Universidad de Oviedo

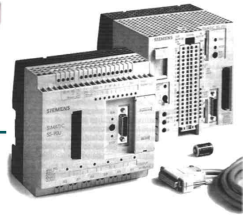
Control por Computador

Ignacio Alvarez García

Octubre - 2011

Indice

Computadores para control

- Microcontrolador 
- Procesador Digital de Señal (DSP) 
- Dispositivos Electrónicos Programables (FPGA, PLD) 
- Computador embebido 
- Ordenador Industrial 
- Autómata Programable (PLC) 

Implantación del control

- Variables: almacenan valores actuales y anteriores de entradas, salidas, estados y parámetros
- Funciones: realizan cálculos habituales (media, convolución, gráfico, …)
- Funcionamiento temporal: los valores evolucionan en el tiempo
- Tiempo-real: importan el valor de la respuesta y su plazo
- Multi-tarea: varias cosas que hacer ‘a la vez’
- Programación orientada a eventos: la secuencia de acontecimientos es desconocida a priori
- Tipos de eventos:
  - Vencimiento temporización
  - Cambio de estado de entrada(s)
  - Recepción de comunicación
  - …

  } Interrupciones

Implantación del control

- Sistemas ‘pequeños’:
  - Sin soporte de Sistema Operativo
  - El programador accede directamente al hardware
    - Programación de puertos de E/S
    - Servicio de interrupciones
  - El programador debe realizar el planificador:
    - Round-robin: chequeo ordenado de las tareas a realizar en cada momento
    - Round-robin con interrupción: las tareas más prioritarias o puntuales son lanzadas por interrupciones hardware y servidas en Rutinas de Servicio de Interrupción (ISR)
    - Function-queue-scheduling: las ISR encolan las tareas a realizar con sus prioridades. Un planificador round-robin comprueba la cola y va sirviendo por orden de prioridad

Implantación del control

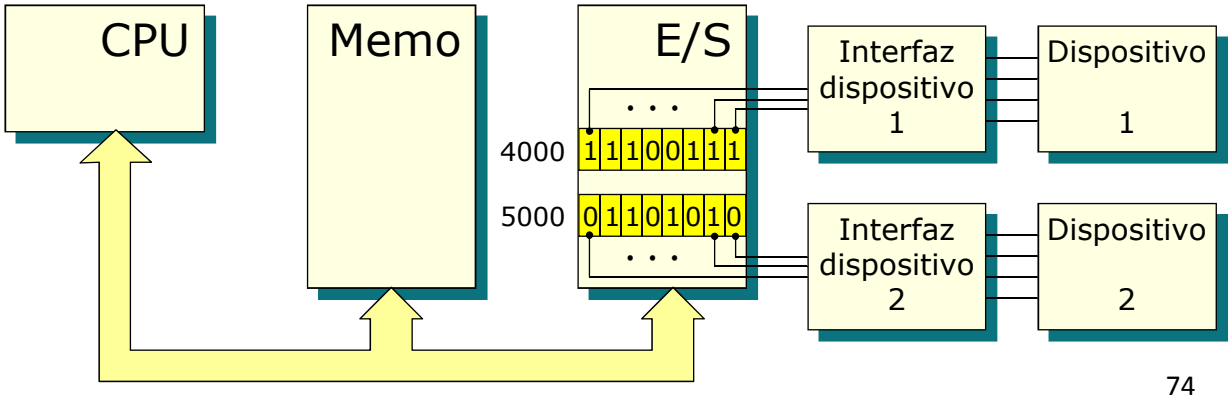
- Sistemas ‘grandes’:
  - Una tarea especial Sistema Operativo (S.O.) se encarga de dar servicio a:
  - Acceso al hardware (modo síncrono y asíncrono):
    - Nivel de S.O.
    - Nivel de driver
  - Planificación de tareas:
    - Cada vez que sucede un ‘evento’, el S.O. toma el control, actualiza estado de tareas, y pasa a ejecutar la más prioritaria.
  - Gestión de memoria
  - El programador debe realizar llamadas a funciones del S.O. para todos los servicios
  - El S.O. puede llamar a funciones del programador (callback) para rutinas de servicio asíncrono

Implantación del control

- Sistemas “pequeños” vs sistemas “grandes”:
  - La tarea S.O. requiere recursos (memoria y tiempo de ejecución), que pueden no estar disponibles en sistemas pequeños.
  - La tarea S.O. requiere ejecutarse en modo privilegiado: la CPU debe disponer de 2 modos de funcionamiento (usuario y privilegiado).
  - La tarea S.O. facilita operaciones que pueden ser requeridas en sistemas grandes:
    - La programación de E/S (no hay que conocer todos los puertos e interrupciones)
    - La gestión de tareas (prioridades, sincronización, memoria)
    - Las protecciones de seguridad basadas en privilegios y claves.
    - Las comunicaciones

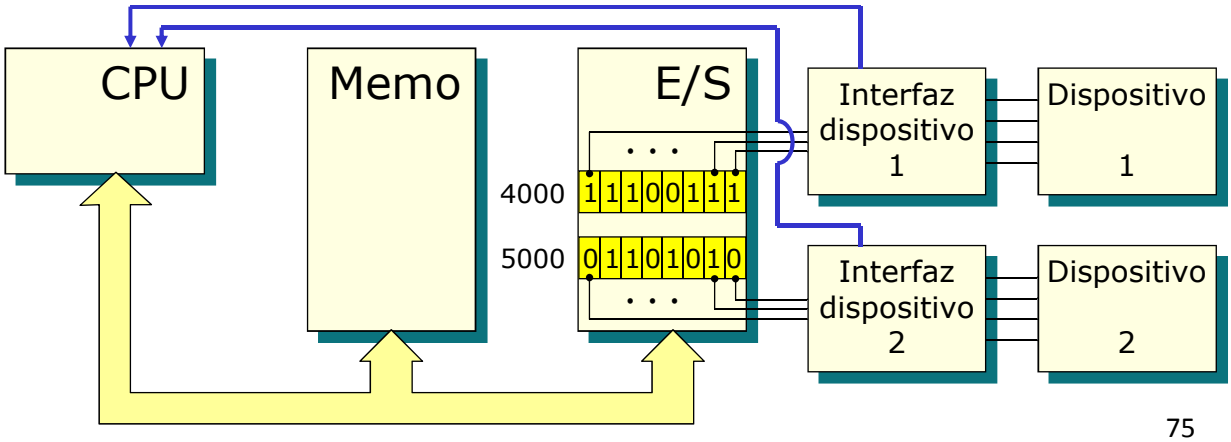
Sistemas “pequeños”

- Programación de E/S:
  - La E/S se realiza escribiendo y leyendo puertos de E/S: lugares de almacenamiento cuyo contenido provoca cambios o es modificado por los dispositivos de E/S.
  - Los puertos están organizados en direcciones.
  - Es necesario conocer las direcciones de E/S de cada dispositivo, y el significado de cada bit de los puertos.



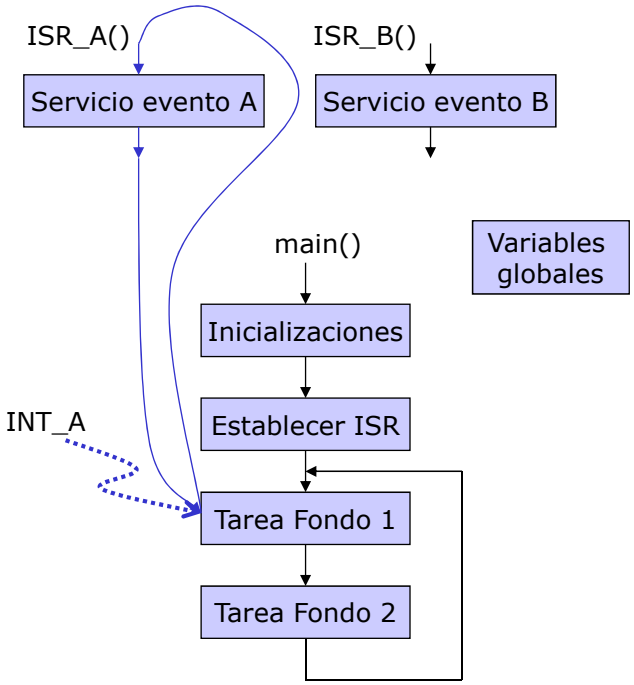
Sistemas “pequeños”

- Servicio de eventos:
  - Los dispositivos de E/S envían una interrupción a la CPU cuando necesitan ser atendidos.
  - La interrupción provoca que la CPU abandone temporalmente el programa que estaba ejecutando, pase a ejecutar una ISR, y retorno al programa cuando ésta termina.



