

Tema 2

Llamadas al sistema



Contenidos

PARTE 1

1. Arquitectura básica
2. Entrada/salida sin sistema operativo
3. Rutinas residentes

PARTE 2

4. Las rutinas residentes como embrión del sistema operativo
5. Un mecanismo de acceso unificado
6. Las llamadas al sistema como interfaz para las aplicaciones

Bibliografía

- C. Rodríguez, I. Alegria, J. González, A. Lafuente: *Descripción Funcional de los Sistemas Operativos*. Síntesis, 1994. [Tema 1 y Apartado 2.1]
- W. Stallings: *Sistemas Operativos*. 5º Ed. Prentice-Hall, 2005. [Apartados 1.2, 1.3, 1.4 y 1.7]
- F.M. Márquez: *UNIX. Programación Avanzada* 3ª Edición. Rama, 2004. [Tema 1]

Tema 2

Llamadas al sistema

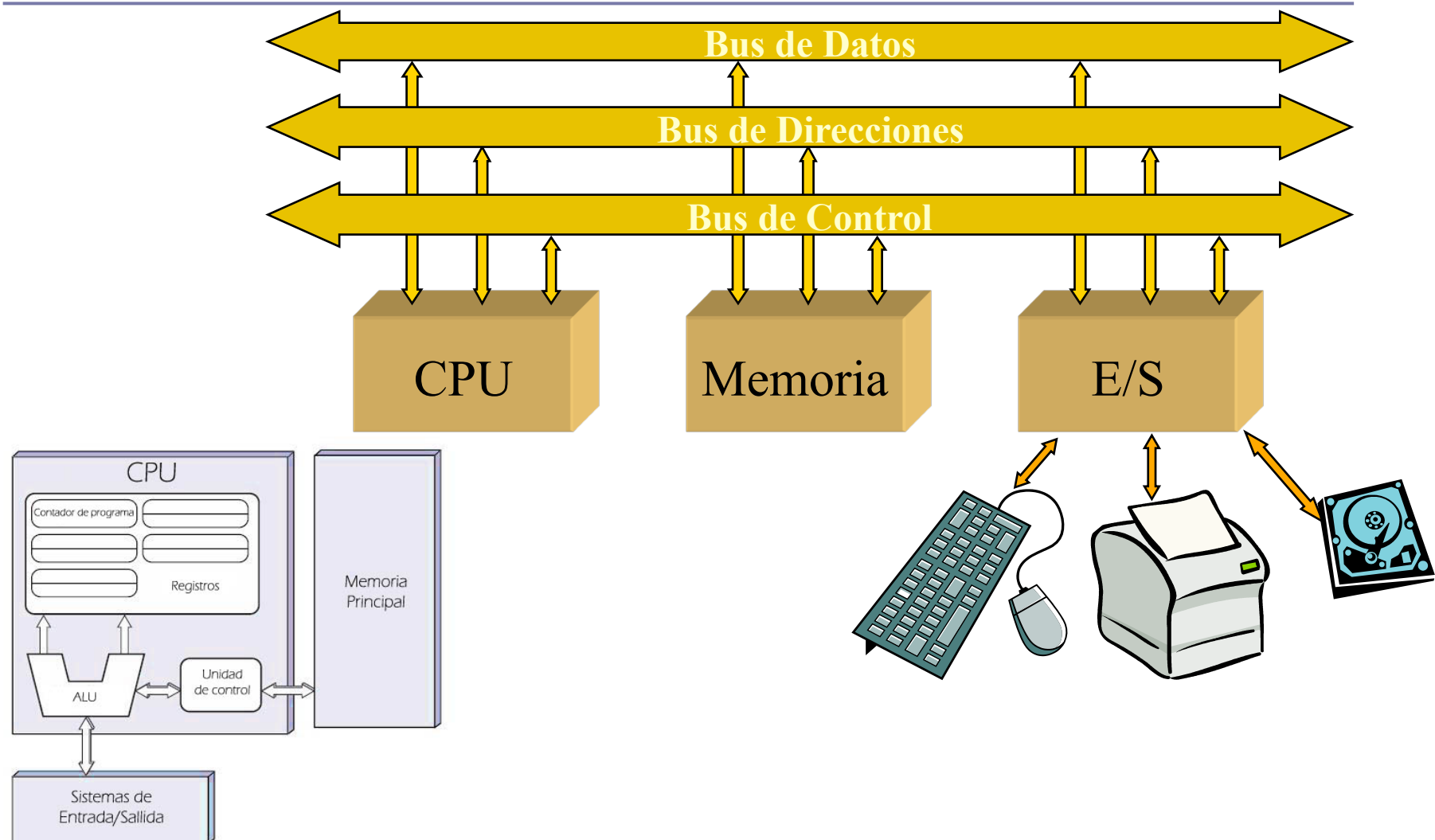


Primera parte:
Cómo controlar el hardware

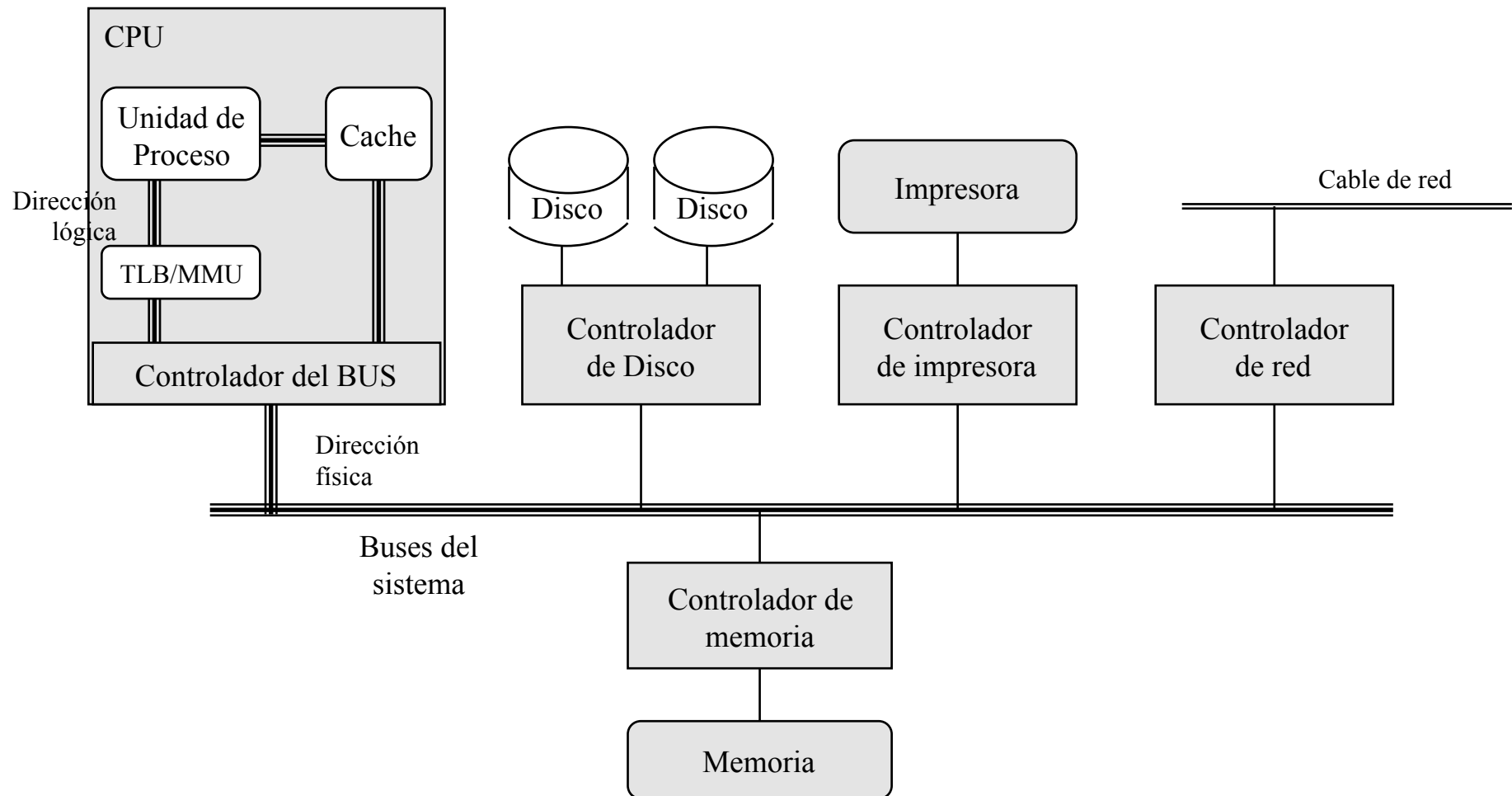
Qué vamos a aprender

Cómo sería la vida del desarrollador
de aplicaciones sin Sistema
Operativo

Arquitectura Von Neumann



Arquitectura básica: controladores



Sin sistema operativo:

Formas de controlar la entrada/salida

- Por encuesta
- Por interrupciones
- Por acceso directo a memoria (DMA)

E/S por encuesta

- Pasos:
 - Mirar si se puede leer/escribir, analizando el registro de estado
 - Cuando esté listo, leer/escribir el dato
 - Indicar al controlador que la lectura/escritura se ha realizado
- Para conocer el estado del controlador, se *encuesta* el registro de estado mediante un bucle de *espera activa* (*busy waiting*)
- Se debe analizar el registro de estado, y en el caso de errores, llamar a una rutina para que gestione los errores

E/S por encuesta

Rutina entrada DIS1

```
rut_entrada_1 (char *vector, int num)
{
    int j, estado;

    for (j=0; j<num; j++) {
        while ((estado=leer_reg_estado(DIS1)) ==
                NO_PREPARADO) NOP;

        errores(DIS1, estado);
        vector[j]= leer_reg_datos(DIS1);
        escribir_reg_control(DIS1,LEIDO);
    }
    return j;
}
```

Rutina salida DIS2

```
rut_salida_2 (char *vector, int num)
{
    int j, estado;

    for (j=0; j<num; j++) {
        while ((estado=leer_reg_estado(DIS2)) ==
                NO_PREPARADO) NOP;

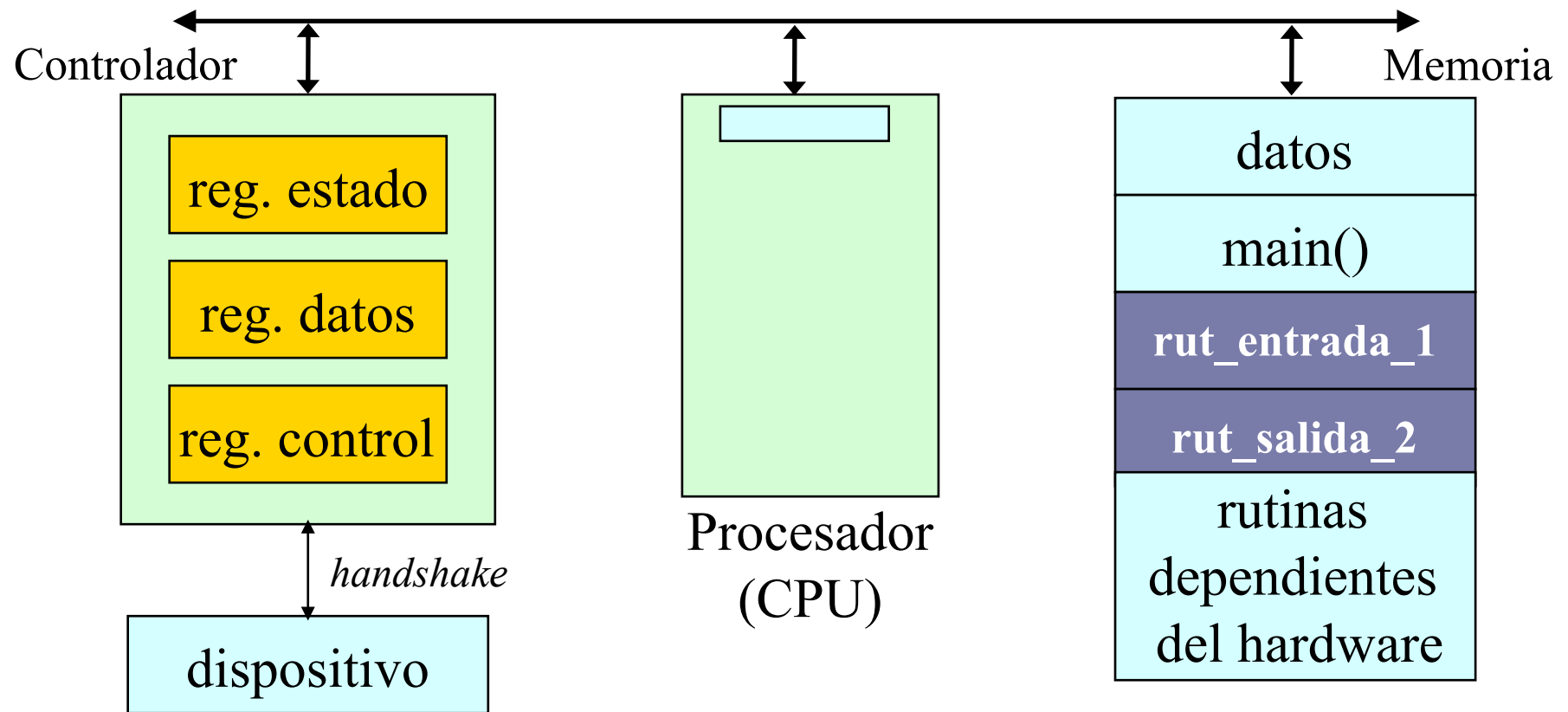
        errores(DIS2, estado);
        escribir_reg_datos(DIS2, vector[j]);
        escribir_reg_control(DIS2,ESCRITO);
    }
    return j;
}
```

Programa usuario

```
main ()
{
    char buff[80];

    while (TRUE)
    {
        rut_entrada_1(buff,80);
        rut_salida_2(buff,80);
    }
}
```

