (“**deadlock**”):

T1: wait(sem1) <....> wait(sem2) <....> signal(sem2) <....> signal(sem1)	T2: wait(sem2) <...> wait(sem1) <....> signal(sem1) <...> signal(sem2)
--	--

- T1 es expulsada antes de ejecutar el segundo **wait**, y después de ejecutar el primero
- T2 pondría **sem2** a 0, y quedaría bloqueada en su segundo **wait**
- Cuando T1 continúe, no podrá pasar del segundo **wait** tampoco
- La condición de bloqueo permanece indefinidamente

- Inanición (“**starvation**”):

- El algoritmo utilizado por el sistema operativo para escoger a el proceso que es desbloqueado lleva a algunos a la suspensión por **tiempo ilimitado** o con **límites indefinidos**

CREACIÓN, ACCESO Y DESTRUCCIÓN DE SEMÁFOROS

Solo están definidos en la 1003.1b. Hay dos tipos de semáforos, con nombre y sin nombre, y para cada uno se hace una cosa diferentes. Pero ambos usan la librería

```
#include <semaphore.h>
```

- **SEMAFOROS CON NOMBRE**

Funciona de forma similar a las colas de los mensajes, es decir, una vez creado, cualquier proceso puede obtener permiso para utilizarlo. Para ello usamos las siguientes funciones:

```
sem_t*      sem_open (const char *nombre, int oflags, mode_t modo,  
Unsigned int inicial)
```

El primer parámetro de esta función, hay que escribir, como en las colas, el nombre del semáforo, por ejemplo “SEM”, y para ello de nuevo lo haremos escribiendo “/SEM”. El segundo y tercer parámetro se usan exactamente igual que en las colas, salvo que sólo pueden usarse los flags O_CREAT y O_EXCL. El cuarto parámetro es el valor inicial que va a tener nuestro semáforo. El puntero tipo sem_t que devuelve apunta al semáforo creado.

```
int  sem_close  (sem_t *semáforo)
```

```
int  sem_unlink (const char *nombre)
```

Estas dos funciones funcionan igual que las colas:

Sem_close: cierra el semáforo para el proceso que lo invoca

Sem_unlink: lo elimina del sistema, una vez que todos los procesos que lo invocaban han invocado sem_close

- **SEMÁFOROS SIN NOMBRE**

Se crean en una zona de memoria que si es compartida por varios procesos y pueden utilizarlo para sincronizarse, sino solo tiene sentido para procesos que tengan muchos hilos de ejecución (threads). Si va a usarse memoria compartida, hay que inicializar esta, que ya veremos mas adelante como lo haremos.

```
int sem_init (sem_t *semáforo,int compartido, unsigned int inicial)
```

Esta función recibe el puntero de tipo sem_t que previamente tenemos que haber puesto apuntando a la zona de memoria compartida creada para almacenar el semáforo:

```
int sem_init (sem_t *semáforo,int compartido, unsigned int inicial)
```

```
sem_t *psem;
```

```
Memoria=<código para crear memoria compartida>
```

```
Psem= (sem_t*)memoria;
```

El segundo argumento que recibe vale "1" si va a usarse en semáforo en memoria compartida, y "0" si va a ser utilizado por diferentes hilos del mismo proceso. El último argumento es de nuevo el valor inicial del semáforo.

Para destruir el semáforo usamos la función:

```
int sem_destroy (sem_t *semáforo)
```

Que sólo recibe el puntero que apunta a la memoria compartida. Es importante notar que si se elimina dicha memoria compartida, también desaparece el semáforo.

SINCRONIZACIÓN CON SEMÁFOROS: AUMENTO Y DECREMENTO DE SU VALOR

```
int sem_wait (sem_t *psem)
int sem_trywait (sem_t *psem)
int sem_post (sem_t *psem)
int sem_getvalue (sem_t *psem, int *value)
```

La primera función es la encargada de decrementar en uno el valor del entero del semáforo, y si dicho valor queda como negativo, pues bloquea al proceso.

La segunda función hace la misma tarea que la anterior pero no bloquea al proceso, se limita a devolver -1 con el error EAGAIN.

La tercera función es la que suma "1" al valor del entero.

La última función almacena en el contenido apuntado por el puntero que recibe como segundo argumento el valor del entero.

6.3 Memoria Compartida en POSIX 1003.1b

Características:

- La zona de memoria debe aparecer (ser "mapeada") en el espacio de direccionamiento virtual de los procesos que la comparten.
- Conviene manejar un número entero de páginas de memoria

Operaciones a realizar:

- Abrir un objeto de memoria compartida: **shm_open**
- Fijar la longitud de dicho objeto: **ftruncate**
- "Mapear" el objeto sobre el espacio de direcciones de procesamiento: **mmap**
- El objeto puede cerrarse, pues es accesible de manera directa: **close**
- Cuando un proceso no la utilice, puede eliminarse del mapeo: **munmap**
- Cuando el objeto ya no sirve, puede destruirse: **shm_unlink**

Las librerías que tenemos que incluir en cualquier caso son las siguientes:

```
#include<sys/mman.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<limits.h>
```

CREACIÓN DEL OBJETO DE MEMORIA COMPARTIDA

int shm_open (const char *nombre, int oflags, mode_t modo)

De nuevo el primer argumento es un nombre, al igual que en las colas de mensajes y semáforos, se escribe entrecomillado y con el carácter “/” antes del nombre, por ejemplo, para llamarlo “MEMORIA”, pondríamos “/MEMORIA”. Los flags y el modo se usan exactamente igual, aunque no puede usarse O_WRONLY. El entero que devuelve es un descriptor de archivo que utilizaremos hasta el paso de “mapeo”.