Sistemas “pequeños”

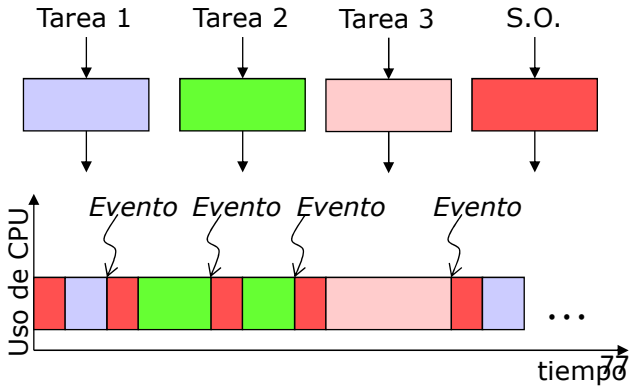
- Organización del programa:
  - Un bucle principal con la(s) tarea(s) de fondo.
  - Funciones de Servicio de Interrupción (ISR) para el servicio de eventos.
  - Las ISR detienen a la tarea de fondo o a otras ISR: ejecución rápida y sin esperas.
  - Gestión de prioridades de las ISR.
  - Las variables compartidas por el programa y las ISR deben ser globales.



Sistemas “grandes”

- El programador organiza su código en tareas, que se ejecutan “en paralelo” o, más exactamente, “en concurrencia”.
- Las tareas no siempre quieren ejecutar su código, están en muchas ocasiones en estado de espera (por teclado, por temporización, por aviso de otra tarea, …)
- Una tarea especial, el Sistema Operativo, se encarga de:
  - Gestión de puertos E/S mediante pequeños programas (drivers)
  - Gestión de eventos y temporizaciones
  - Planificación de tareas
  - Gestión de memoria

Ante cualquier evento:
- El S.O.toma el control (la CPU pasa a ejecutar su código).
- El evento puede provocar el cambio de estado de alguna tarea.
- El S.O. revisa las tareas pendientes y cede el control a la más prioritaria que necesite ejecutarse, hasta el siguiente evento.

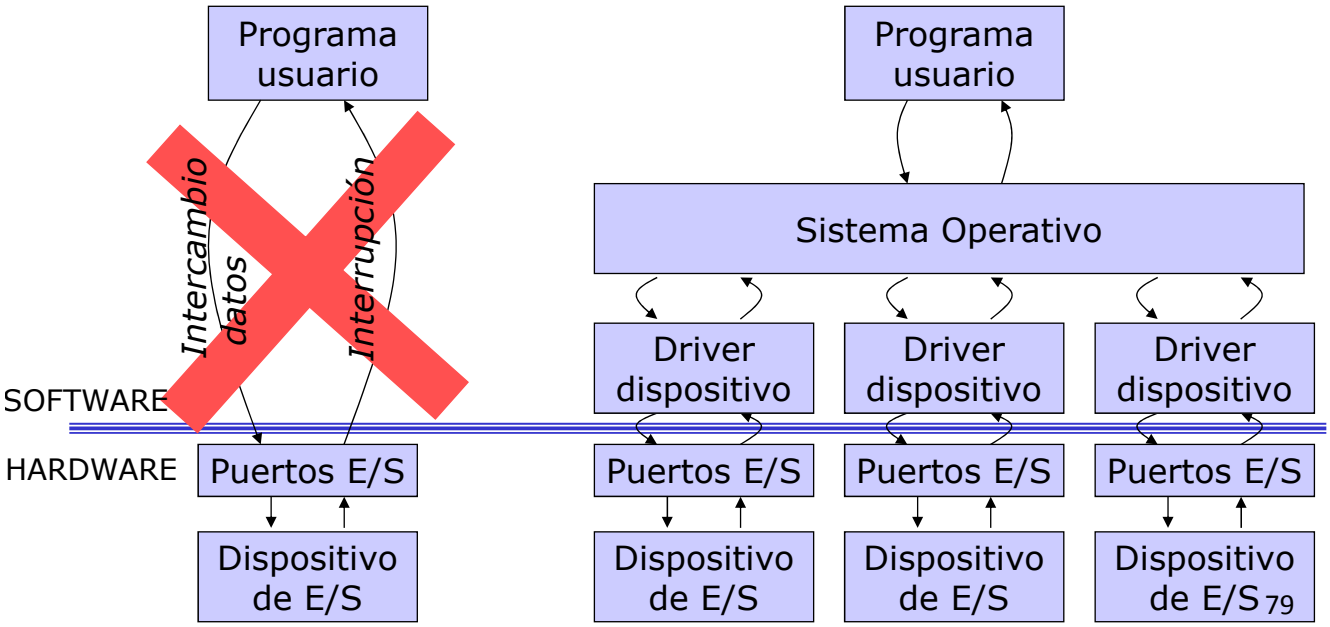


Sistemas “grandes”

- Variedad de Sistemas Operativos:
  - Windows, Unix, Linux, Mac OS, Android, VX-Works, …
- Tipos de Sistemas Operativos:
  - De tiempo compartido:
    - El retardo en ejecutar una tarea no es importante.
    - El S.O. distribuye tiempos de ejecución entre todas las tareas.
  - De tiempo real:
    - El retardo en ejecutar una tarea puede ser muy importante.
    - El S.O. ejecuta siempre la tarea más prioritaria.
- POSIX: Portable Operating System Interface:
  - Basado en Unix
  - Mismo interfaz con las tareas (system calls) para diferentes Sistemas Operativos.

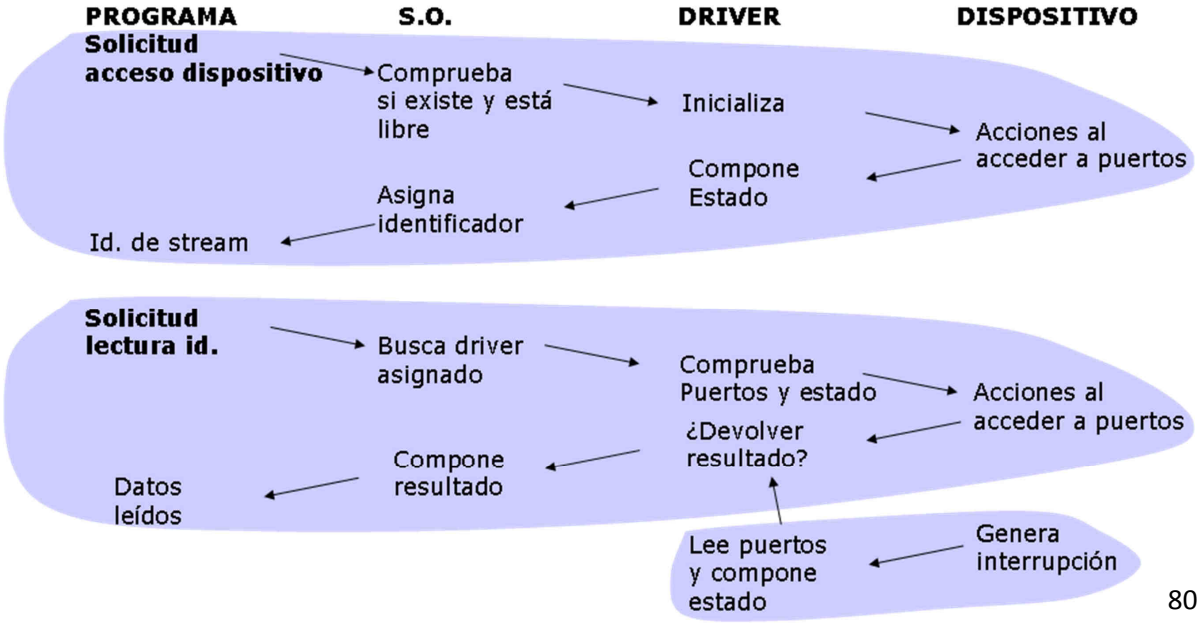
Gestión de E/S con S.O.