Arquitectura soporte de la E/S por encuesta



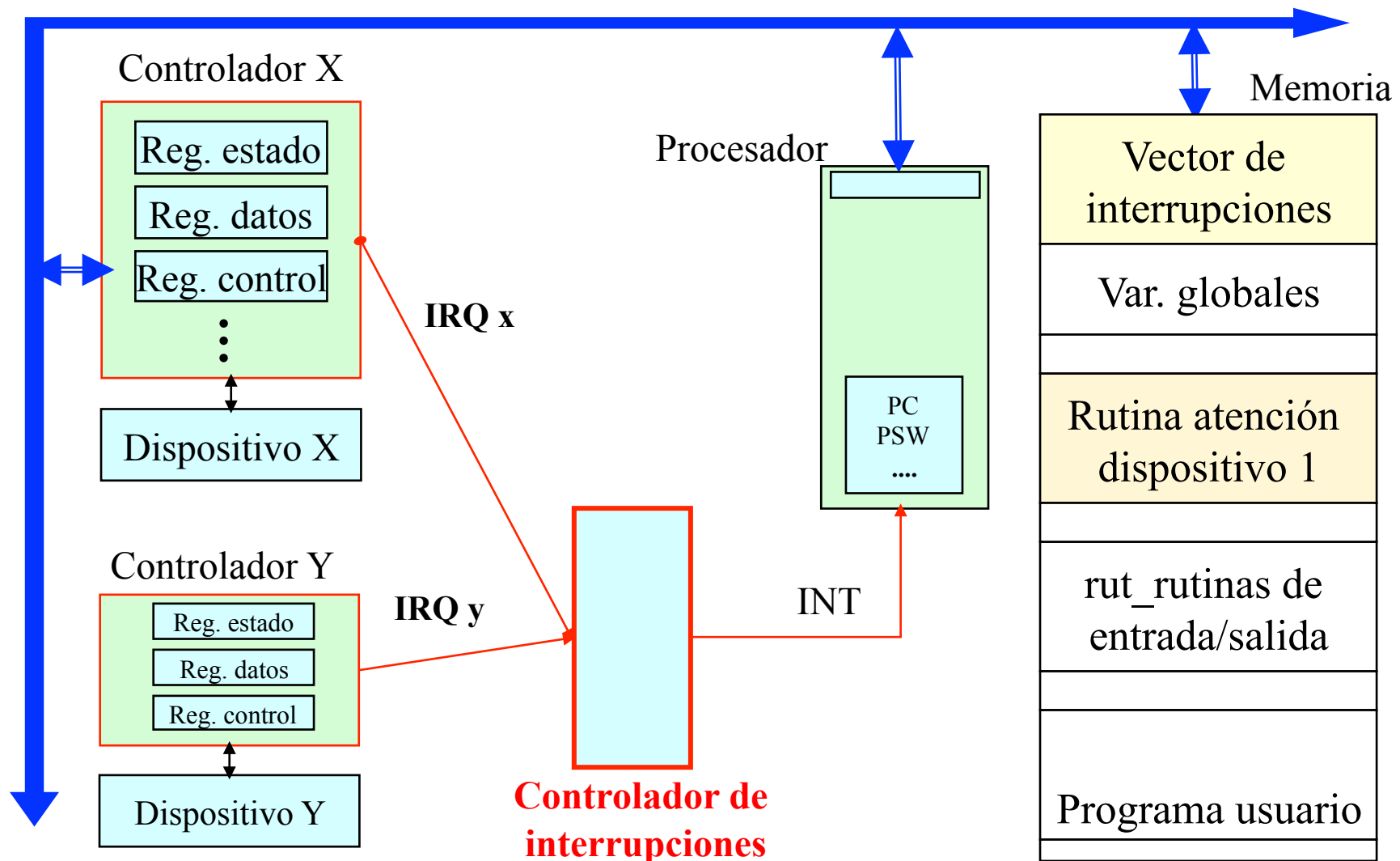
Problemas de la E/S por Encuesta

- Mientras se realiza la E/S la CPU está ocupada por una espera activa, realizando una tarea poco útil.
- El tiempo de espera puede ser muy largo
 - Dispositivos de E/S muy lentos (comparados con la CPU)
- ¿Cómo podemos gestionar la E/S de forma que la CPU no pierda su tiempo en la espera activa?
 - ➔ *Yo sigo con lo mío y ya me interrumpirá el controlador cuando el dispositivo esté listo.*

E/S por interrupción

- **Interrupción.** Evento producido en un computador que afecta al flujo de ejecución de las instrucciones.
- Tipos de interrupciones
 - Hardware
 - Generadas por un evento hardware externo:
 - Evento de reloj
 - Dispositivo de E/S (indica que ha ocurrido algo en el dispositivo)
 - Errores hardware (error del bus)
 - Generadas como consecuencia de la ejecución de un programa:
 - Errores software (overflow, instrucción desconocida, ref. a memoria errónea...)
 - Software o *traps*
 - Generadas por la ejecución de una instrucción especial

Arquitectura soporte a E/S por interrupción



E/S por interrupción

- ❑ Ofrece *paralelismo*: las operaciones pueden ser tanto **síncronas** como **asíncronas**
- ❑ Además de las rutinas de entrada y salida se debe suministrar una **rutina de atención**, que se ejecutará al producirse la interrupción
- ❑ El dispositivo, cada vez que lee o escribe un byte/bloque genera una **interrupción**

E/S por Interrupción (modo síncrono)

programa usuario

```
main ()
{
    char buff[80];

    modificar_entrada_vector(DIS1,rutina_atencion_1);
    modificar_entrada_vector(DIS2,rutina_atencion_2);
    while (TRUE)
    {
        rut_entrada_1(buff,80);
        rut_salida_2(buff,80);
    }
}
```

rutina entrada DIS1

```
rut_entrada_1 (char *vector, int num)
{
    final1 = FALSE; buff1 = vector;
    indice1=0;
    varnum1=num;
    while ((final1)==FALSE) nop;
}
```

Variables globales

```
int final1=TRUE, final2=TRUE;
char *buff1, *buff2;
int varnum1=0, varnum2=0;
int indice1, indice2;
```

rutina salida DIS2

```
rut_salida_2 (char *vector, int num)
{
    int estado;
    final2 = FALSE;
    // primera escritura por encuesta
    while ((estado=leer_reg_estado(DIS2)) ==
            NO_PREPARADO) NOP;

    errores(DIS2, estado);
    buff2= vector; indice2=1; varnum2=num;
    escribir_reg_datos(DIS2, buff2[0]);
    escribir_reg_control(DIS2, ESCRITO);
    while ((final2)==FALSE) nop;
}
```

E/S por Interrupción (rutinas de atención)

```
// rutina atención dispositivo 1
void interrupt rut_atencion_1 ()
{
    int estado;
    if (varnum1!=0)
    {
        estado=leer_reg_estado(DIS1);
        errores(DIS1, estado);
        buff1[indice++] = leer_reg_datos(DIS1);
        varnum1--;
        if (varnum1==0) final1=TRUE;
    } /* else lectura anticipada */
    escribir_reg_control(DIS1,LEIDO);

    fin_rut_int();//eoi(); iret
}
```

```
// rutina atención dispositivo 2
void interrupt rut_atencion_2 ()
{
    int estado;
    estado=leer_reg_estado(DIS2);
    errores(DIS2, estado);
    varnum2--;
    if (varnum2>0)
    {
        escribir_reg_datos(DIS2, buff2[indice2++]);
        escribir_reg_control(DIS2,ESCRITO);
    }
    else final2=TRUE;

    fin_rut_int() //eoi(); iret
}
```