FIJAR LA LONGITUD DEL OBJETO DE MEMORIA

Inicialmente el objeto de memoria tiene una longitud nula. Aunque se puede poner de cualquier tamaño. Nosotros pondremos un número entero de páginas de la memoria virtual. El tamaño de las páginas puede obtenerse con:

Long int sysconf (int name)

El argumento que le pasaremos será “_SC_PAGESIZE” y la dimensión de la página de la memoria virtual se almacenará en el entero long, que para nosotros será una variable tipo “off_t”

Una vez obtenida la dimensión de la ‘pagina de la memoria virtual, para fijar la longitud del objeto de memoria usaremos:

Int ftruncate (int fd, off_t tamaño)

El primer argumento es el descriptor de archivo del objeto compartido del cual se va a fijar la longitud, y como segundo argumento el tamaño que queramos para el mismo (un número entero de páginas)

MAPEO DEL OBJETO EN EL ESPACIO DE DIRECCIONES

```
void* mmap(void *donde, size_t long, int protec, int mapflags, int fd,
off_t offset)
```

El primer argumento es un puntero que indica al sistema dónde ubicar el objeto de memoria, pero no implica que lo vaya a colocar ahí porque es posible que no pueda. Lo normal es que este argumento valga 0 (el sistema lo colocará donde quiera).

El segundo argumento indica el tamaño de la zona mapeada, nosotros le pasaremos el mismo tamaño que le pasamos como segundo argumento a `ftruncate`, porque otro valor se usaría simplemente si quisiéramos acceder a un fragmento de la memoria compartida.

El tercer argumento son las protecciones de memoria que queremos establecer. Pueden ser: **PROT_READ** (habilita lectura), **PROT_WRITE** (habilita escritura), **PROT_NONE** (no habilita ningún modo de acceso)

El cuarto argumento podrá tomar 3 valores: **MAP_SHARED** (Permite compartir el mapeo con otros procesos) Este es el más usado. **MAP_PRIVATE** (Creando una copia privada del objeto mapeado, así que los cambios que haga el proceso que ejecuta `mmap` no los verán el resto de procesos), **MAP_FIXED** (obliga al sistema a aceptar el valor del primer argumento como dirección para almacenar el objeto de memoria).

El quinto argumento es el descriptor del archivo que devolvió **shm_open**. Es la última vez que lo usaremos, después lo destruiremos.

El último argumento siempre será 0.

El valor de retorno será útil para identificar al objeto de memoria compartida, por ejemplo, para usarlo en los semáforos sin nombre, y será un puntero a entero (para que sea así utilizaremos un molde). Una vez hayamos usado `mmap`, ejecutaremos lo siguiente:

```
Int close (int fd)
```

CIERRE DEL PSEUDOARCHIVO

Primero tenemos que liberar la memoria que no usaremos, de igual forma que hacíamos con “mq_close”, con la función:

```
Int    munmap(void *comienzo, size_t long)
```

El primer argumento es un puntero que apunta al principio de la memoria o donde se empezó a borrar, y se borrará desde donde apunta más la cantidad fijada en el segundo. Para liberar toda la memoria el primer argumento será el identificador del objeto de memoria que nos devuelve “mmap” y el segundo tamaño que le pusimos. Para destruir completamente el objeto de memoria se utilizará la función:

```
Int    shm_unlink (const char *nombre)
```

Solo recibe como argumento el nombre del objeto que le pusimos en **shm_open**, escrito tb. “/NOMBRE”. Hasta que todos los que abrieron no liberen toda la memoria con “munmap”, no se eliminará por completo el objeto, sólo se impedirá el acceso.

```
int descriptor;  
int *objeto;  
off_t tamañoobjeto;
```

```
Tamaño = sysconf(_SC_PAGESIZE);
```

```
descriptor=shm_open("/Ejemplo",O_CREAT | O_RDWR, S_IRWXU);
```

```
ftruncate(descriptor,tamaño);
```

```
objeto=(int *)mmap(0,tamaño,PROT_READ | PROT_WRITE,  
MAP_SHARED, descriptor,0);
```

```
close(descriptor);
```

```
<lo que sea que se haga con el objeto de memoria>
```

```
munmap(objeto,tamaño);  
shm_unlink("/Ejemplo");
```

6.4 Mutex

Un MUTEX es similar a un semáforo binario, pues puede estar “abierto” o “cerrado”. Se usa exclusivamente para impedir o permitir el acceso a parte de su código a los hilos de un mismo proceso. Una característica muy importante de los mutex es que, si hay varios hilos bloqueados, al activarse el mutex que los bloquea solo pasará uno de ellos, y dependerá de la prioridad de cada uno y del algoritmo de planificación elegido. Estos parámetros se fijan en los atributos del mutex.