- El programador interacciona con los dispositivos de E/S a través del S.O. y drivers de dispositivo.



Gestión de E/S con S.O.

- Los dispositivos son tratados como “streams”: corrientes de datos de entrada o salida



E/S por stream

- Stream: corriente de datos → se envían o reciben un conjunto ordenado de bytes al/del dispositivo
- Todos los dispositivos son tratados de igual forma: se leen o escriben corrientes de bytes
- Tipos de streams:
  - De entrada / salida / entrada y salida
  - Con memoria (archivos en dispositivos de almacenamiento) / sin memoria (resto de dispositivos)
  - Orientados a texto / binarios
- Denominación de dispositivos: cadena de caracteres (ejs "COM1", "/dev/ttyS0", "C:\usuario\datos.txt")
- El S.O. se encarga de la organización lógica de los dispositivos (nombres, drivers, derechos de acceso, libre/ocupado,…).
- Los drivers se encargan del acceso físico a los dispositivos (leer/escribir puertos, gestión de interrupciones).

E/S por stream

- E/S bloqueante y no bloqueante
  - E/S bloqueante (síncrona): la tarea espera hasta la finalización de la operación de E/S.
    - La tarea se queda en estado "espera" hasta la finalización.
    - El S.O. reactiva la tarea cuando termine la operación de E/S.
    - Ejemplo: scanf("%d",&x);
  - E/S no bloqueante: la tarea no espera la finalización de la operación de E/S.
    - La tarea continúa con su ejecución normal.
    - La finalización puede ser notificada a la tarea:
      - De forma asíncrona (similar a una interrupción)
      - Mediante espera síncrona

E/S por stream en C

- El id. de stream es un FILE*
- Todas las funciones de E/S son bloqueantes.
- Existen funciones específicas para streams de texto.
- Llamadas de libería para gestionar E/S en C:

Descripción	**Función** <stdio.h>
Crear conexión con un dispositivo (o archivo)	fopen()
Cerrar conexión con un dispositivo (o archivo)	fclose()
Escribir datos en un dispositivo de texto	fputc(), fputs(), fprintf()
Leer datos de un dispositivo de texto	fgetc(), fgets(), fscanf()
Funciones auxiliares	ferror(), clearerr(), fflush()
Funciones auxiliares para dispositivos con almacenamiento	feof(), rewind(), fseek(), ftell()

E/S por stream en POSIX

- El id. de stream es un int
- Las funciones de E/S son directas (sin transformación de datos)
- Llamadas al S.O. para gestionar E/S **síncrona**:

Descripción	**Función** <unistd.h>
Crear conexión con un dispositivo (o archivo)	open()
Cerrar conexión con un dispositivo (o archivo)	close()
Escribir datos en un dispositivo	write()
Leer datos de un dispositivo	read()
Configurar driver de dispositivo	ioctl()
Configurar acceso a un dispositivo	fcntl()
Esperar finalización de operaciones de E/S no bloqueantes	select()
Configurar puerto serie (<termios.h>)	tcgetattr(), tcsetattr()

E/S asíncrona en POSIX

- El id. de stream es un int
- Las funciones de E/S son directas (sin transformación de datos)
- Llamadas al S.O. para gestionar E/S **asíncrona**:

Descripción	**Función** <aio.h>
Escribir datos en un dispositivo	aio_write()
Leer datos de un dispositivo	aio_read()
Comprobar estado de una solicitud asíncrona	aio_error()
Comprobar resultado de una solicitud asíncrona	aio_return()
Esperar finalización de solicitudes asíncronas	aio_suspend()
Cancelar una solicitud asíncrona	aio_cancel()
Preparar varias solicitudes asíncronas a la vez	lio_listio()

E/S asíncrona en POSIX

- Notificaciones en funciones de E/S asíncrona:
  - Notificación por señal
  - Notificación mediante lanzamiento de nuevo hilo

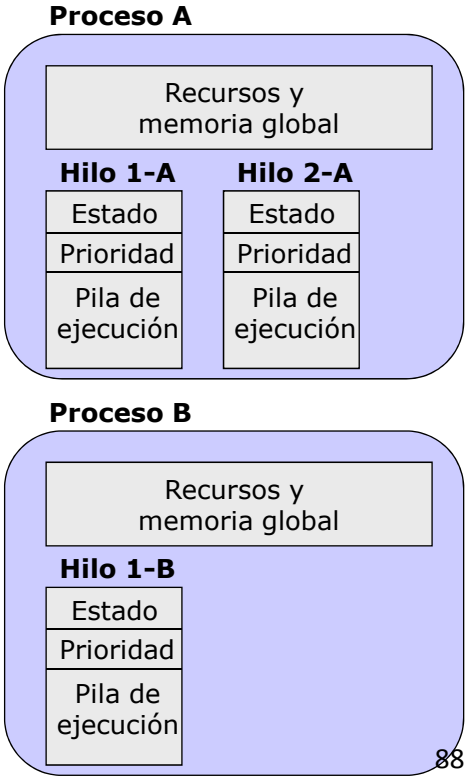
```
union sigval
{
    int     sival_int;          /* Integer value */
    void   *sival_ptr;          /* Pointer value */
};

struct sigevent
{
    int          sigev_notify; /* Notification method: SIGEV_NONE / SIGEV_THREAD /
                                                        SIGEV_SIGNAL / SIGEV_THREAD_ID */
    int          sigev_signo;  /* Notification signal */
    union sigval sigev_value;  /* Data passed with notification */
    void       (*sigev_notify_function) (union sigval);       /* Function used for thread
                                                                 notification (SIGEV_THREAD) */
    void        *sigev_notify_attributes;      /* Attributes for notification thread (SIGEV_THREAD) */
    pid_t        sigev_notify_thread_id;       /* ID of thread to signal (SIGEV_THREAD_ID) */
};
```

Gestión de eventos y tareas

- Los drivers sirven las interrupciones de los dispositivos de E/S, y “avisan” al S.O.
- Las tareas se ejecutan concurrentemente, esto es, en competencia por usar un recurso único: la CPU.
- Las tareas indican al S.O. si necesitan ejecutar código o están pendientes de algún “aviso”.
- Las tareas indican al S.O. su prioridad de ejecución frente a otras tareas.
- Las tareas necesitan sincronizarse entre sí, para efectuar “avisos” o utilizar recursos compartidos.
- Ante cualquier aviso del driver o solicitud de una tarea:
  - El S.O. actualiza el estado de la(s) tarea(s).
  - El S.O. revisa la(s) tarea(s) en estado de necesidad de ejecución.
  - El S.O. cede la ejecución a la tarea más prioritaria.

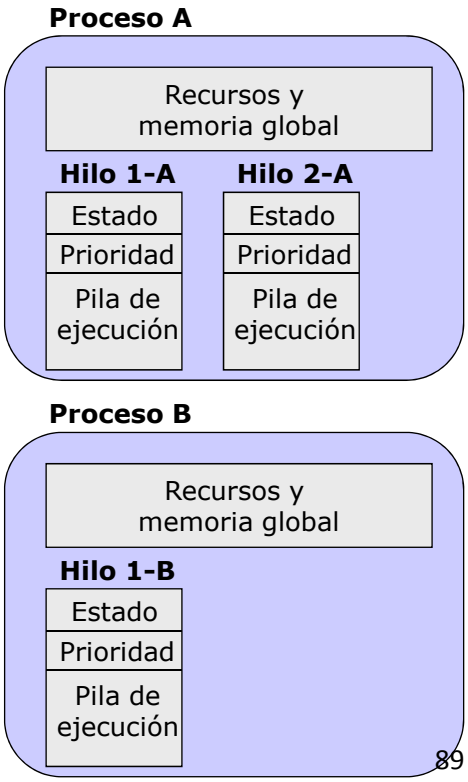
Gestión de tareas

- Las tareas se clasifican en **procesos** e **hilos** (threads).
- Procesos:
  - El S.O. ejecuta los procesos en espacios de memoria separados (memoria virtual), y protege sus recursos de otros procesos.
  - Los procesos sólo pueden comunicarse a través del S.O.
  - Cada proceso tiene al menos un hilo.
- Hilos:
  - Los hilos son subprocesos dentro de un proceso.
  - Cada hilo tiene su propia pila de ejecucion, prioridad y estado.
  - Los hilos de un proceso comparten todos los recursos, excepto su pila.
  - Los hilos de un proceso pueden comunicarse mediante las variables globales.