E/S por Interrupción (síncrono/asíncrono)

rutina entrada DIS1

```
rut_entrada_1 (char *vector, int num, int asin_sin)
{
    final1 = FALSE; buff1 = vector;
    indice1=0;
    varnum1=num;
    if (asin_sin== SINCRO)
        sincronizacion(&final1);
}
```

rutina de sincronización

```
void sincronización (int *final)
{
    while ((*final)==FALSE) nop;
}
```

Variables globales

```
int final1=TRUE, final2=TRUE;
char *buff1, *buff2;
int varnum1=0, varnum2=0;
int indice1, indice2;
```

rutina salida DIS2

```
rut_salida_2 (char *vector, int num, int asin_sin)
{
    int estado;
    final2 = FALSE;
    // primera escritura por encuesta
    while ((estado=leer_reg_estado(DIS2)) ==
            NO_PREPARADO) NOP;

    errores(DIS2, estado);
    buff2= vector; indice2=1; varnum2=num;
    escribir_reg_datos(DIS2, buff2[0]);
    escribir_reg_control(DIS2, ESCRITO);
    if (asin_sin== SINCRO)
        sincronizacion(&final2);
}
```

E/S por Interrupción (síncrono/asíncrono)

```
// Programa usuario (síncrono)
```

```
main ()
{
    char buff[80];

    modificar_entrada_vector(DIS1, rut_atencion_1);
    modificar_entrada_vector(DIS2, rut_atencion_2);
    while (TRUE)
    {
        rut_entrada_1(buff, 80, SINCRO);
        rut_salida_2(buff, 80, SINCRO);
    }
}
```

rutina de sincronización

```
void sincronización (int *final)
{
    while ((*final)==FALSE) nop;
}
```

```
//Programa usuario (asíncrono)
```

```
main ()
{
    char vec1[80], vec2[80];

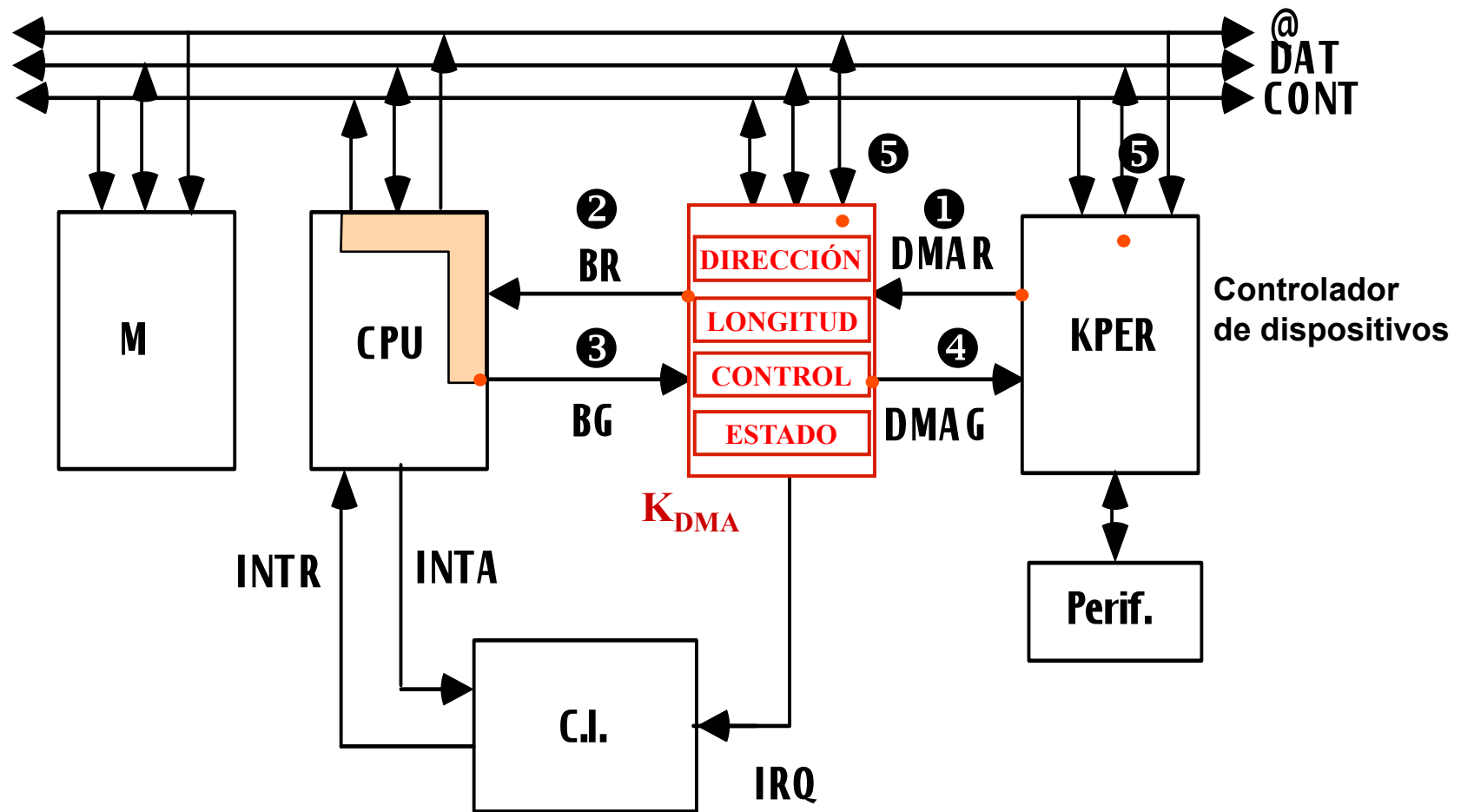
    modificar_entrada_vector(DIS1, rut_atencion_1);
    modificar_entrada_vector(DIS2, rut_atencion_2);
    while (TRUE)
    {
        rut_entrada_1 (vec1, 80, SINCRO);
        sincronización(&final2);
        rut_salida_2(vec1, 80, ASINC);
        rut_entrada_1 (vec2, 80, SINCRO);
        sincronización(&final2);
        rut_salida_2(vec2, 80, ASINC);
    }
}
```

Acceso Directo a Memoria

(DMA: Direct Memory Access)

- ❑ ¿Interrumpir a la CPU por cada byte transferido?
- ❑ DMA se utiliza para E/S de *bloques* de información con dispositivos rápidos (discos, comunicación...)
- ❑ Transfiere datos directamente entre el dispositivo de E/S y la memoria sin pasar por la CPU
- ❑ Requiere un controlador de DMA
 - El controlador genera interrupciones de fin de transferencia DMA.