INICIALIZACIÓN DE UN MUTEX

- **INICIALIZACIÓN ESTÁTICA**

Sencillo, pero no permite modificar los atributos del mutex.

```
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER
```

Donde **pthread_mutex_t** es una variable identificador de nuestro mutex, “mut” es el nombre que le hemos puesto a dicha variable y **PTHREAD_MUTEX_INITIALIZER** es la constante con la inicializaremos estáticamente nuestro mutex.

- **INICIALIZACIÓN DINÁMICA**

```
pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr)
```

Recibe como primer argumento el puntero a mutex con el que identificaremos a nuestro mutex, y que tenemos que inicializar antes (ahora lo vemos con el ejemplo). El segundo argumento son los atributos que queremos otorgarle a nuestro mutex, o NULL. Vemos un ejemplo de inicialización dinámica:

```
pthread_mutex_t *mut;  
mut = (pthread_mutex_t *)malloc(sizeof(pthread_mutex_t));  
pthread_mutex_init(mut, NULL);
```


ABRIR Y CERRAR UN MUTEX

Para cerrar usaremos:

Int pthread_mutex_lock (pthread_mutex_t *mutex)

Recibe un puntero al mutex que vamos a cerrar. Si el mutex se encuentra ya cerrado, la función bloquea al hilo hasta que dicho mutex se abra, en cuyo caso el hilo que ha invocado pthread_mutex_lock es el que cierra el mutex. Recordemos que cerraremos un mutex cuando queramos usar variables que queremos vetar al resto de hilos.

Para abrir de nuevo el mutex y permitir que otro hilo acceda a dichas variables:

Int pthread_mutex_unlock (pthread_mutex_t *mutex)

Existe una función similar que es sem_trywait en semáforos, y es:

Int pthread_mutex_trylock(pthread_mutex_t *mutex).

En este caso, si el mutex esta ya cerrado, en vez de bloquear al hilo simplemente devuelve el error EBUSY.

DESTRUIR UN MUTEX

Int pthread_mutex_destroy (pthread_mutex_t *mutex)

6.5 Variables de condición

Sirven para la sincronización de hilos eficazmente. Cada variable está ligada a un mutex y a una condición binaria, cuyo estado depende de unos datos que protegerá el mutex, para que no puedan actuarlo muchos hilos a la vez, sólo uno, el que permite el mutex.

Imaginemos que tenemos dos hilos, “HILOA”, Y “HILOB”, que usan la variable global “A”, cuyo valor incrementa “HILOA”, y cuyo acceso controla el mutex “MUT”, y que existe una variable condición “COND”, que se activa cuando “A” vale 5 y permite que “HILOB” ponga “A” a 0.

“HILOA”

```
Bucle infinito (while(1), por ejemplo)
  Bloqueo MUT
  Incremento el valor de A
  Compruebo la condición: si A = 5 -> activo la variable de condición
  si no: no hago nada
  Libero MUT y acabo
Fin bucle
```

“HILOB”

```
Bucle infinito
  Bloqueo MUT
  Bucle: mientras A sea menos de 5, hago
    Espero la variable de condición (esta espera libera el mutex
    asociado
      Mientras espera, y lo bloquea
      cuando
      Se termina de esperar)
  Fin bucle
  Pongo A a cero
  Libero MUT
Fin bucle
```

Cuando se espera a que se active la variable condición, se hace dentro de un bucle que comprueba dicha condición porque una vez se activa la variable todos los hilos que están esperando compiten para cerrar el mutex asociado, y el primero que lo consiga cerrará el mutex y modificará variables y por tanto, cuando cabe de hacer lo que haga, la condición que gobierna la variable de condición puede que no se cumpla, pero al liberar el mutex otro hilo de los que competían antes continuará sus acciones, a pesar de que la condición puede con cumplirse. Por eso se hace el bucle: porque al continuar, comprueba que se cumpla, y si es así, continúa, sino, vuelve a esperar.

INICIALIZACIÓN DE UNA VARIABLE CONDICIÓN

- **Inicialización estática**

Sencillo, pero al igual que con los mutex, no podemos modificar los atributos de la variable condición:

```
pthread_cond_t          cond = PTHREAD_COND_INITIALIZER;
```

Donde pthread_cond_t es una variable identificador de nuestra condición, "cond" es el nombre que le hemos puesto, y lo demás es la constante que inicializaremos estáticamente.

- **Inicialización dinámica**

```
int pthread_cond_init (pthread_cond_t *cvar, const pthread_condattr_t *attr)
```

Recibe como primer argumento el puntero a variable de condición con el que identificaremos a nuestra variable, y que tenemos que inicializar antes. El segundo argumento son los atributos que queremos asignarle a dicha variable, o NULL (defecto).

Ej:

```
pthread_cond_t *cond;  
cond = (pthread_cond_t *)malloc(sizeof(pthread_cond_t));  
pthread_cond_init(cond, NULL);
```

ESPERA DE UNA CONDICIÓN

Para ello, recordemos que siempre que esperamos una condición lo haremos dentro de un bucle que comprueba que esa condición se cumple, ya que una vez que se termina la espera de la activación de la variable condición, es posible que se nos adelante otro hilo y ya no se cumpla la condición.

```
int pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mutex)
```

Recibe como primer argumento el puntero a la variable de condición inicializada, y como segundo el mutex asociado a ella. De esta forma, cuando se invoca esta función, lo primero que hace ella es liberar el mutex, y después se pone a esperar. Cuando se active la variable de condición, intenta cerrar el mutex. Si alguien lo ha cerrado ya, sigue esperando hasta que vuelva a liberarse el mutex, momento en el cual intenta cerrarlo de nueva. Una vez lo consiga, acaba.