Gestión de tareas

- El Sistema Operativo se encarga de:
  - Determinar qué proceso e hilo se debe ejecutar en cada momento.
  - Guardar el contexto de cada proceso e hilo cuando no se ejecuta.
  - Recuperar el contexto del proceso e hilo que pasa a ejecutarse.
- Contexto de un hilo:
  - Valor de los registros en el momento de su suspensión (PC, SP, etc.).
- Contexto de un proceso:
  - Identificadores de streams abiertos.
  - Asignaciones de memoria virtual.
  - Derechos de ejecución y acceso.



Gestión de procesos en POSIX

- Un programa en ejecución es un proceso. Se distinguen mediante un identificador de proceso (pid)
- Un proceso puede crear otro: el creador se llama padre, y el creado se llama hijo.
- Llamadas al S.O. para gestionar procesos:

Descripción	Función	Cabecera
Crear un nuevo proceso	fork()	<unistd.h>
Cambiar la imagen de un proceso	exec…()	<unistd.h>
Obtener identificador del proceso actual	getpid()	<unistd.h>
Obtener identificador del proceso padre	getppid()	<sys/types.h>
Terminar el proceso actual	exit()	<stdlib.h>
Solicitar terminación de otro proceso	kill()	<signal.h> <sys/types.h>
Esperar terminación de un proceso hijo	waitpid()	<sys/wait.h>
Esperar terminación de cualquier proceso hijo	wait()	<sys/types.h>

Gestión de procesos en POSIX

- Creación de un nuevo proceso: fork() + exec…()
- fork(): crea un nuevo proceso igual al actual.
  - Los dos procesos (padre e hijo) continúan su ejecución en la instrucción siguiente a fork().
  - El valor devuelto por fork() indica si es el proceso hijo (0) o padre (devuelve el pid del hijo).
- exec…(): cambia la imagen de un proceso por otro ejecutable.
  - Las instrucciones siguientes a exec() no se ejecutan.
  - Varias posibilidades: execl(), execlp(), execle(), execv(), …

```
#include …

int main (void)
{
    pid_t childpid;
    childpid = fork();
    if (childpid == -1)
    {
        --- Error al crear el hijo ---
    }
    else if (childpid ==0)
    {
        execl("MIPROG.EXE",…);
        --- No se ejecuta nada más ---
    }
    --- Continúa el proceso padre ---
    --- Puede usar childpid para  ---
    --- comunicarse con el hijo   ---
}
```


Gestión de hilos en POSIX

- Un hilo de un proceso se llama thread. Se distinguen mediante un identificador de hilo (tid)
- Un hilo puede crear otro. Ambos son “hermanos”.
- Un proceso termina cuando lo hacen todos sus hilos.
- Llamadas al S.O. para gestionar hilos:

Descripción	**Función** <pthread.h>
Crear un nuevo hilo	pthread_create()
Terminar el hilo actual	pthread_exit()
Solicitar terminación de otro hilo	pthread_cancel()
Esperar terminación de otro hilo	pthread_join()
Obtener identificador del hilo actual	pthread_self()
Cambio de atributos de un hilo	pthread_attr_...()
Otras:	pthread_test_cancel() pthread_set_cancel…()

Gestión de hilos en POSIX

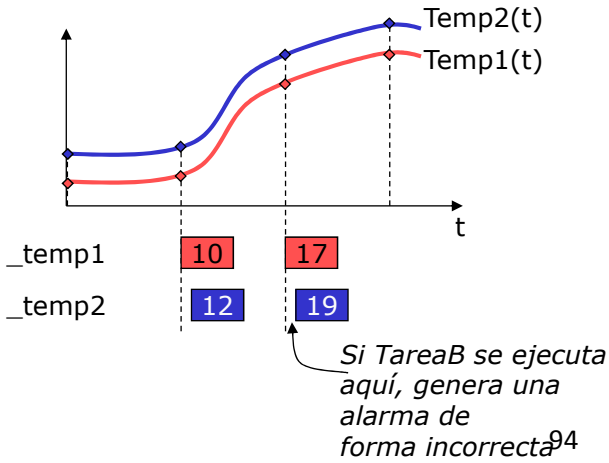
- Creación de un nuevo hilo:
  - int pthread_create (pthread_t *tid, const pthread_attr_t *attr, void *(*start_routine)(void*), void *arg);

```
#include <pthread.h>

void* Hilo2(void* param)
{
    --- Ejecución Hilo 2 (en concurrencia con Hilo 1) ---
    --- El Hilo 2 se termina cuando se termine esta función ---
}

int main (void)     // main() es siempre el 1er hilo del proceso
{
    pthread_t id_hilo2;
    int err;

    err=pthread_create(&id_hilo2,NULL,Hilo2,NULL);
    --- Continúa el Hilo 1. Puede usar id_hilo2 para  ---
    --- comunicarse con el hijo                        ---
    --- El hilo 1 se termina cuando se termine esta función ---
}
```


Necesidades de sincronización

- La sincronización entre tareas es necesaria para:
  - Que una tarea espere a que otra finalice un determinado trabajo.
  - Que una tarea avise a otra de que ha realizado un determinado trabajo.
  - Que dos tareas utilicen adecuadamente recursos compartidos (memoria, dispositivos de E/S, …)

```
int _temp1,_temp2;

Fn_TareaA()
{
    _temp1=LeerDispositivoES(1);
    _temp2=LeerDispositivoES(2);
}

Fn_TareaB()
{
    if (_temp1>_temp2)
        --- ALARMA ---
}
```



Sincronización de tareas

- Entre hilos de un mismo proceso:
  - **Mútex**: exclusión mútua a un solo recurso.
  - **Variable de condición**: espera por un evento en un recurso con exclusión mútua.
- Entre hilos del mismo o distintos procesos:
  - **Semáforos**: exclusión mútua para múltiples recursos.
  - **Señales**: aviso de eventos.
  - **Colas de mensajes**: aviso + datos.
- La sincronización también es necesaria en sistemas sin S.O.
  - En estos casos, la sincronización se realiza mediante la inhibición/habilitación de interrupciones, y precisa código específico del programador.

Mútex en POSIX

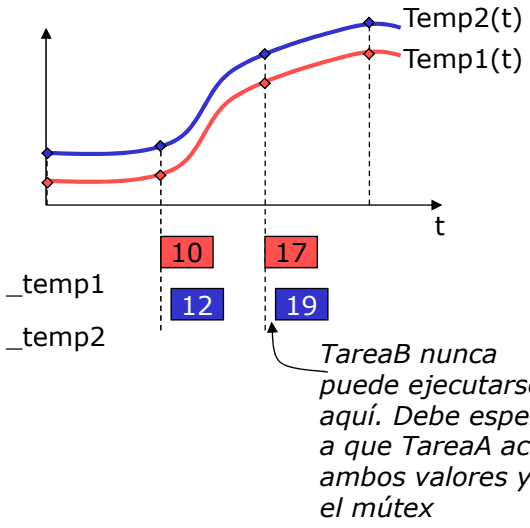
- Mútex o cerrojo: permite evitar/permitir acceso a un recurso compartido entre hilos.
- Si un hilo bloquea un mútex, otros hilos que quieran bloquearlo deben esperar a que se libere.
- Si un hilo libera un mútex, uno de los hilos que estaban esperando por él puede pasar a ejecutarse, con el mútex bloqueado por el nuevo hilo.
- Llamadas al S.O. para gestionar mútex:

Descripción	**Función** <pthread.h>
Crear un nuevo mútex	pthread_mutex_init()
Destruir un mútex	pthread_mutex_destroy()
Bloquear un mútex (espera si estaba bloqueado por otro hilo)	pthread_mutex_lock()
Libera un mútex (si estaba bloqueado por este hilo)	pthread_mutex_unlock()
Comprueba un mútex y bloquea si está libre	pthread_mutex_trylock()

Sincronización con mútex

- Solución del problema de sincronización con mútex:

```
int _temp1,_temp2;
pthread_mutex_t _mutex;

Fn_TareaA()
{
    pthread_mutex_lock(&_mutex);
    _temp1=LeerDispositivoES(1);
    _temp2=LeerDispositivoES(2);
    pthread_mutex_unlock(&_mutex);
}

Fn_TareaB()
{
    pthread_mutex_lock(&_mutex);
    if (_temp1>_temp2)
        --- ALARMA ---
    pthread_mutex_unlock(&_mutex);
}
```



Sincronización con mútex

- A tener en cuenta al sincronizar con mútex:
  - La variable mútex debe ser global (accesible por varios hilos).
  - Un hilo que bloquea un mútex debe desbloquearlo lo antes posible, ya que retrasa la ejecución de otros hilos.
  - Nunca se debe olvidar desbloquear un mútex.
  - Si una tarea accede al recurso compartido sin utilizar el mútex, se mantiene el problema.
  - Se pueden utilizar mútex distintos para recursos independientes entre sí.
  - La ejecución de las funciones de mútex conlleva tiempo de CPU.

Sincronización con mútex+vble. condición

- En el ejemplo anterior:
  - TareaB debe ejecutarse periódicamente para chequear el estado de alarma, usando tiempo de CPU.
  - Si no hay alarma, TareaB no hace nada.
  - Sería bueno que TareaB sólo se ejecute cuando es necesario (hay alarma), bloqueando el mútex sólo en ese caso.
- Solución:
  - Mútex + variable de condición

```
int _temp1,_temp2;
pthread_mutex_t _mutex;

Fn_TareaA()
{
    pthread_mutex_lock(&_mutex);
    _temp1=LeerDispositivoES(1);
    _temp2=LeerDispositivoES(2);
    pthread_mutex_unlock(&_mutex);
}

Fn_TareaB()
{
    pthread_mutex_lock(&_mutex);
    if (_temp1>_temp2)
        --- ALARMA ---
    pthread_mutex_unlock(&_mutex);
}
```


Variables de condición en POSIX

- Una vble. de condición está asociada a un mútex.
- Un hilo puede esperar por una vble. de condición: libera el mútex a la espera de que otro hilo señale esa condición.
- Un hilo puede señalar una condición a otro, liberándolo de la espera (a falta de disponer del mútex).
- Llamadas al S.O. para gestionar variables de condición:

Descripción	**Función** <pthread.h>
Crear una nueva variable de condición	pthread_cond_init()
Destruir una variable de condición	pthread_cond_destroy()
Espera por variable de condición (+mútex)	pthread_cond_wait()
Espera por variable de condición (+mútex) con tiempo máximo	pthread_cond_timedwait()
Libera a un solo hilo que espera por la vble. de condición	pthread_cond_signal()
Libera a todos los hilos que esperan por la vble. de condición	pthread_cond_broadcast()

Sincronización con mútex+vble. condición

- Ejemplo anterior con variable de condición:
  - La tarea B no ejecuta código hasta que no ocurre la condición que desea.
  - La tarea B sólo ejecuta su código cuando:
    - Se ha cumplido la condición
      Y
    - Se ha liberado el mútex.
  - La tarea B ejecuta su código con el mútex bloqueado por ella.

```
int _temp1,_temp2;
pthread_mutex_t _mutex;
pthread_cond_t _cond;

Fn_TareaA()
{
    pthread_mutex_lock(&_mutex);
    _temp1=LeerDispositivoES(1);
    _temp2=LeerDispositivoES(2);
    if (_temp1 > _temp2)
        pthread_cond_broadcast(&_cond);
    pthread_mutex_unlock(&_mutex);
}

Fn_TareaB()
{
    while (1)
    {
        pthread_mutex_lock(&_mutex);
        pthread_cond_wait(&_cond,&_mutex);
        if (_temp1>_temp2)
            --- ALARMA ---
        pthread_mutex_unlock(&_mutex);
    }
}
```


Semáforos en POSIX

- Semáforo: permite evitar/permitir acceso a un recurso entre un conjunto compartido.
  - El semáforo implementa un contador con el nº de recursos en el conjunto.
  - Si un hilo bloquea un semáforo, decrementa el contador.
  - Si el contador llega a 0, otros hilos que deseen bloquear el semáforo deben esperar a que se libere.
  - Al liberar un semáforo, se incrementa el contador.
- Los semáforos pueden ser:
  - Sin nombre: entre hilos del mismo proceso, o entre procesos padres/hijos.
  - Con nombre: entre hilos del mismo o distintos procesos.

Semáforos en POSIX

- Llamadas al S.O. para gestionar semáforos:
  - Operación wait: decrementa cuenta y continúa si no ha llegado a 0, o espera si ha llegado a 0.
  - Operación post: incrementa cuenta y libera a otro hilo si sube por encima de 0.

Descripción	**Función** <semaphore.h>
Crear un nuevo semáforo sin nombre	sem_init()
Crear un nuevo semáforo con nombre	sem_open()
Eliminar un semáforo sin nombre	sem_destroy()
Eliminar un semáforo con nombre	sem_unlink()
Desligar un proceso de un semáforo con nombre	sem_close()
Solicita acceso a un semáforo	sem_wait()
Comprueba semáforo y obtiene si está libre	sem_trywait()
Libera un semáforo	sem_post()
Obtiene el valor de la cuenta actual	sem_getvalue()

Semáforos: ejemplo

- Ejemplo productores / consumidor:
  - Una/varias tareas productoras van añadiendo datos a una tabla o lista.
  - Una tarea consumidora va retirando datos de la tabla o lista.
  - Los productores deben esperar para añadir cuando la tabla esté llena.
  - El consumidor se debe esperar para leer cuando la tabla esté vacía.
  - Se usa un semáforo para tabla llena, otro para tabla vacía.

```
struct datos _tabla[NMAX];
pthread_mutex_t _mutex;
sem_t _sem_vacia,_sem_llena;

Hilo_Consumidor()
{
    while (1)
    {
        sem_wait(&_sem_vacia);
        // No está vacía: extraer dato
        pthread_mutex_lock(&_mutex);
        --Extrae dato de tabla y procesa--
        pthread_mutex_unlock(&_mutex);
        sem_post(&_sem_llena);
    }
}

Fn_Hilo_Productor()
{
    sem_wait(&_sem_llena);
    // No está llena: añadir dato
    pthread_mutex_lock(&_mutex);
    --Añade dato a tabla--
    pthread_mutex_unlock(&_mutex);
    sem_post(&_sem_vacia);
}

Init()
{
    sem_init(&_sem_llena,0,NMAX);
    sem_init(&_sem_vacia,0,0);
}
```

Diferencias entre semáforos y mútex

Característica	Mútex	Semáforo
Utilizable entre	Hilos del mismo proceso	Hilos del mismo proceso Hilos de distintos procesos
Cuenta	0 ó 1 (inicialmente 1)	Valor inicial indicado por en la creación.
Propietario del mecanismo	Sólo el hilo que lo bloquea el mútex puede desbloquearlo.	Sin propietario específico: cualquier hilo puede decrementar o incrementar el contador.
Múltiples bloqueos	Múltiples bloqueos por el mismo hilo no tienen efecto.	Múltiples bloqueos por el mismo hilo van decrementando el contador.
Múltiples desbloqueos	Múltiples desbloqueos por el mismo hilo no tienen efecto.	Múltiples bloqueos por el mismo hilo van incrementando el contador.

Señales en POSIX

- Señal: permite que una tarea reciba notificaciones de otra tarea o del S.O., o envíe notificaciones a otra tarea.
- La procedencia de las señales puede ser:
  - Generadas por el S.O. a consecuencia de una interrupción hardware (dispositivo).
  - Generadas por el S.O. a consecuencia de una excepción.
  - Generadas por una tarea a voluntad.
- El servicio de las señales puede ser:
  - Síncrono: espera mediante función específica (no excepciones).
  - Asíncrono: llamada por el S.O. a una función callback, deteniendo temporalmente el hilo.
- Las señales en POSIX pueden ser:
  - Convencionales.
  - De tiempo real.

