Tema 8:

Introducción al multiprocesamiento empotrado

Programación de sistemas y dispositivos

José Manuel Mendías Cuadros
Dpto. Arquitectura de Computadores y Automática
Universidad Complutense de Madrid

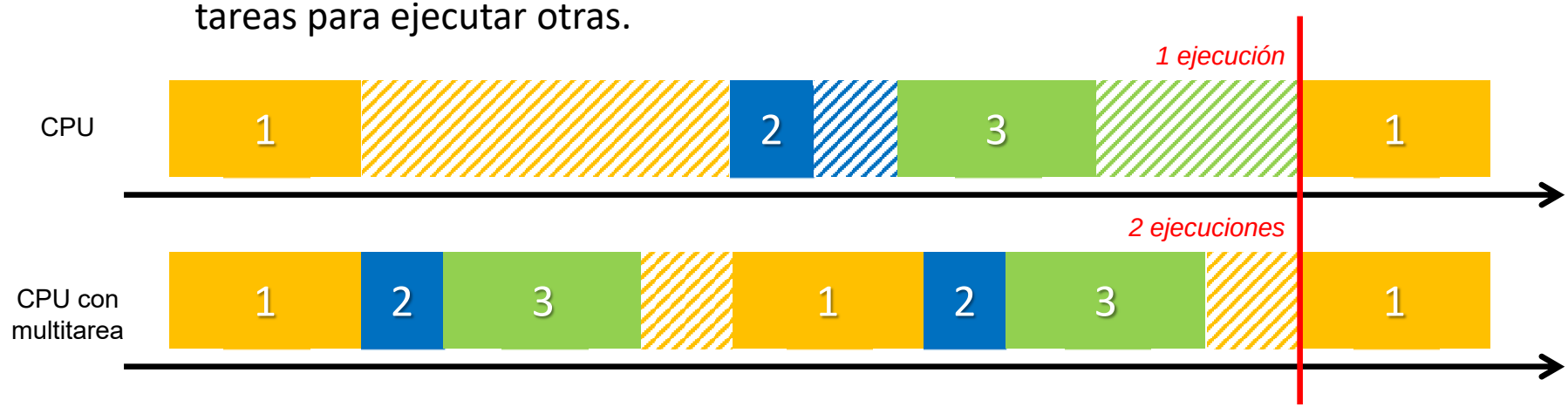
Contenidos

- ✓ Multitarea vs. multiprocesamiento.
- ✓ Multiprocesamiento: arquitecturas HW
- ✓ Multiprocesamiento: sincronización y comuniación
- ✓ Multiprocesamiento empotrado.
- ✓ Microcontroladores multicore.
- ✓ Planificación cooperativa en multiprocesadores.

J.M. Mendías
2022

Multitrea vs. multiprocesamiento

- Multitarea: ejecución concurrente de varias hebras en una única CPU
  - Aumenta el rendimiento del sistema aprovechando los tiempos de espera de unas tareas para ejecutar otras.

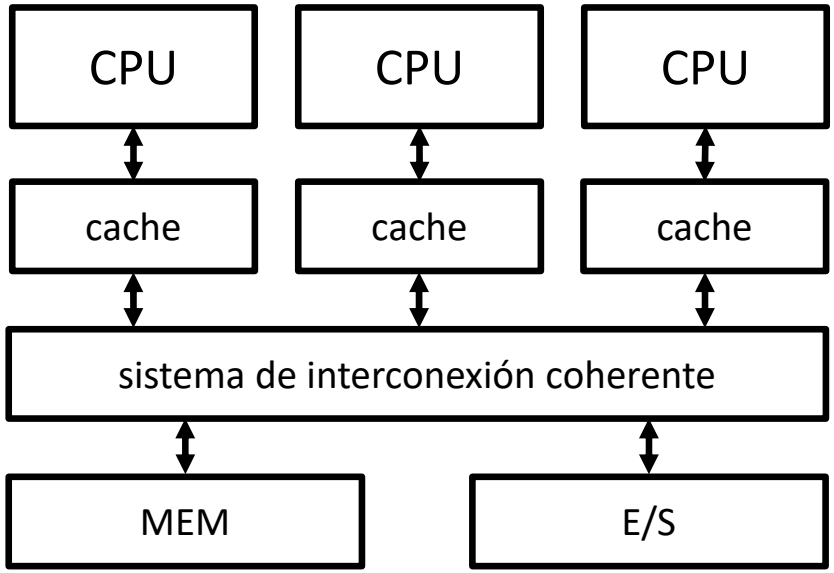


- Multiprocesamiento: ejecución paralela de varias hebras en varias CPU
  - Aumenta el rendimiento del sistema al ejecutar varias tareas simultáneamente
  - Cada CPU, a su vez, ser multitarea