Arquitectura soporte del DMA



Entrada/Salida por Acceso Directo a Memoria (DMA)

- El dispositivo y el hardware se encargan de realizar toda la transferencia de forma **paralela**
- Los buses de acceso a memoria se comparten entre la CPU y el K_{DMA}
 - Protocolo de acceso al bus:
 - Por robo de ciclo
 - Por ráfaga
- El K_{DMA} genera una **interrupción** al finalizar la transferencia del bloque de datos. El programador debe programar la **rutina de atención** del DMA (o del dispositivo)

Entrada/Salida por Acceso Directo a Memoria (DMA)

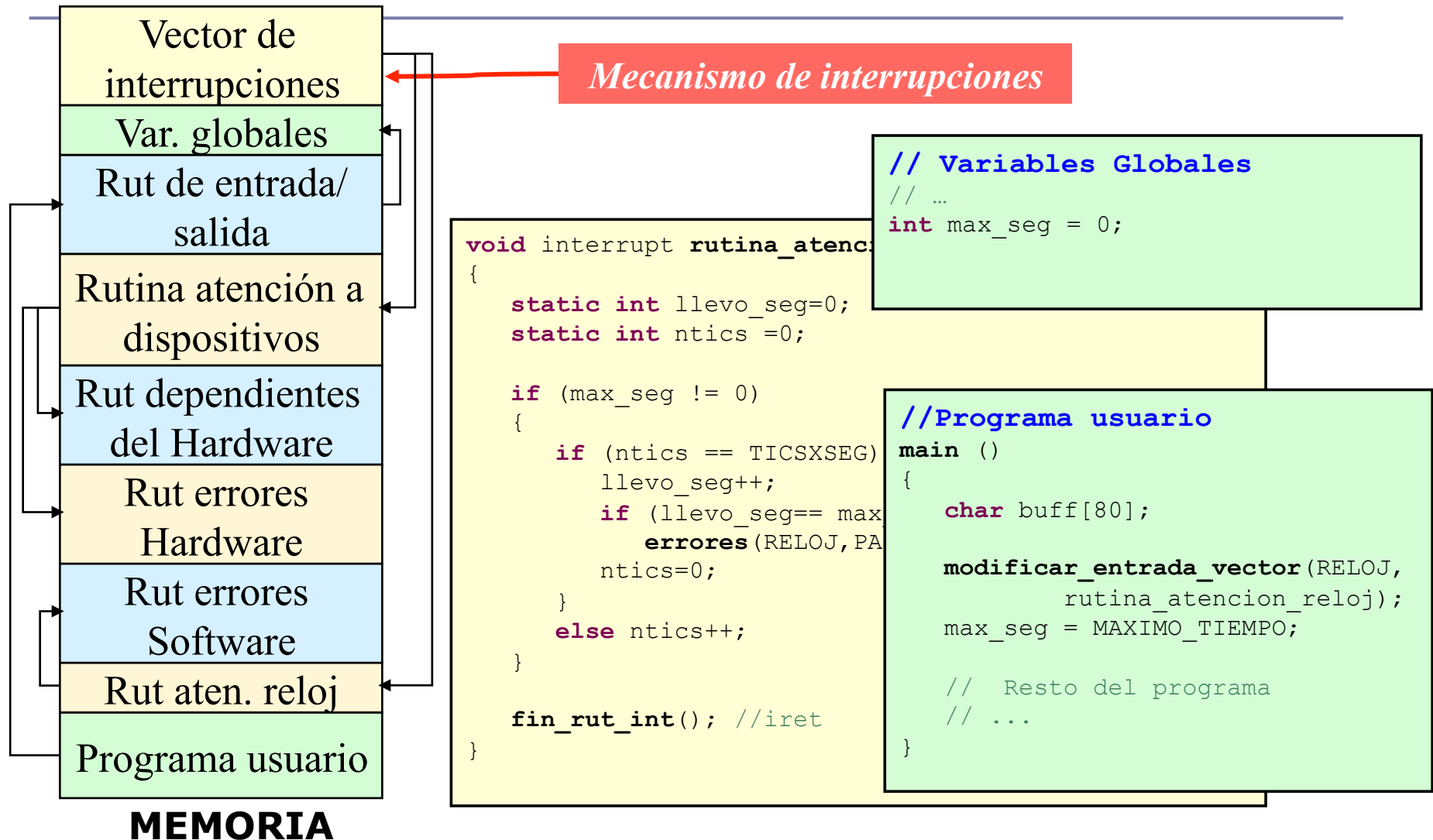
```
// rutina atención DIS2
void interrupt rutina_atención_DIS2()
{
    int estado;
    estado = leer_reg_estado(KDMA);
    errores(KDMA, estado);
    final2 = TRUE;

    fin_rut_int(); //iret
}
```

```
// rutina salida
void rut_salida_2 (char *vector, int num,
                  int asin_sin)
{
    int estado;
    final2 = FALSE;
    while ((estado=leer_reg_estado(KDMA)) ==
           NO_PREPARADO)
        NOP;

    errores(KDMA, estado);
    {
        escribir_reg_direccion(KDMA, &(vector[1]));
        escribir_reg_longitud(KDMA, num-1);
        escribir_reg_control(KDMA, INICIO_ESCRITURA);
        programar_controlador(DIS2, vector[0], num);
        if (asin_sin == SINCRO)
            sincronizacion(&final2);
    }
}
```