Señales POSIX

- Manejo de señales:
  - Tipos de señales
  - Bloquear / desbloquear
  - Servicio síncrono / asíncrono
  - Enviar señales
- Tipos de señales:
  - Interrupciones hardware
  - Excepciones
  - Generada por un proceso
- En <signal.h> se definen constantes para las diferentes señales

SIGKILL Elimina el proceso
SIGALRM Alarm clock
SIGSEGV Violación de memoria
SIGUSR1 User defined signal 1
SIGUSR2 User defined signal 2
SIGRTMIN first (highest-priority) realtime signal
SIGRTMAX last (lowest-priority) realtime signal
…

Señales POSIX

- Bloquear/desbloquear señales:
  - *int **pthread_sigmask** (int how, const sigset_t *set, sigset_t *oset);*
  - *how* puede ser: SIG_BLOCK, SIG_UNBLOCK, SIG_SETMASK
- Previamente hay que llenar el conjunto set:
  - Inicializar un conjunto excluyendo todas las señales:
    - *int **sigemptyset**(sigset_t *set);*
  - Inicializar un conjunto incluyendo todas las señales:
    - *int **sigfillset**(sigset_t *set);*
  - Añadir la señal indicada a un conjunto:
    - *int **sigaddset**(sigset_t *set, int signo);*
  - Excluir la señal indicada de un conjunto:
    - *int **sigdelset**(sigset_t *set, int signo);*

Señales POSIX

- Servicio asíncrono de señales:
  - Bloquear señales a manejar
  - Establecer rutina de manejo:
    - *int **sigaction**(int sig, const struct sigaction *act, struct sigaction *oact);*
  - Desbloquear señales a manejar
- *struct sigaction {*

```
int sa_flags;
void (*sa_handler)(int);    /* Función asíncrona o  SIG_DFL (función por
                               defecto) o SIG_IGN (ignorar) */
sigset_t sa_mask;           /* Señales a bloquear durante ejecución */
} ;
```

Para señales TR, usar *sa_flags=SA_SIGINFO* y cambiar *sa_handler* por sig_action:

*void (*sa_sigaction)(int, siginfo_t *, void *);*

Señales POSIX

- Servicio síncrono de señales:
  - Bloquear señales a manejar
  - Esperar con:
    - *int **sigwait**(sigset_t *set,int* sig); /* Señales no T.R. */*
  - O bien para señales en T.R.:
    - *int **sigwaitinfo**(const sigset_t *set, siginfo_t *info);*
    - *int **sigtimedwait**(const sigset_t *set, siginfo_t *info, const struct timespec *timeout);*
- Envío de señales:
  - A un proceso:
    - *int **kill**(pid_t pid,int signo);*
    - *int **sigqueue**(pid_t pid, int signo, const union sigval value);*
  - A un hilo determinado (del mismo proceso):
    - *int **pthread_kill**(pthread_t thread, int sig);*

Ejemplo señales

- Ejemplo de servicio síncrono de una señal:
  - Solicita una lectura en dispositivo no bloqueante.
  - Realiza otras operaciones.
  - Espera la terminación de la operación de E/S

```
…
int main()
{
...
}
```


Excepciones

- Ejemplo de manejo de señales para servir una excepción de acceso a memoria:
  - Una excepción siempre se sirve de forma asíncrona.
  - No se debe regresar normalmente después de una excepción: setjmp() / longjmp()

```
…
void ErrorMemoria(int num_senyal, siginfo_t
     *info_senyal, void *no_usar)
{
    if (num_senyal==SIGSEGV)
    {
        printf("<<Error acceso memoria>>\n");
        ...
    }
}

int main()
{
    sigset_t senyales;
    int err;
    struct sigaction accion_errmem;
    int x[2];
    char c;

    err=sigemptyset(&senyales);
    err=sigaddset(&senyales,SIGSEGV);
    err=pthread_sigmask(SIG_BLOCK,&senyales,NULL);

    accion_errmem.sa_flags=SA_SIGINFO;
    accion_errmem.sa_sigaction=ErrorMemoria;
    accion_errmem.sa_mask=0;
    err=sigaction(SIGSEGV,&accion_errmem,NULL);

    err=pthread_sigmask(SIG_UNBLOCK,&senyales,NULL);
    x[1000000]=100;
    ... // Este código jamás se ejecutará
}
```

Tiempo en POSIX

- Conocer el tiempo absoluto y relativo
- Generar retraso relativo y absoluto
- Temporizaciones periódicas: generan SIGALRM
  - Servicio síncrono: sigwait()
  - Servicio asíncrono: sigaction()

- Tiempo absoluto y relativo (time.h):
  - *time_t*: segundos desde 1/1/1970, 00:00 h.
    - entero con signo (int): 2^{31} seg ≡ 68 años
    - nueva especificación de 64 bits
  - *struct timespec {*
    time_t tv_sec; / segundos */*
    long tv_nsec; / nanosegundos */*
    } ;

Tiempo en POSIX

- *clockid_t:* Identificador de reloj
- Puede haber varios relojes hard y soft; se suele utilizar el reloj en tiempo real (*clockid=CLOCK_REALTIME*)
- Conocer el tiempo absoluto:
  - *int **clock_gettime** (clockid_t clockid, struct timespec *tp);*
- Establecer tiempo absoluto:
  - *int **clock_settime** (clockid_t clockid, const struct timespec *tp);*
- Obtener la resolución del reloj:
  - *int **clock_getres** (clockid_t clockid, struct timespec *res);*
- Retardos relativos:
  - *unsigned int **sleep** (unsigned int seconds);* (a nivel de proceso)
  - *int **nanosleep** (const timespec_t *rqtp, timespec_t *rmtp);* (a nivel de hilo; también puede ser despertado por una señal)
- Retardos absolutos: usar *clock_gettime()*, calcular el nº de segundos o nanosegundos, usar *sleep()* o *nanosleep().*

Tiempo en POSIX

- Temporizaciones periódicas:
  - *timer_t:* Identificador de temporizador
  - *struct itimerspec {*
    struct timespec it_interval; / periodo */*
    struct timespec it_value; / expiración */*
    };
- Están asociadas a la señal SIGALRM (por defecto)
- Crear temporizador:
  - *int **timer_create** (clockid_t clock_id, struct sigevent *evp, timer_t *timerid);*
- Establecer periodos del temporizador:
  - *int **timer_settime** (timer_t timerid, int flag, const struct itimerspec *value, struct itimerspec *ovalue);*
    (usar flag=TIMER_ABSTIME para tiempo absoluto)
- Eliminar temporizador:
  - *int **timer_delete** (timer_t timerid);*

Mensajes en POSIX

- Cola de mensajes: buzón intermedio donde se pueden dejar y recoger mensajes.
- Los mensajes los pueden enviar y recibir varios procesos / hilos
- Cabecera: mqueue.h
- Descriptor de cola de mensajes: *mqd_t;*
- Atributos de cola de mensajes:
  - struct **mq_attr** {
    long mq_flags; */* message queue flags (0 or O_NONBLOCK)*/*
    long mq_maxmsg; */* maximum number of messsages */*
    long mq_msgsize; */* maximum message size */*
    long mq_curmsgs */* number of messages queued */*
    } ;
- Colas de mensajes: siempre con nombre

Colas de mensajes POSIX

- Funciones de cola de mensajes:
  - Crear cola de mensajes:
    - *mqd_t **mq_open**(const char *name, int oflag, mode_t mode, struct mq_attr *attr);*
      name: nombre de archivo válido
      oflag: O_CREAT | otros flags (O_RDONLY, O_WRONLY, O_RDWR, O_NONBLOCK, O_EXCL).
  - Abrir cola de mensajes existente:
    - *mqd_t **mq_open**(const char *name, int oflag);*
      name: nombre de una cola que ya exista
      oflag: no debe tener O_CREAT
  - Cerrar cola de mensajes:
    - *int **mq_close**(mqd_t mqdes);*
  - *Eliminar cola de mensajes:*
    - *int **mq_unlink**(mqd_t mqdes);*