J.M. Mendías
2022

Multiprocesamiento
arquitecturas hardware (i)

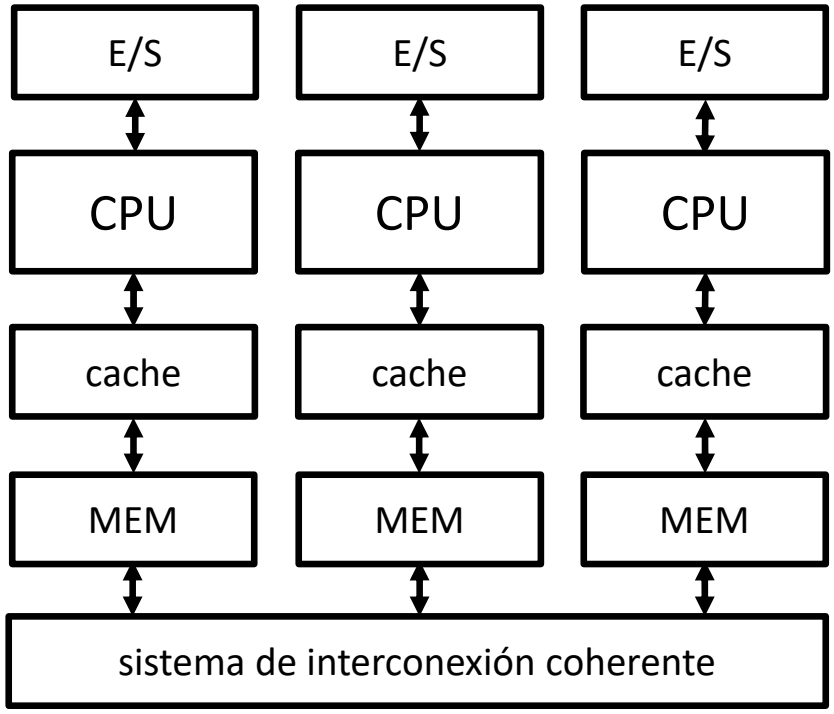
- Multiprocesamiento (de memoria compartida) simétrico (SMP)
  - Todos los cores son idénticos y tienen una cache local coherente.
  - Todos los cores se conectan a una única memoria compartida.
    - El tiempo de acceso a la memoria compartida es uniforme (UMA)
  - Todos los cores tienen acceso a todos los dispositivos de E/S.
  - De existir SO/RTOS, es único y se encarga de repartir recursos/cores entre hebras/aplicaciones de manera transparente y dinámica.



Multiprocesamiento

arquitecturas hardware (ii)

- Multiprocesamiento de memoria compartida distribuida (cc-NUMA)
  - Variante del multiprocesamiento simétrico
  - Cada core tiene a una memoria local que comparte con el resto de cores de manera transparente.
    - El tiempo de acceso a la memoria compartida es no uniforme (NUMA).