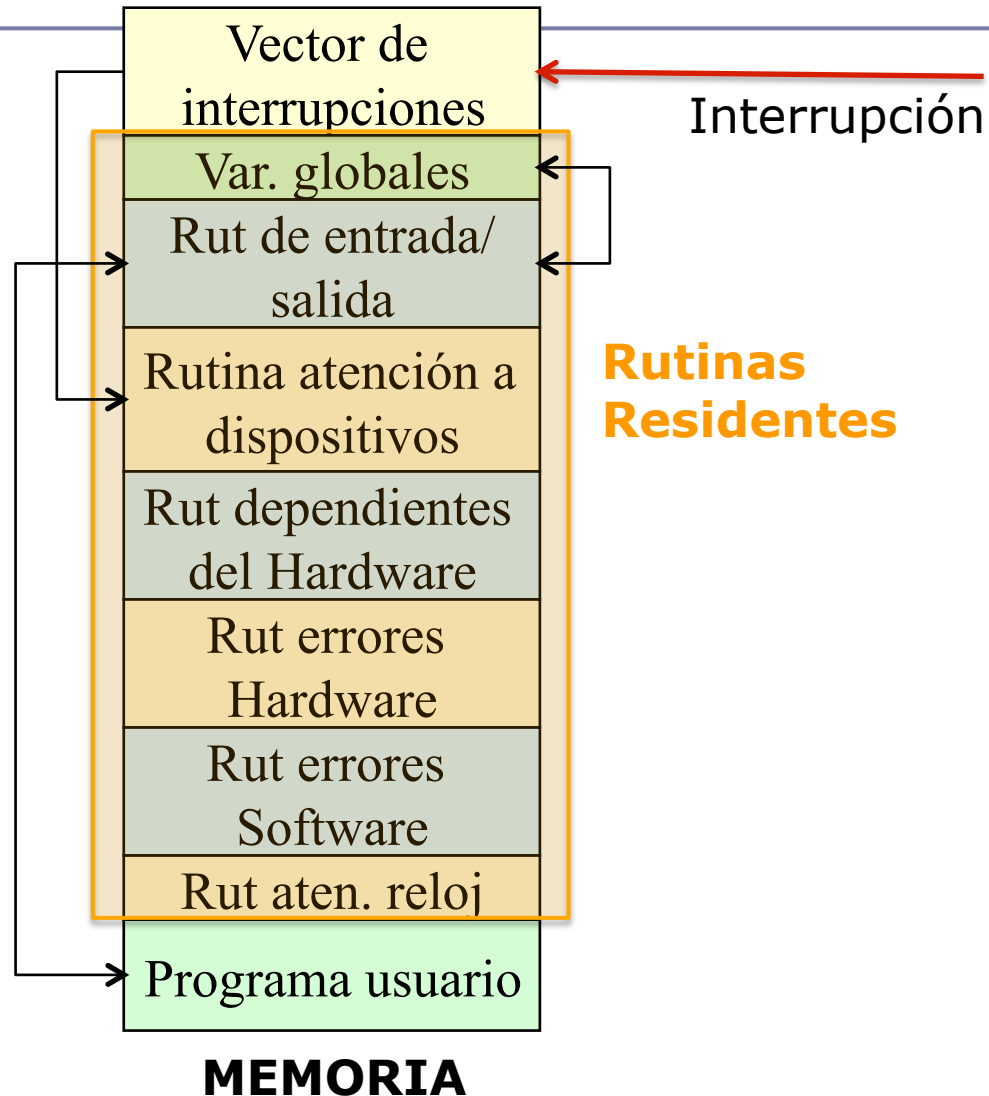
Control de errores y tiempo



Rutinas residentes

- ¿Deben preocuparse los desarrolladores de aplicaciones de los detalles de la programación de los dispositivos?
 - ¿Qué pasa si se cambia el hardware?
 - ¿Quién mantiene las rutinas de E/S, de interrupción, etc?
- ➔ Proporcionar estas rutinas, comunes a las aplicaciones, como **residentes** en memoria y utilizables por todas las aplicaciones

Rutinas residentes




Rutinas residentes

¿Cómo acceden las aplicaciones de forma segura a las rutinas residentes?