Colas de mensajes POSIX

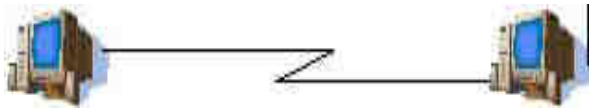
- Envío de mensage:
  - *int **mq_send**(mqd_t mqdes, const char *msg_ptr, size_t msg_len, unsigned int msg_prio);*
    (se bloquea si no hay sitio en la cola)
- Recepción de mensaje:
  - *size_t **mq_receive**(mqd_t mqdes, char *msg_ptr, size_t msg_len, unsigned int *msg_prio);*
    (se bloquea si no hay mensajes en la cola)
- Establecer un aviso asíncrono:
  - *mqd_t **mq_notify(**mqd_t mqdes**,** const struct sigevent ***notification**);*

Comunicaciones

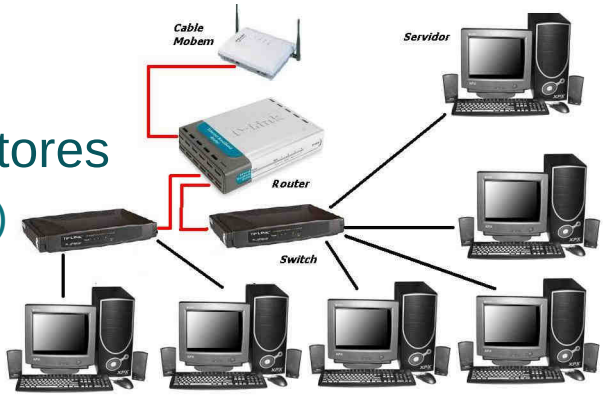
- Comunicaciones:
  - Mecanismo(s) de intercambio de información entre 2 ó más computadores conectados entre sí o a través de otros.
- Terminología:
  - **Trama**: unidad de información a transmitir
  - **Medio**: elemento físico por el que circula la información, y características de dicho medio.
  - **Interfaz**: conexión del computador al medio físico.
  - **Emisor**: dispositivo/programa que genera y envía una trama.
  - **Receptor**: dispositivo/programa que debe recibir y procesar una trama.
  - **Cliente**: dispositivo/programa que inicia una secuencia de comunicación.
  - **Servidor**: dispositivo/programa que espera una comunicación.

Comunicaciones

- Punto a punto: sólo hay un emisory un receptor
  - Serie (RS-232,USB)
  - Paralelo
  - Inalámbrica (Bluetooth)

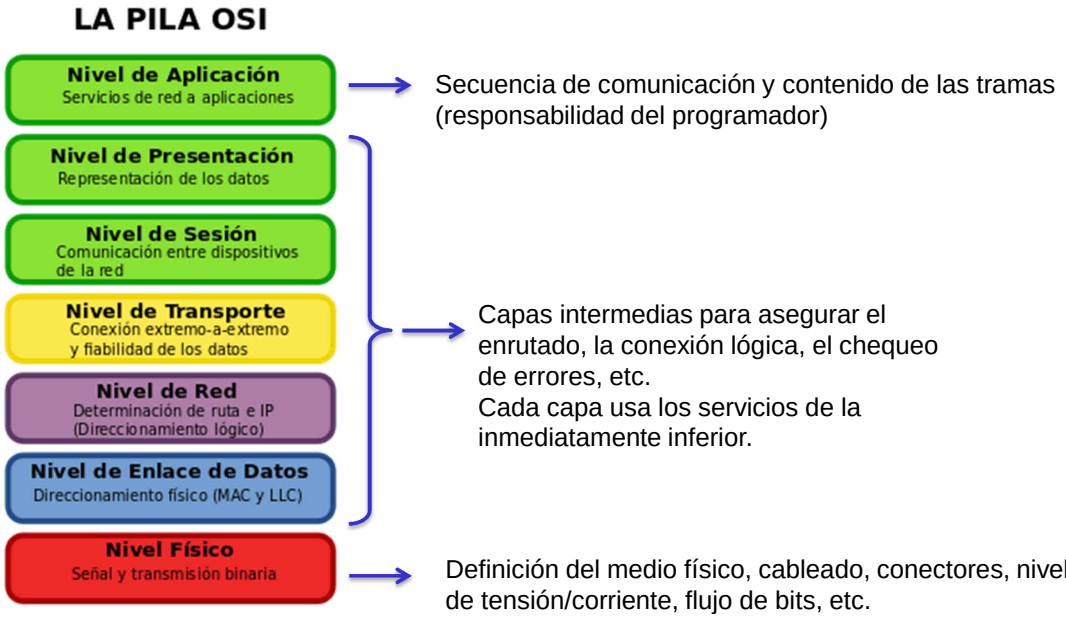


- Multipunto: un emisor, múltiples posibles receptores
  - Serie (I2C, CAN, RS-485)
  - Red (Ethernet)
  - Inalámbrica (Wifi)



Comunicaciones

- Especificación de comunicaciones: modelo de capas (ISO/OSI)



Comunicaciones

- La programación de comunicaciones a nivel de capa de aplicación:
  - Codificar en tramas los contenidos a transmitir.
  - Definir la secuencia de comunicación: establecimiento de conexión, envío/recepción, fin de conexión.
  - Identificar al receptor en comunicaciones multi-punto.
  - Establecer protocolo de acceso al medio si no existe en las capas inferiores.
  - Establecer método de detección/corrección de errores si no existe en las capas inferiores.
- Equipos “pequeños” (sin S.O.): programación del interfaz a nivel de puertos e interrupciones.
- Equipos “grandes” (con S.O.): programación a nivel de dispositivos de E/S.

Comunicaciones en red TCP/IP

- **Redes de ordenadores**: grupo de ordenadores autónomos interconectados
  - Interconexión de dos o más ordenadores: situación en la que éstos son capaces de intercambiar información.
  - Autónomos: excluye aquellos casos donde existe una clara relación maestro/esclavo.

- Utilidades de las redes de Ordenadores para las empresas:
  - Compartición de recursos
  - Mayor fiabilidad
  - Reducción de costes
  - Mayor flexibilidad
  - Simplificación y agilización de comunicación

- Utilidades de las redes de Ordenadores para el público general:
  - Acceso a información remota
  - Una nueva forma de comunicación personal
  - Nuevas formas de entretenimiento

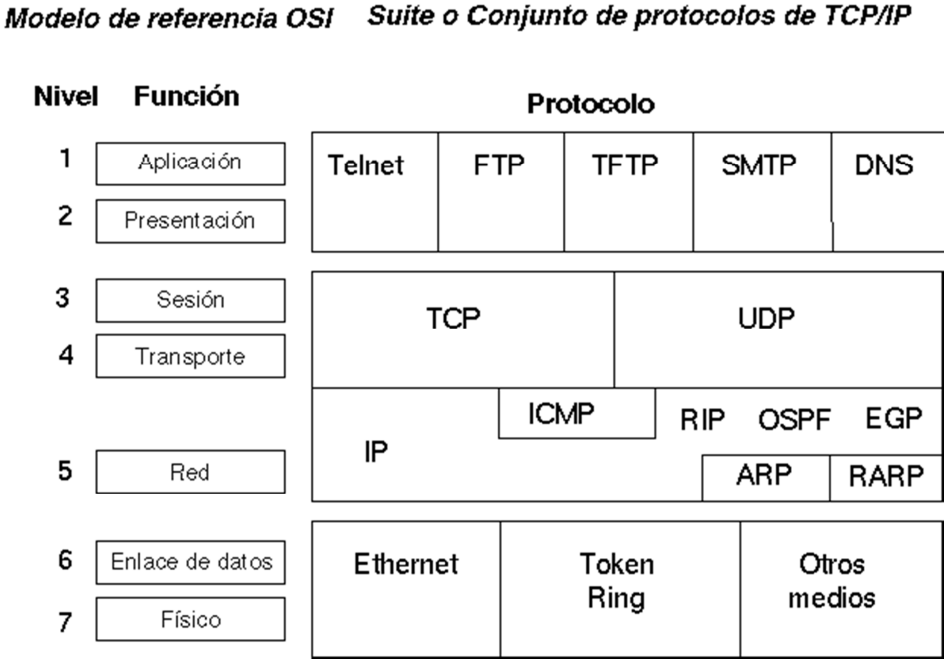
- Utilidades de las redes de Ordenadores para la industria:
  - Implementación del control distribuido