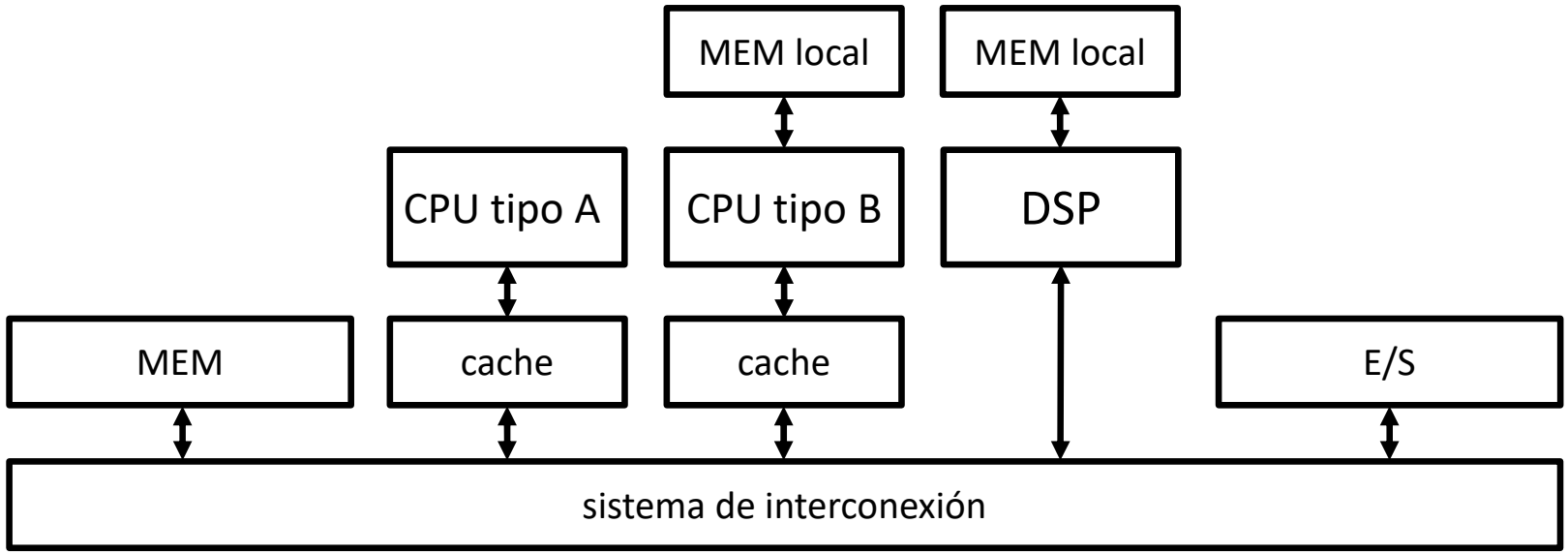


J.M. Mendias
2022

Multiprocesamiento

arquitecturas hardware (iii)

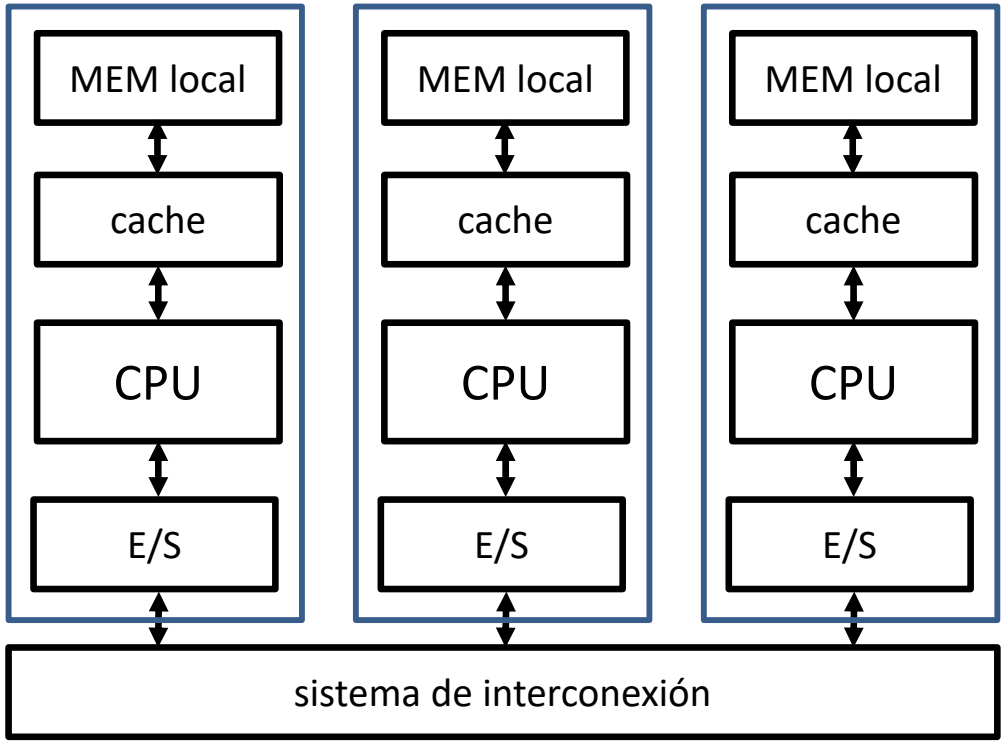
- Multiprocesamiento (de memoria compartida) asimétrico (SMP)
  - Los cores son diferentes y pueden tener una memoria y/o cache local no coherente
  - Todos los cores tienen acceso a una memoria compartida.
  - Cada core puede tener un acceso a una parte de los dispositivos de E/S.
  - De existir SO/RTOS, cada core puede corre el suyo que puede ser diferente al resto.
  - El reparto de recursos/cores entre hebras/aplicaciones es explícito.
- Es la arquitectura más común de las MCU multicore