Es posible ponerle condiciones de tiempo a la espera de la activación:

```
int pthread_cond_timedwait (pthread_cond_t *cond, pthread_mutex_t *mutex,  
const struct timespec *abstime)
```

El tercer parámetro es el tiempo **ABSOLUTO** en el cual se acaba la espera. Tenemos que obtener el tiempo con "clock_gettime" y después sumarle el tiempo que queramos esperar. Retorna 0 si la causa por la que termina es la variable condición o retorna ETIMEDOUT si se ha alcanzado el límite de tiempo.

ACTIVACIÓN DE UNA VARIABLE DE CONDICIÓN

Cuando el hilo se bloquea esperando con “pthread_cond_wait”, solo puede ser desbloqueado si otro hilo activa la variable de condición (además del mutex abierto) ya sea porque otro hilo es el que hace que se cumpla la condición, o detecta que esta condición ha sido cumplida por un tercero. La función que se utiliza para activar la variable condición es:

int pthread_cond_signal (pthread_cond_t *cond)

Recibe el puntero de la variable de condición que activa. Esta función hace que al menos uno de los hilos bloqueados se desbloquee y continúe, cerrando el mutex.

Hay otra función para activar la variable condición:

int pthread_cond_broadcast (pthread_cond_t *cond)

Funciona igual, pero a diferencia de que TODOS los hilos sean desbloqueados y que compitan por cerrar el mutex, y después del que lo gane lo libere, sigan compitiendo.

Lo normal es que el hilo que activa la variable condición haya cerrado previamente el mutex, así que después de usar estas dos funciones, debe liberarlo.

6.6 Regiones crítica condicionales

- **Regiones críticas condicionales**

- Los datos compartidos se agrupan en recursos, con nombre conocido
- El lenguaje permite definir las regiones críticas, relacionándolas con un determinado recurso
- El sistema asegurará la ejecución de las regiones críticas relacionadas con el mismo recurso en exclusión mutua
- La sincronización con condiciones se resuelve con “guardas”

Recurso A:

<definición de variables compartidas>

int i;

Fin A.

Tarea P1:

<...>

Region A, cuando i!=0

<región crítica>

Fin Region.

<...>

Fin P1

- **Solución poco estructurada**
- **Reevaluación de condiciones:** elevado coste en conmutaciones de procesos

6.7 Monitores

Los monitores agrupan los datos compartidos y las rutinas (funciones) que permiten realizar las operaciones con los datos compartidos. De esta forma, varios procesos o hilos pueden acceder a unas mismas variables, pero solo uno de ellos estará a la vez dentro de él.

- **Principios de funcionamiento:**
 - Datos compartidos, inaccesibles de manera directa
 - Funciones que permiten realizar operaciones con los datos compartidos con acceso exclusivo
 - Sincronización con condiciones: operaciones **wait** y **signal**
- **Conclusiones:**
 - Encapsulamiento de datos y procedimientos
 - Mezcla de características de alto y bajo nivel (**wait** y **signal**) poco deseable
 - Servicios de **mutex** y **variables de condición** (POSIX 1003.1c) pensados para construir monitores

```
monitor A:
  condiciones:
    noescero;
    nomax;
  datos:
    <...>
    int MAX = 50;
    int i;
  fin datos;
  entrada dec:
    if(i==0) wait(noescero);
    i=i-1;
    signal(nomax);
    <...>
  fin dec;
  entrada inc:
    if(i==MAX) wait(nomax);
    <...>
    i = i + 1;
    signal(noescero);
  fin inc;
```

6.8 Datos y Objetos protegidos (ADA)

- **Principios de funcionamiento:**
 - Los datos compartidos solo son accesibles a través de procedimientos.
 - El acceso a los procedimientos del objeto protegido se realiza con garantías de exclusión mutua.
 - La sincronización con condiciones se realiza mediante barreras de entrada ("**barriers**")
- **Acceso al objeto protegido:**
 - **function**: Acceso concurrente de varias tareas, con "**read lock**": no se pueden modificar los datos compartidos; no se permite el acceso a **procedure** o **entry** hasta que quede libre el recurso.
 - **procedure**: Pueden modificarse los datos compartidos, y se garantiza el acceso exclusivo; el recurso está bajo "**read/write lock**".
 - **entry**: Llevan asociada una condición ("**barrier**") que debe verificarse, además de estar libre el recurso.

```
protected type ejemplo is
  function lee return Integer;
  procedure reset;
  entry dec(decre: in Integer);
  entry inc(inc: in Integer);
private
  midato: Integer := 0;
  maximo: Integer := 50;
end;
protected body ejemplo is
  function lee return Integer is
  begin
    return midato;
  end lee;
  procedure reset is
  begin
    midato := 0;
  end reset;
  entry dec(decre: in Integer)
  when midato > 0 is
  begin
    midato = midato - decre;
  end dec;
  entry inc(inc: in Integer)
  when midato <= maximo is
  begin
    midato = midato + inc;
  end inc;
end ejemplo;
```

Objetos protegidos (ADA)

- Evaluación de las condiciones de barrera:
 - Una tarea llama a una entry, y existe la posibilidad de que la condición se haya modificado desde su última evaluación.
 - Una tarea sale del objeto protegido, existen tareas encoladas a causa de una barrera que no se verifica y existe la posibilidad de que la tarea que abandona el objeto haya provocado el cambio de su condición asociada.