Comunicaciones en red TCP/IP

- Sistema de comunicación: conjunto de hardware y software que permite la comunicación entre estaciones.
- Funciones del sistema de comunicación:
  - Identificar las estaciones que conforman la red
  - Establecimiento de conexiones y multiplexación de canales
  - Control de errores durante la comunicación
  - Fragmentación y reconstrucción de los mensajes
  - Compactación de mensajes
  - Manejo de congestiones y control del flujo de la información
  - Sincronización
  - Establecimiento de distintos niveles de prioridad

Comunicaciones en red TCP/IP

- Modelo de capas ISO/OSI



Comunicaciones en red TCP/IP

- Protocolo de red IP (Internet Protocol):
  - Identificación de un nodo en la red: dirección IP, formada por 4 bytes
  - Identificación de un programa usuario del nodo: nº de puerto
- Protocolo de transporte con conexión: TCP
  - Conexión punto a punto (dirección IP + nº puerto)
  - Conexión -> envío de datos -> cierre conexión
  - Conexión segura: el protocolo asegura que no hay pérdida de mensajes, el troceado, la reconstrucción y el orden.
- Protocolo de transporte sin conexión: UDP
  - No hay conexión
  - Envío y recepción de datos a cualquier punto (dirección IP + nº puerto)
  - Envío y recepción no seguros

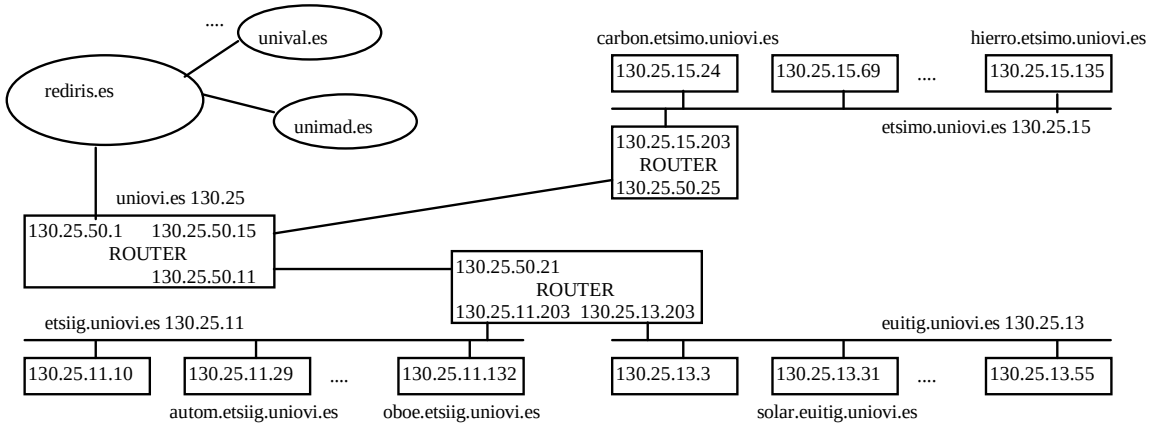
Puertos IP

- **Puertos estándar:**
  20 FTP data (File Transfer Protocol)
  21 FTP (File Transfer Protocol)
  22 SSH (Secure Shell)
  23 Telnet
  25 SMTP (Send Mail Transfer Protocol)
  43 whois
  53 DNS (Domain Name Service)
  68 DHCP (Dynamic Host Control Protocol)
  79 Finger
  80 HTTP (HyperText Transfer Protocol)
  110 POP3 (Post Office Protocol, version 3)
  115 SFTP (Secure File Transfer Protocol)
  119 NNTP (Network New Transfer Protocol)
  123 NTP (Network Time Protocol)
  137 NetBIOS-ns
  138 NetBIOS-dgm
  139 NetBIOS
  143 IMAP (Internet Message Access Protocol)
  161 SNMP (Simple Network Management Protocol)
  194 IRC (Internet Relay Chat)
  220 IMAP3 (Internet Message Access Protocol 3)
  389 LDAP (Lightweight Directory Access Protocol)
  443 SSL (Secure Socket Layer)
  445 SMB (NetBIOS over TCP)
  666 Doom
  993 SIMAP (Secure Internet Message Access Protocol)
  995 SPOP (Secure Post Office Protocol)

- **Aplicaciones de usuario:**
  1024 a 65535

Comunicaciones en red TCP/IP

- Protocolo de encaminamiento: IP
  - Permite hacer llegar los paquetes a su destino a través de routers o encaminadores
  - Dirección IP compuesta por 4 números de 8 bits: x.x.x.x
  - Servicio de nombres (DNS) para evitar conocer la dirección IP de cada nodo



Comunicaciones en red TCP/IP

- ***socket***: identificación de una conexión punto a punto entre 2 dispositivos conectados en red (dirección + puerto).
- Declaración de identificador:

```
int sock;
```

- 1er paso: crear id. válido de socket (tipo SOCK_STREAM para TCP, tipo SOCK_DGRAM para UDP):

```
sock=socket(AF_INET,SOCK_STREAM,0);
```

- 2º paso: asignar puerto local al socket:

```
struct sockaddr_in add_local;
… Rellenar campos de add_local: dirección IP, puerto
  (network order), protocolo
err = bind(sock,(struct sockaddr*) &add_local, sizeof(struct
  sockaddr_in));
```

- Si no ha habido errores, el socket ya puede ser utilizado para:
  - Conectar + enviar/recibir + desconectar (TCP)
  - Enviar/recibir (UDP)

Comunicaciones en red TCP/IP

- Programación de sockets sin conexión (UDP)

- Enviar datos:

```
ssize_t sendto(int sockfd, const void *buf, size_t len,
                   int flags, const struct sockaddr *dest_add,
                   socklen_t add_len);
ssize_t send (int sockfd, const void *buf, size_t len,int flags);
```

- Recibir datos:

```
ssize_t recv(int sockfd, void *buf, size_t len, int flags);

ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
                   struct sockaddr *src_addr, socklen_t *addrlen);
```

- Cerrar el socket:

```
close(sock);
```

- No hay seguridad de que los datos hayan llegado aunque err==0
- Si se envían varios mensajes, no hay seguridad del orden de llegada (pueden seguir rutas distintas)
- Si el mensaje es demasiado grande, es responsabilidad del remitente trocearlo y enviarlo, y del receptor componer los trozos.

Comunicaciones en red TCP/IP

- Programación de sockets con conexión (TCP)

- Establecimiento de conexión: modelo cliente/servidor.
  - Cliente: solicitante de una conexión (ej. navegador web)
  - Servidor: receptor de una solicitud de conexión (ej. servidor de páginas web).
- Hasta que no se establezca una conexión no se pueden enviar y recibir datos
- Se asegura la llegada de los mensajes y su orden
- Se trocean y recomponen automáticamente los mensajes demasiado grandes.

LADO CLIENTE

- Solicitud de conexión:

```
struct sockaddr_in addr_servidor;
… Rellenar campos de addr_servidor …
err=connect(sock,(struct sockaddr*)
&addr_servidor,sizeof(struct
sockaddr_in));
```

- Si no hay error, se ha establecido la conexión: se pueden enviar y recibir datos con send() y recv()
- Cerrar conexión con close()

LADO SERVIDOR

- Admitir solicitudes de conexión: listen()

```
err=listen(sock,n_conex_max);
```

- Esperar por una conexión: accept()

```
int conectado;
struct sockaddr_in addr_cliente;
int len=sizeof(struct sockaddr_in);
conectado=accept(sock,(struct
sockaddr*) &addr_cliente,&len);
```

- Si no hay error, se ha establecido la conexión con el socket 'conectado': se pueden enviar y recibir datos con send() y recv().
- Cerrar conexión con close(conectado);