Multiprocesamiento

arquitecturas hardware (iv)

- Multiprocesamiento de memoria distribuida
  - Están formados por nodos completos (CPU+MEM+E/S) eventualmente diferentes
  - Cada CPU (que puede ser multicore) se conecta a una memoria y cache local.
  - Los nodos se comunican mediante paso de mensajes a través de operaciones explícitas de E/S.
  - De existir SO/RTOS, cada nodo puede corre el suyo que puede ser diferente al resto.



Multiprocesamiento

sincronización y comunicación

- Comunicación por memoria compartida
  - Similar a la que se realiza en un sistema monoprocesador.
  - Las hebras comparten datos ubicándolos en la memoria compartida entre cores.
  - El acceso mutuamente exclusivo a los datos compartidos se garantiza usando semáforos hardware que también se usan para sincronizar hebras
  - Pueden construirse mecanismos de más abstractos: flags, mailbox, FIFOs...

- Comunicación por paso de mensajes
  - Las hebras se sincronizan y comparten datos enviándose mensajes cuyo formato (payload) es definido por SW.
    - Síncronos: los mensajes pasan directamente del emisor al receptor, por lo que las hebras se bloquean si el otro extremo no está listo para la comunicación.
    - Asíncronos: los mensajes se envían/reciben a/desde un buffer intermedio, por lo que las hebras no esperan al otro extremo para la comunicación.
  - En sistemas distribuidos, el paso de mensajes se efectúa sobre un interfaz de E/S especifico (I2C, RS-232, RS-485, CAN, Bluetooth, Zigbee, WiFi...)
  - En sistemas multicore, se implementa sobre memoria compartida