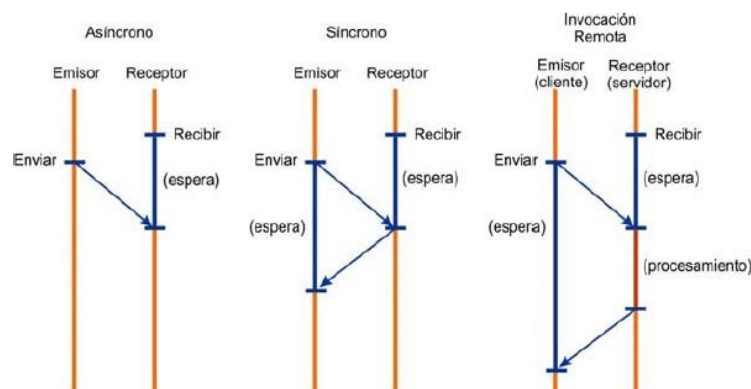
7. MENSAJES

1. Introducción

Son un tipo de comunicación entre proceso. Una cola de mensajes es como un espacio en memoria que crea un proceso, pero no dentro de él, por tanto, son independiente de los procesos. Una vez hemos creado dicha cola, cada proceso puede “abrir” dicha cola con unos permisos (R,W,Execute) y a partir de ahí intercambiar mensajes con dicha cola. Un proceso puede cerrar la cola, aunque no la destruye, solo interrumpe su comunicación con ella. (aunque es posible destruirla).

MODELOS DE SINCRONIZACIÓN

- **Asíncrono:** El emisor no espera a que los datos sean recibidos. Es necesaria una cola de espera.
- **Síncrono:** El emisor espera a que se produzca la recepción.
- **Invocación remota:** Protocolo llamada-respuesta.
- El modelo asíncrono puede utilizarse para emular los otros dos mediante un protocolo de mensajes.
- Inconvenientes:
 - Necesidad de una zona de almacenamiento intermedio (cola) que puede llenarse
 - Dificultad para verificar la corrección del sistema
 - Menor eficiencia si se utiliza para emular otros modelos



REFERENCIA AL EMISOR/RECEPTOR

- **Según el tipo de asociación entre las dos partes**
 - **Explícita:** En la orden de envío/recepción se especifica un identificador de la otra parte. “send <proceso destinatario>”
 - **Indirecta:** En la orden de envío/recepción se especifica el identificador de una estructura intermedio, como una cola de recepción “send <cola>”.
- **Situación en los dos procesos (emisor y receptor)**
 - **Simétrica:** En ambos lados la referencia es del mismo tipo (explícita o indirecta).
 - **Asimétrica:** No se realiza la referencia del mismo modo.

2. Descripción a nivel de lenguaje: ADA

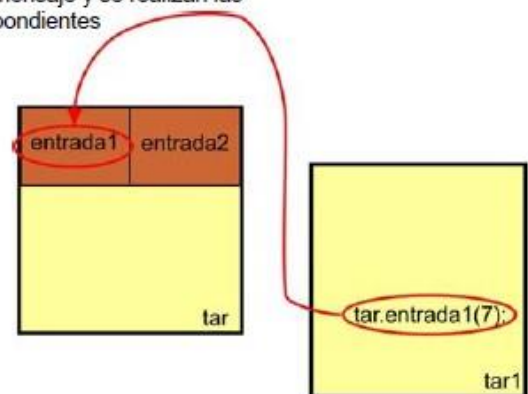
- “Entry”: Concepto de ADA que implementa el paso de mensajes

– Tarea tar; posee varias “entry”
task tar is –Esta es la especificación “externa” de la tarea
 entry entrada1(l: in Integer);
 entry entrada2

end tar;

task body tar is
 <sentencias>
 accept entrada1(l: in integer) do –Aquí se acepta el mensaje y se realizan las
 <tratamiento del mensaje> – operaciones orrespondientes
 end entrada1;
 <sigue el cuerpo de la tarea>
end tar;

– La tarea tar1 utiliza una entry de la tarea tar
task body tar1 is
 <sentencias>
 tar.entrada1(7);
 <más sentencias>
end tar1;



- Sentencia **select**: alternativa entre varias “entry”

```
<sentencias>
select
  accept entrada1(l: in integer) do
    <sentencias>
  end entrada1;
or
  accept entrada2(.....) do
    <sentencias>
  end entrada2;
  <resto de posibilidades>
or
  (...)
end select;
<más sentencias>
```


3. Paso de mensajes local en POSIX

Opciones:

- **PIPES (POSIX 1003.A)**
 - Cola circular de datos
 - Acceso similar a un fichero utilizando las llamadas de bajo nivel (read,write), aunque con bloqueo cuando no se puede realizar la operación (cola vacía o cola llena).
 - Se construyen en la fase de inicialización, entre **fork** y **exec**.
 - Inconvenientes:
 - Mecanismos de creación complicados y difíciles de entender.
 - Unidireccionales.
 - En la práctica es necesario que los procesos implicados tengan un “antepasado” común.
 - Fácil de utilizar desde el intérprete de comandos, para crear un “pipeline” de comandos, conectado salidas y entradas “standard”

ls | grep “.c” | wc

- **FIFOs (POSIX 1003.A)**
 - Concepto similar a los “pipes”, pero más flexibles y fáciles de usar.
 - La “FIFO” es un objeto que existe en el sistema y tiene un nombre.
 - Cualquier proceso con permisos adecuados puede crearla o abrirla con un fichero normal para enviar (write) o recibir (read) datos.
 - Son objetos que persisten hasta ser explícitamente destruidos, aunque desaparezcan los procesos que los utilizaron.
 - No existe el concepto de mensaje. Emisor y receptor pueden enviar y recibir número de octetos distintos.
- **Inconvenientes FIFO Y PIPES**
 - No tienen en cuenta la prioridad del mensaje.
 - No se suele conocer o modificar fácilmente el espacio disponible en la cola.
 - No se puede conocer el estado de la cola.
 - No se puede definir una estructura de mensaje, ni siquiera un tamaño
 - Número de colas limitados por el máximo número de archivos abiertos.
 - Al estar asociadas al sistema de archivos, la implementación puede ser ineficiente.

- **Colas de mensajes (POSIX 1003.1B)**
 - Medio de comunicación bastante más adecuado que las “FIFOs”.
 - Creación y manejo muy similar al de los ficheros, pero con un conjunto de llamada separado, y espacio de nombres propio.
 - La llegada de mensajes puede ser notificada mediante señales.
 - Ventajas respecto a “FIFOs”
 - Cada mensaje tiene una prioridad asociada y se recibe antes que los de menos prioridad.
 - Al crear la cola se define explícitamente un tamaño máximo de mensaje y una capacidad.
 - Puede conocerse en cualquier momento el número de mensajes encolados.
 - Aunque no existe estructura de mensaje, si se conoce su tamaño. Un mensaje se envía y recibe obligatoriamente con una sola llamada.
 - La implementación de las colas puede ser independiente de la de los archivos, y por tanto, más eficiente y con menos limitaciones.
- **Sockets** (también pueden usarse localmente)

Funciones	Archivos ANSI-C	Archivos bajo nivel	Mensajes	Semáforos
Abrir/Crear	fopen	open	mq_open	sem_open
Cerrar	fclose	close	mq_close	sem_close
Acceder	fread/fwrite	read/write	mq_receive/ mq_send	sem_post/ sem_wait
Destruir	remove	unlink	mq_unlink	sem_unlink

4. Colas de mensaje

CREACIÓN Y APERTURA DE UNA COLA DE MENSAJES

```
#include <mqueue.h>
```

```
#include <sys.stat.h>
```

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

Esta librería incluye elementos importantes:

```
Mqd_t //Identificador
```

```
Struct mq_attr {  
    Long mq_maxmsg;  
    Long mq_msgszie;  
    Long mq_flags;  
    Longs mq_curmsgs;  
    ...  
};
```

Elementos de la estructura:

1. El máximo número de mensajes que puede almacenar la cola.
2. El máximo tamaño de un mensaje que almacena la cola.
3. El único Flag es O_NONBLOCK, pero nosotros siempre pondremos 0.
4. Indica cuantos mensajes quedan por recibir en la cola.

Para crear y abrir una cola usaremos:

```
mqd_t mq_open (const char *mq_name, int oflag, mode_t modo, struct  
mq_attr *atributos);
```

Argumentos:

1. Nombre que le ponemos a la cola. “/NOMBRE”.
2. Especificamos para qué abrimos la cola:
 - **O_RDONLY**: abre la cola, solo lectura.
 - **O_WRONLY**: abre la cola, solo escritura.
 - **O_RDWR**: abre la cola para lectura y escritura.
 - **O_CREAT**: si no existe, se crea.

- **O_EXCL**: si se pone con O_CREAT, la función devolverá un fallo (EEXIST) a través de errno si ya existía la cola.
- **O_NONBLOCK**: el programa no se bloqueará pase lo que pase cuando llamemos a esta función.

3. Indica los permisos que tendrá el nombre del proceso que abre la cola sobre la misma, siendo estos definidos de la siguiente forma:

S_I + R, W, X + USR/GRP

El primer sumando nunca cambia. El segundo: lectura (R), escritura (W) o ejecutar (X). El último sumando es quien tiene el permiso, si usuario (USR) o grupo de usuarios (GRP).

Por ejemplo, S_IRUSR permite leer al usuario. Si queremos leer y recibir al usuario, ponemos **S_IRUSR**. Si queremos leer y escribir, ponemos **S_IRUSR | S_IWUSR**.

En caso de querer habilitar los tres permisos, existen constantes especiales: **S_IRWXU** (usuario) y **S_IRWXG** (grupo).

4. Puntero a los atributos definidos en una estructura del tipo mq_attr. Si ya está creada la cola, pondremos NULL.

El argumento que devuelve es el identificador de la cola creada/abierta, y será -1 siempre que haya habido algún error.