Tema 2

Llamadas al sistema

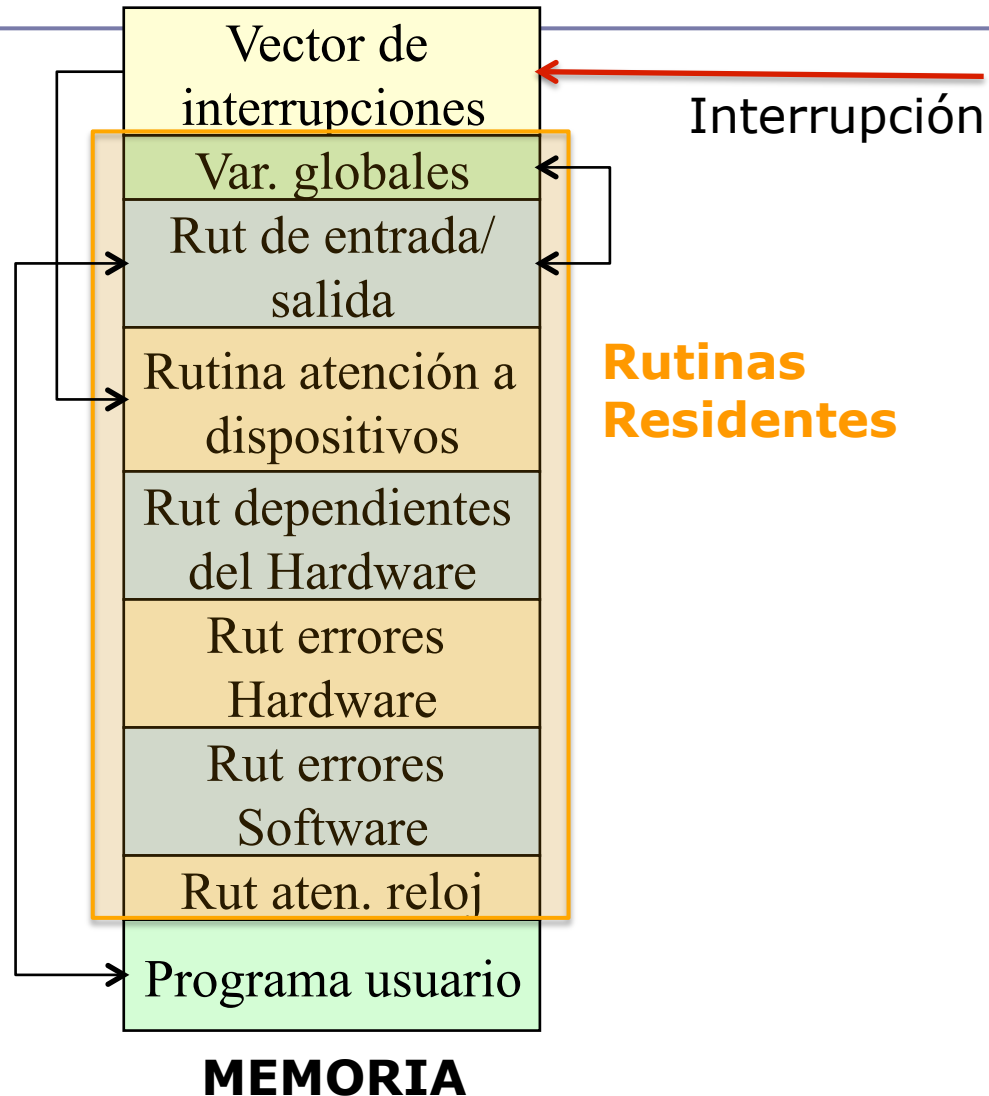


Segunda parte:
Cómo acceder a los servicios
del sistema operativo

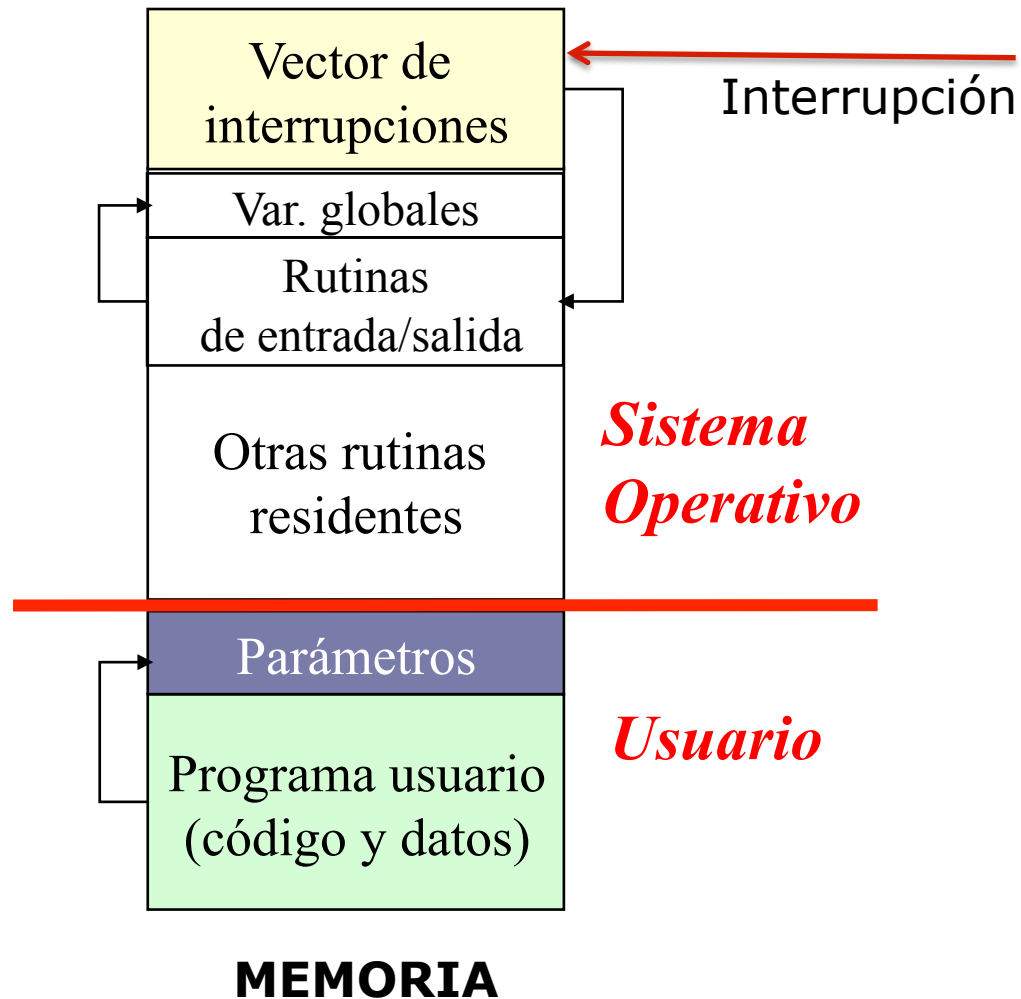
Qué vamos a aprender

Cómo el sistema operativo ofrece