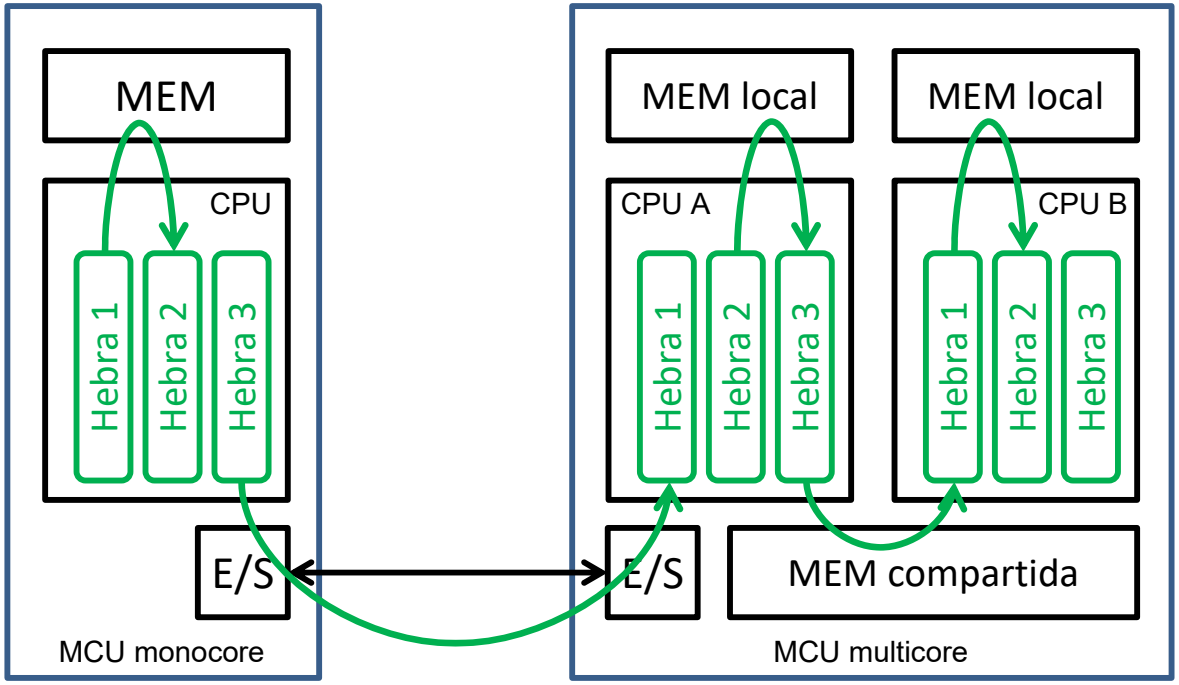
Multiprocesamiento empotrado

- Sistema empotrado monoprocesador:
  - Modelo Single App / Single Core: una única aplicación (típicamente multihebra) se ejecuta en un único procesador.
- Sistema empotrado multiprocesador fuertemente acoplado:
  - Modelo Single App / Multiple core: una única aplicación se particiona en distintas hebras que se ejecutan en paralelo sobre distintos procesadores.

- ¿Por qué plantear un sistema empotrado multiprocesador?
  - La aplicación no puede ejecutarse en tiempo real con una única MCU
    - Solución: usar una MCU multicore para ejecutar varias hebras en paralelo (task-paralelism)
  - La aplicación debe atender a más dispositivos de los posibles con una única MCU
    - Solución: adoptar una arquitectura local (mismo PCB/chasis) de múltiples MCU
  - La aplicación debe ser tolerante a fallos
    - Solución: adoptar una arquitectura redundante local de múltiples MCU
  - La aplicación debe interactuar son dispositivos físicamente lejanos
    - Solución: adoptar una arquitectura distribuida de múltiples MCU débilmente acoplados
    - Suelen seguir el modelo Multiple App / Multiple core: cliente-servidor, peer-to-peer...

Multiprocesamiento empotrado

- Tendremos distintos conjuntos de hebras que se ejecutan...
  - Concurrentemente (mismo core) y paralelamente (distinto core/dispositivo)
- ... y que se comunican y sincronizan:
  - Memoria local + semáforos SW (mismo core)
  - Memoria compartida + semáforos HW (distinto core / mismo dispositivo)
  - Paso de mensajes a través de E/S (distinto core / distinto dispositivo)



J.M. Mendías 2022

Microcontroladores multicore

regiones de memoria compartida (i)

- Cada core tiene un mapa de memoria propio en el que existen:
  - Regiones locales: a las que físicamente solo puede acceder el core.
  - Regiones globales compartidas: físicamente accesibles por todos los cores.
- Cualquier core puede leer/escribir en una región de memoria compartida
  - El arbitro de bus ordena los accesos simultáneos de los cores a la memoria.
  - En ellas se ubican las variables globales compartidas entre hebras de distintos cores.
  - Existe el problema de consistencia de variables globales en memoria compartida.
    - Hebras de distintos cores pueden tener copias locales de la variable inconsistentes.
    - Se resuelve utilizando semáforos hardware
  - Si los cores disponen de cache, aparecen problemas adicionales de consistencia.
    - La cache de cada core puede tener una copia distinta de la variable global.

```
volatile int8 *const sharedVar = (int32 *) 0x...;
```
puntero constante a una variable int32 volátil

```
void main( void )
{
  ...
  *sharedVar++;
  ...
}
```

```
ldr r0, =sharedVar
ldr r1, [r0]
add r1, r1, #1
str r1, [r0]
```

El orden de acceso de los cores a la variable afecta a su valor final (carrera)