Ejemplo:

```
mqd_t cola;

struct mq_attr atributos;
atributos.mq_maxmsg = 100;
atributos.mq_msgsize = 128;
atributos.mq_flags = 0;

cola = mq_open("/NOMBRESITO", O_CREAT | O_RDWR, S_IRWXU, &atributos);
```

Creamos una estructura atributos, donde definimos el número máximo de mensajes que puede almacenar la cola (100), el tamaño máximo de cada mensaje (128) y ponemos los flags a 0. Después creamos la cola "/NOMBRESITO", cuyo identificador es la variable cola, y la hemos creado para leer y escribir en ella, además de abrirla a nuestro proceso para que usuario pueda leer, escribir y ejecutar.

CERRAR Y ELIMINAR COLAS DE MENSAJES

- **int mq_close (mqd_t cola)**

Recibe el identificador de la cola que se va a dejar de acceder, con la que se interrumpe la comunicación, pero NO la destruye. Devuelve -1 si hay error o 0 si ha sido todo correcto.

- **int mq_unlink (const char *nombre)**

Recibe el nombre de la cola que se quiere eliminar. La destruye permanentemente, liberando los recursos asociados a ella. Si algún proceso estaba usándola, se destruye cuando el último proceso que la esté usando use "mq_close". Devuelve -1 si hay error, 0 si va bien.

Por seguridad, antes de crear la cola, invocamos mq_unlink para destruirla, por si ya existía.

```
mq_close(colas); mq_unlink("/NOMBRESITO");
```

ENVIO Y RECEPCIÓN DE MENSAJES

```
int mq_send (mqd_t cola, const char *datos, size_t longitud,  
unsigned int prioridad);
```

```
size_t mq_receive (mqd_t cola, const char *datos, size_t longitud,  
unsigned int *prioridad);
```

Argumentos:

1. Identificador de la cola que va a mandarse o de la que se va a recibir un mensaje.
2. Puntero a una cadena de caracteres, mensaje que se manda o mensaje que se recibe.
3. Tamaño máximo del mensaje enviado. Poner tamaño máximo definido en los atributos.
4. Prioridad: [0, MQ_PIO_MAX (al menos 32)]. En mq_send ponemos directamente la prioridad. Y en el caso de "mq_receive" se pasa el puntero a una variable unsigned int, y en él se guarda la prioridad del mensaje recibido.

Devuelve -1 si algo va mal. Las llamadas bloquean el proceso si no pueden bien mandar el mensaje (no hay espacio en la cola) bien recibir un mensaje (no hay mensajes en la cola), pero no bloquearan el proceso si se activó el flag O_NONBLOCK, solo devolverán un error EAGAIN en error.

Ejemplo:

```
char mensaje[128],recibido[128];  
unsigned int prio;  
mensaje="ejemplito";  
mq_send(colas,mensaje,128,0);  
mq_receive(colas,recibido,128,&prio);
```

Ahora "recibido" tiene también el mensaje "Esto es un ejemplo".

ACTIVACIÓN DE SEÑALES POR MENSAJE

1. Usamos estructura sigevent:

```
struct sigevent {  
    ...  
    int sigev_notify;  
    ...  
    int sigev_signo;  
    ...  
    union sigval sigev_value;  
};
```

Argumentos:

1. SIGEV_SIGNAL

2. Ponemos el nombre de la señal que queremos que se active cuando llegue un mensaje.

3. Tenemos que pasar como dato adicional un puntero vacío de la cola con la que trabajamos.

2. Usamos `mq_notify`:

```
int mq_notify (mqd_t cola, const struct sigevent *espec)
```

Recibe el identificador de la cola en la que la recepción de un mensaje activa la señal. El segundo argumento es un puntero a la estructura sigevent que hemos creado previamente.

****Una vez que llega por primera vez un mensaje y se activa la señal, el efecto de mq_notify desaparece, es decir, que cuando llegue otro mensaje la señal no se activará. Para que eso ocurra habría que volver a invocar a mq_notify, y suele hacerse desde el manejador que trata la señal que activa el mensaje:*

```
struct sigevent senal;  
senal.sigev_notify = SIGEV_SIGNAL;  
senal.sigev_signo = SIGRTMIN;  
senal.sigev_value.sival_ptr = (void *)&cola;  
mq_notify(cola,&senal);
```

Cuando llegue la un mensaje a la cola, se activará la señal SIGRTMIN. Si quisiéramos que siempre que llegaran mensajes se activase SIGRTMIN, tendríamos que hacer:

```
void man (int sig, siginfo_t *info, void *nada)  
{  
    mqd_t *cola;  
    struct sigevent espec;  
    cola = (mqd_t *)info->sival_ptr;  
    espec.sigev_notify = SIGEV_SIGNAL;  
    espec.sigev_signo = SIGRTMIN;  
    espec.sigev_value.sival_ptr = (void *)cola;  
    mq_notify(*cola,&espec);  
}
```

CONOCER LOS ATRIBUTOS DE UNA COLA

```
int mq_getattr (mqd_t cola, struct mq_attr *atributos)
```

Recibe el identificador de la cola cuyos atributos se quieren conocer, y un puntero a una estructura `mq_attr` donde se almacenan los atributos de la cola consultada. Si todo va bien, devuelve 0.