sus servicios a las aplicaciones de
forma segura

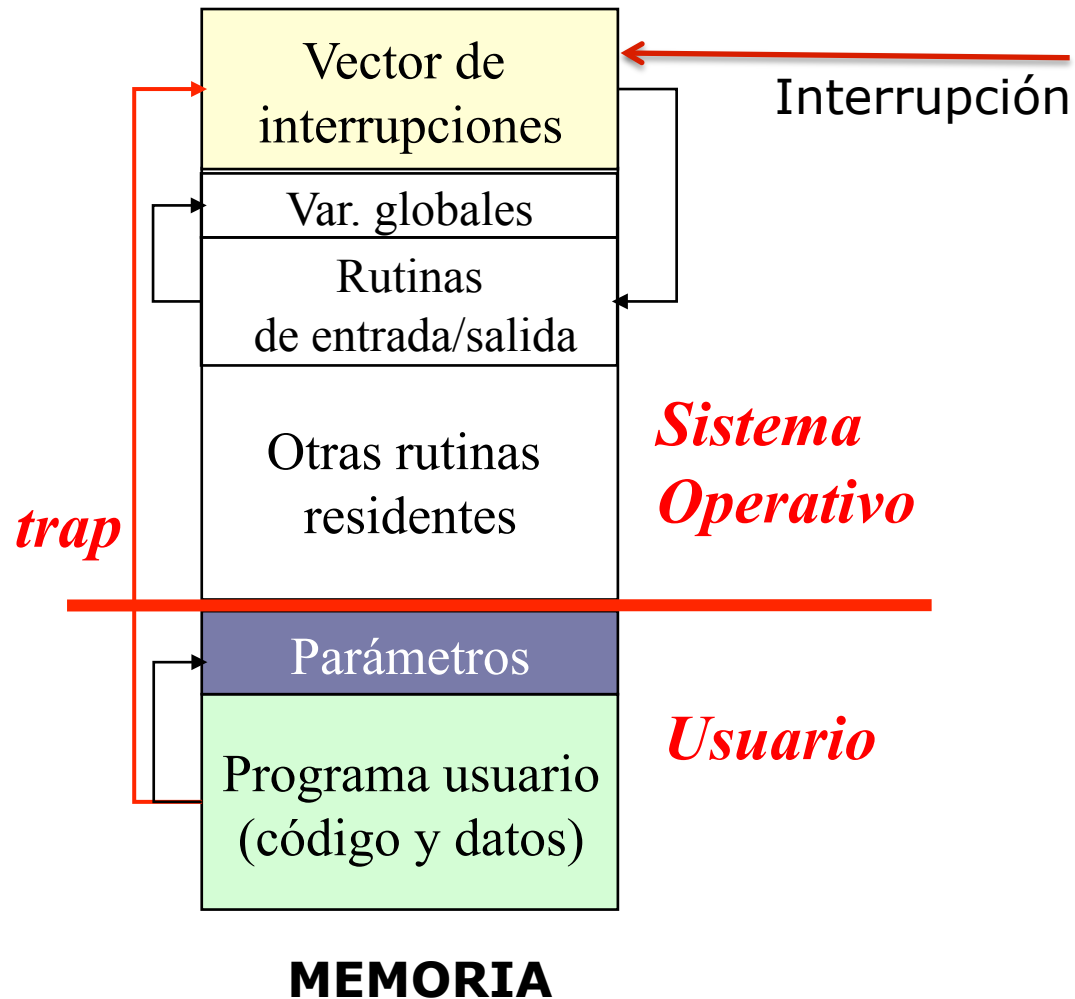
Rutinas residentes como embrión del sistema operativo



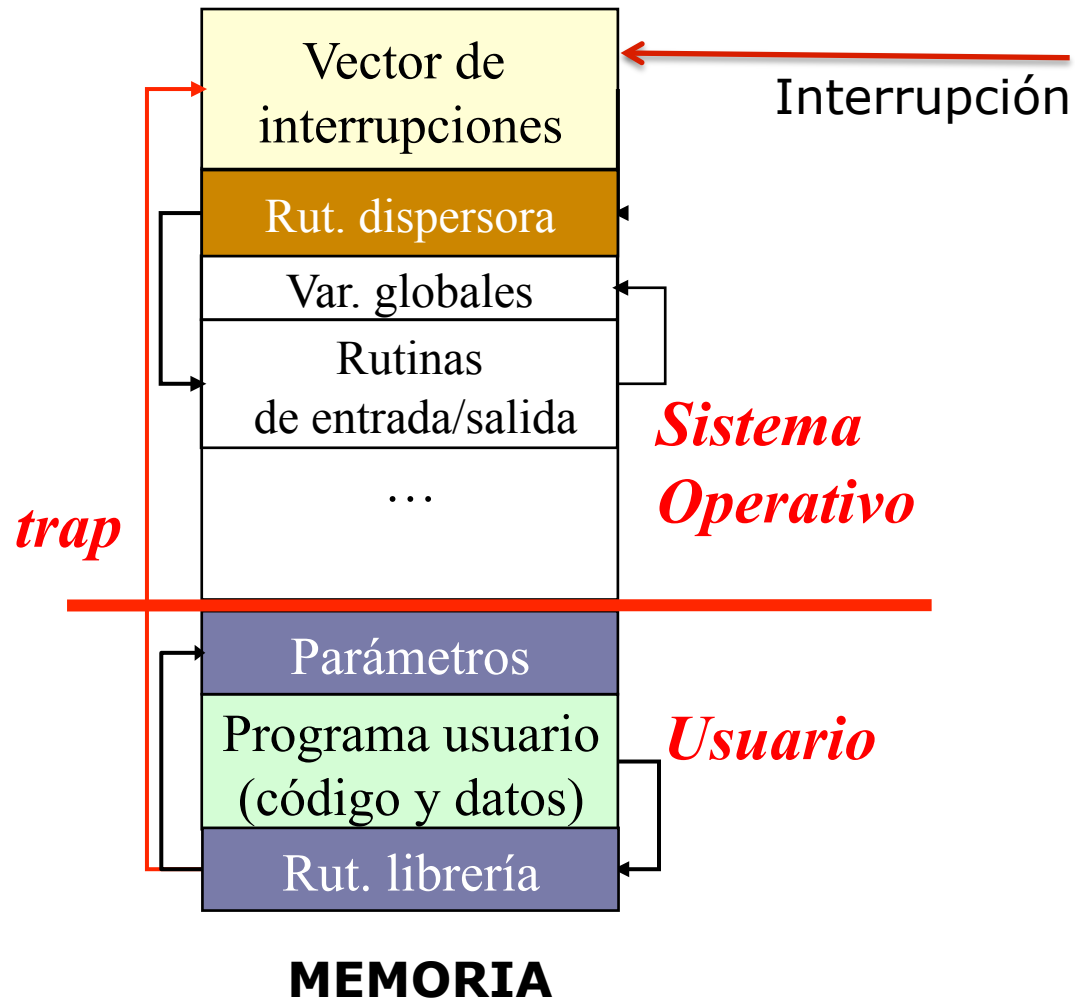
Rutinas residentes como embrión del sistema operativo