CORE A / CORE B

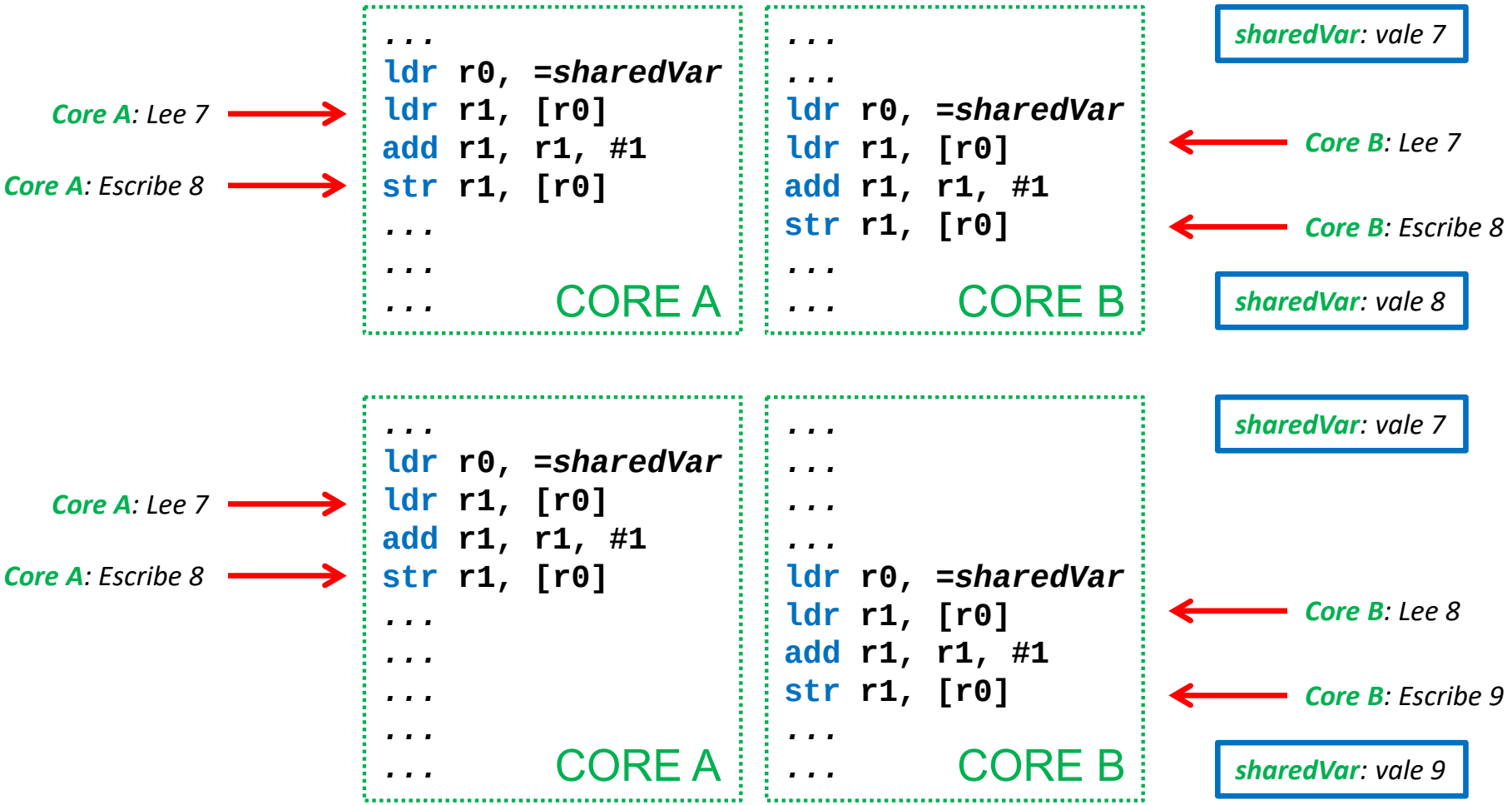
INCORRECTO

J.M. Mendías 2022

Microcontroladores multicore

regiones de memoria compartida (ii)

- Si el acceso por distintos cores a una variable global en memoria compartida no es mutuamente exclusivo, aparecen carreras.



Microcontroladores multicore

regiones de memoria compartida (iii)

- Multicores sin cache
  - No existen problemas de consistencia adicionales.
- Multicores con caches coherentes
  - Implementan un protocolo hardware que mantiene consistentes las copias de datos de la memoria compartida ubicadas en cada cache.
  - Cuando un core escribe un dato en memoria compartida, las caches actualizan automáticamente las eventuales copias que tengan del dato.
  - El programador se desentiende de este problema.
- Multicores con caches no coherentes
  - No tienen implementado protocolo de coherencia de cache alguno.
  - Cuando un core escribe un dato en memoria compartida, las caches pueden tener una copia de valores anteriores e inconsistentes del dato.
  - El programador debe poner soluciones SW:
    - Deshabilitar las caches de datos.
    - Marcar como no cacheables las regiones de memoria compartida.
    - Invalidar los bloques de cache antes de acceder a la región de memoria compartida.