Es interesante para conocer el valor de “`mq_curmsgs`”, que son los mensajes pendientes de llegar a la cola.

```
struct mq_attr receptor;  
mq_getattr(cola,&receptor);
```


ANEXO

2. PUNTEROS

Podemos imaginarnos un puntero como una flecha apuntando a una dirección de memoria. Por ejemplo:

```
Char *puntero;
```

```
*puntero = 'A';
```

Tenemos una flecha que apunta a una dirección de memoria, más bien, a cualquier dirección de memoria, al azar. Si después intentamos guardar algo en dicha dirección, se dan dos posibilidades:

1. Dirección aleatoria a la que apunta el puntero pertenezca a nuestro programa. Introducimos 'A' en la dirección y aparentemente no ha pasado nada. El error puede presentarse en cualquier otro lado, en un sitio aparentemente en buen funcionamiento.
2. Dirección a la que apunta no pertenezca nuestro programa. Esto es mejor, ya que nos dará un error de violación de memoria.

Para evitar esto, inicializaremos todos los punteros a NULL:

```
Char *puntero = NULL;
```

DIFERENCIAS ENTRE '*' Y '&'

'&': Me da la dirección de un objeto en memoria

'*': Me da el contenido de una posición de memoria.

```
pc = &c;
```

```
printf ("\nNo tiene lo mismo %c que %d", c, pc); /* Ojo, %d para pc*/
```

```
printf ("\nTiene lo mismo %d que %d", &c, pc); /* Direcciones */
```

```
printf ("\nTiene lo mismo %c que %c", c, *pc); /* Caracteres */
```

APUNTAR UN PUNTERO A UNA DIRECCIÓN DE MEMORIA

- Apuntarlo a una **dirección de memoria ya reservada por nuestro programa**. Basta solo con asignarlo a la dirección de cualquier variable que tengamos declarada o igualarlo a otro puntero que ya esté apuntando a una dirección adecuada.

```
char caracter;
```

```
char *puntero = NULL;
```

```
...
```

```
puntero = &carácter;
```

El puntero apunta a una dirección de memoria en la que está carácter. Podemos usar con seguridad la memoria a la que apunta puntero, ya que lo que pongamos ahí también se pone en carácter.

```
*puntero = 'A';
```

- Reservar una **zona de memoria específica** para nuestro puntero. Usamos malloc(), que reservará una zona de memoria del tamaño que le indiquemos y nos devuelve su dirección.

```
puntero = malloc (...);
```

```
*puntero = 'A';
```

Una vez reservada la zona de memoria, podemos usarla con seguridad. Si no queremos que nuestro programa consuma excesiva memoria, debemos liberarla cuando no la necesitemos más:

```
free (puntero);
```

PROBLEMAS:

- Si nuestro puntero apunta a una variable local, al desaparecer la variable, nuestro puntero quedará apuntando a una dirección de memoria no válida:

```
char *función ()
```

```
{
```

```
    Char resultado;
```

```
    Char *puntero;
```

```
    Resultado=algo;
```

```
    Puntero = &resultado;
```

```
    Return puntero;  
}
```

La variable resultado tiene sentido dentro de la función, pero cuando acaba la función, desaparece la variable, y puntero apuntaba a la dirección de memoria que ocupaba esa variable, cuando queremos usar el puntero, esa memoria ya está libre y es posible que se sobre-escriba.

NO DEVOLVER NUNCA PUNTEROS A VARIABLES LOCALES A UNA FUNCIÓN

- Al liberar zona de memoria con free() nuestro puntero apunta a una dirección que no es correcta.

```
Char *puntero = NULL;
```

```
Puntero = malloc();
```

```
Free(puntero);
```

```
*puntero = 'A';
```

No tendremos ningún problema de violación de memoria, sin embargo, alguien puede sobre-escribir esa dirección de memoria.

APUNTAR A NULL LOS PUNTEROS DESPUES DE LIBERARLOS

```
Free(puntero);
```

```
Puntero = NULL;
```

Cuando lo intentemos utilizar de forma incorrecta, el programa caerá inmediatamente, con lo que será más sencillo depurar.

- Liberar dos veces la misma zona de memoria puede dar problemas, al liberar por segunda vez el programa no da error, pero deja "corrupto" al gestor de memoria. Lo más probable es que después nos de fallo en malloc.

Es bastante habitual hacer que una función reserve un espacio de memoria y lo devuelva. Es necesario cuando hacemos un malloc(), tener claro quien va a liberar esa memoria y donde, para evitar problemas

POR CADA MALLOC(), HACER UN ÚNICO FREE().

PUNTEROS Y LOS ARGUMENTOS A FUNCIONES

```
{VERSION ERRONEA}  
intercambia (int a, int b) {  
    int tmp;  
  
    tmp = a;  
    a = b;  
    b = tmp;  
}
```

```
{VERSION CORRECTA}  
intercambia (int *a, int *b) {  
    int tmp;  
  
    tmp = *a;  
    *a = *b;  
    *b = tmp;  
}
```

Para que un parámetro de una función pueda ser modificado, ha de pasarse por referencia, y en C eso solo es posible pasando la dirección de la variable en lugar de la propia variable.