Un mecanismo de acceso unificado



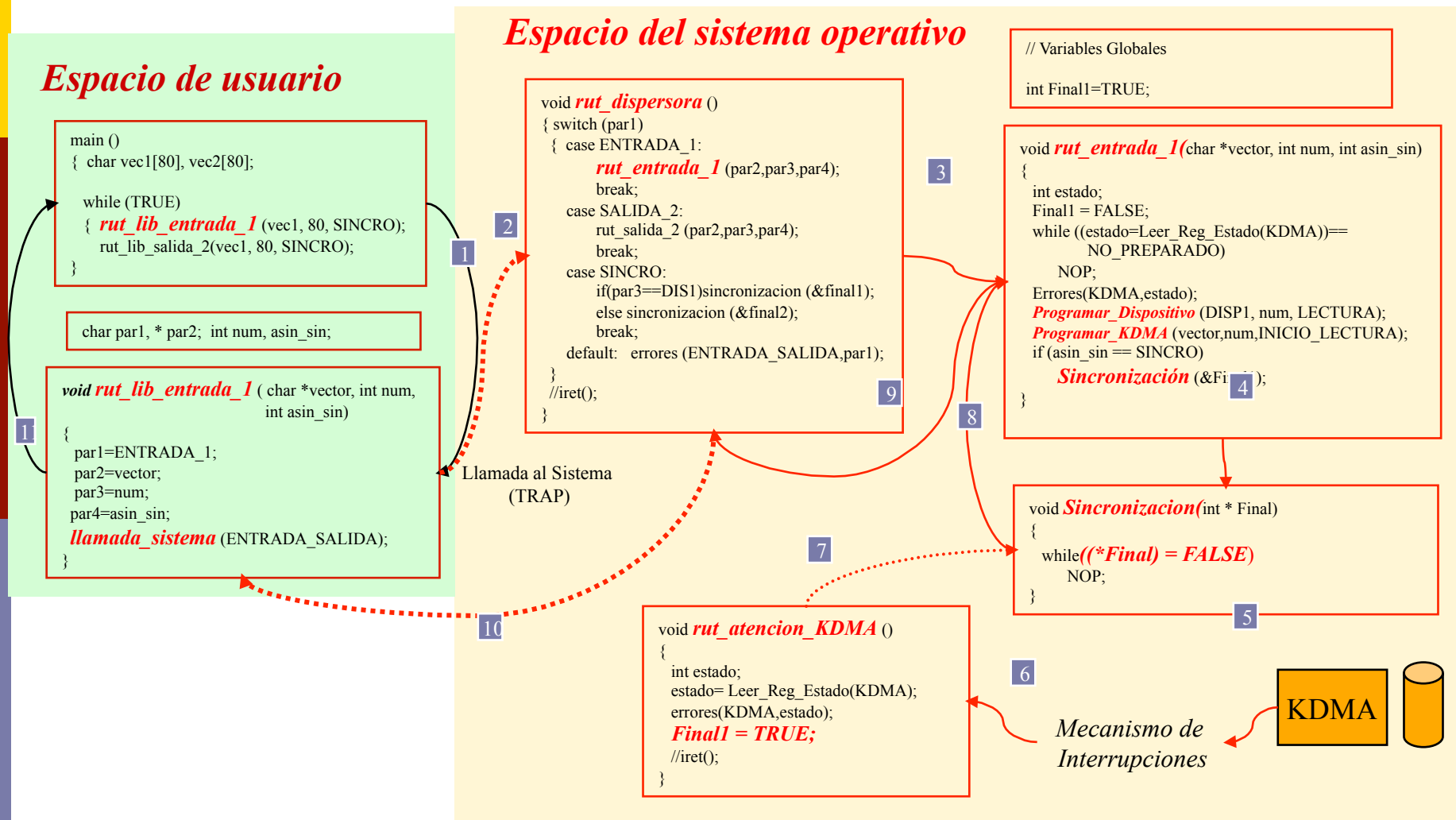
Un mecanismo de acceso unificado



Las llamadas al sistema como interfaz para las aplicaciones

- Las rutinas dispersoras permiten independizar las llamadas al sistema del tamaño del vector de interrupciones.
- Las rutinas de librería (en el espacio de usuario) permiten invocar las llamadas al sistema como funciones normales.
- Cada lenguaje puede añadir encima sus propias funciones de biblioteca para independizar el lenguaje del sistema operativo:
 - Windows:
 - `fopen()` ⇨ `CreateFile()`
 - `fclose()` ⇨ `CloseHandle()`
 - Unix:
 - `fopen()` ⇨ `open()`
 - `fclose()` ⇨ `close()`

Las llamadas al sistema como interfaz para las aplicaciones



Las llamadas al sistema como interfaz para las aplicaciones

Comparación UNIX - Windows

UNIX	Win32	Description
fork	CreateProcess	Create a new process
waitpid	WaitForSingleObject	Can wait for a process to exit
execve	(none)	CreateProcess = fork + execve
exit	ExitProcess	Terminate execution
open	CreateFile	Create a file or open an existing file
close	CloseHandle	Close a file
read	ReadFile	Read data from a file
write	WriteFile	Write data to a file
lseek	SetFilePointer	Move the file pointer
stat	GetFileAttributesEx	Get various file attributes
mkdir	CreateDirectory	Create a new directory
rmdir	RemoveDirectory	Remove an empty directory
link	(none)	Win32 does not support links
unlink	DeleteFile	Destroy an existing file
mount	(none)	Win32 does not support mount
umount	(none)	Win32 does not support mount
chdir	SetCurrentDirectory	Change the current working directory
chmod	(none)	Win32 does not support security (although NT does)
kill	(none)	Win32 does not support signals
time	GetLocalTime	Get the current time