J.M. Mendías 2022

Microcontroladores multicore

semáforos harware (i)

- Dispositivo hardware que ofrece un acceso mutuamente exclusivo a un registro compartido entre cores. Tiene 2 primitivas:
  - Lock, permite a cualquier core cerrar el semáforo de manera atómica.
    - Típicamente escribiendo en el semáforo el ID del core (e incluso el ID de la hebra).
    - Si varios cores tratan de cerrarlo simultáneamente, solo uno conseguirá escribir su ID.
  - Unlock, permite a cualquier core abrir el semáforo.
    - Típicamente borrando el ID del core almacenado en el semáforo.
    - Existe la posibilidad de que la apertura genere una interrupción en otro core.
- Según implementaciones, el cierre del semáforo puede ser:
  - 1 paso (test & set HW): la lectura del estado del semáforo lo cierra
    - Si estaba abierto, la hebra del core lector puede acceder a la sección crítica.
    - Si estaba cerrado, la hebra del core lector deberá repetir el proceso hasta conseguirlo.
  - 2 pasos (set & test SW): se cierra el semáforo y posteriormente se lee su estado
    - El core lector compara su ID con el ID almacenado en el semáforo.
    - Si es el mismo, el semáforo estaba abierto, el cierre ha sido efectivo y el core lector puede acceder a la sección crítica.
    - Si es distinto, el semáforo fue cerrado por otro core, el cierre no ha tenido efecto y el core lector deberá repetir el proceso.

Microcontroladores multicore

semáforos harware (ii)

- Cierre en un paso:

```
#define HWSEM (*(volatile uint8 *)0x...)  ·····  dirección del semáforo HW
                                                  (se trata como cualquier otro dispositivo)
void main( void )
{
  ...
  while( !HWSEM );  ·····  lee y cierra el semáforo HW, si estaba cerrado lo reintenta
  ...sección critica...
  HWSEM = 0;  ·····  abre el semáforo HW
  ...
}
```

CORE A / CORE B

- Cierre en dos pasos:

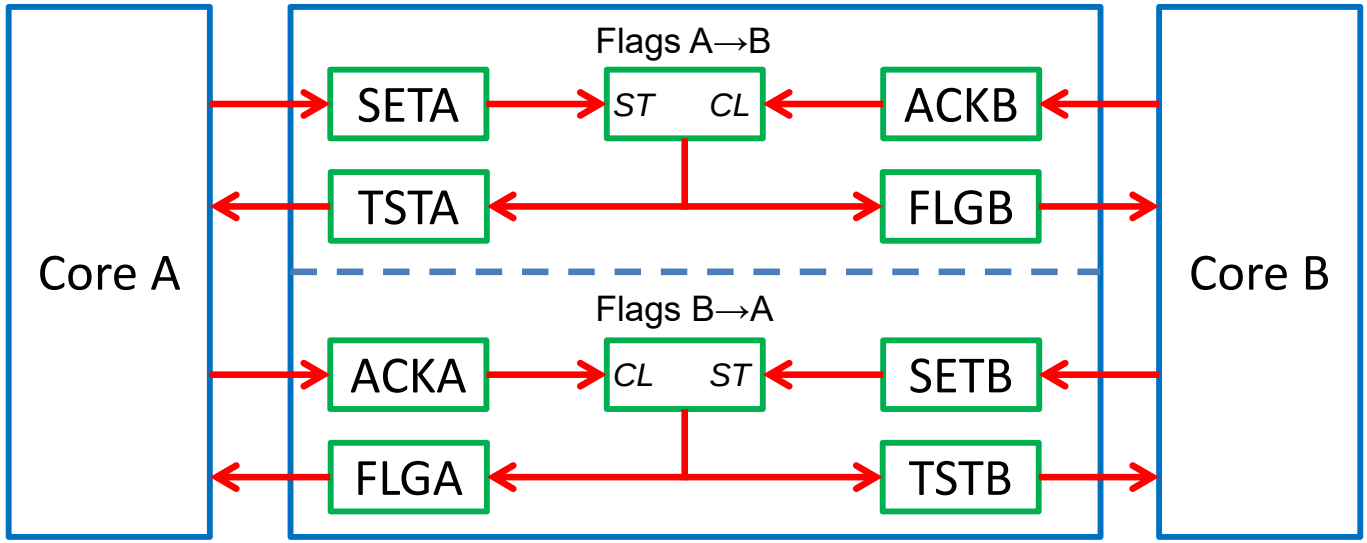
```
#define HWSEM (*(volatile uint8 *)0x...)  ·····  dirección del semáforo HW
                                                  (se trata como cualquier otro dispositivo)
void main( void )
{
  ...
  myID = core_getID();
  do { HWSEM = myID; }  ·····  cierra el semáforo HW
    while( HWSEM != myID );  ·····  si fue cerrado por otro core, lo reintenta
  ...sección critica...
  HWSEM = 0;  ·····  abre el semáforo HW
  ...
}
```

J.M. Mendías
2022

Microcontroladores multicore

flags hardware (i)

- Dispositivo HW dedicado a la señalización unidireccional entre cores
  - Registro cuyos bits puede activarse/desactivarse por un core diferente.
  - Los flags son accesible a través de distintos registros lógicos:
    - SETx (W): permite al core local activar un flag del core remoto.
    - ACKx (W): permite al core remoto desactivar uno de sus flags.
    - TSTx (R): permite al core local leer el estado de un flag del core remoto.
    - FLGx (R): permite al core remoto leer el estado de uno de sus flags.
    - CLRx (W): permite al core local desactivar un flag del core remoto.
  - Además, la activación de un flag puede generar una interrupción en el core remoto.



J.M. Mendías
2022

Microcontroladores multicore

flags hardware (ii)

- Los flag HW permiten sincronizar hebras en ejecución en distintos cores
  - Una hebra activa el flag.
  - La otra hebra espera la activación del flag, para desactivarlo.

```
#define FLAG_NUM (...)

void main( void )
{
  ...
  SETA &= (1 << FLAG_NUM);
  ...
}
```

Core A señaliza

CORE A

```
#define FLAG_NUM (...)

void main( void )
{
  ...
  SETB &= (1 << FLAG_NUM);
  ...
}
```

Core B señaliza

CORE B

```
#define FLAG_NUM (...)

void main( void )
{
  ...
  while( !( FLGB & (1<<FLAG_NUM) ) );
  ACKB &= (1<<FLAG_NUM);
  ...
}
```

Core B espera

CORE B

```
#define FLAG_NUM (...)

void main( void )
{
  ...
  while( !( FLGA & (1<<FLAG_NUM) ) );
  ACKA &= (1<<FLAG_NUM);
  ...
}
```

Core A espera

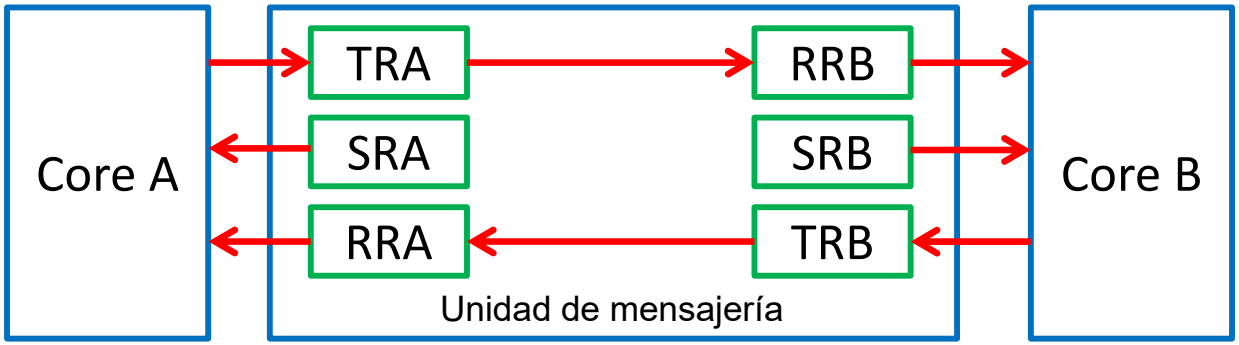
CORE A

J.M. Mendías 2022

Microcontroladores multicore

unidades de mensajería (i)

- Dispositivo HW dedicado al paso de mensajes entre cores
  - Esta formado por parejas de registros de transmisión/recepción unidireccionales
    - Pueden complementarse con parejas de flags de señalización en un registro de estado.
    - La escritura/lectura por parte de un core puede generar interrupciones en el otro
  - TRx (W): su escritura por el core local permite enviar un dato al core remoto.
    - De existir registros de estado SRx, esta escritura borra el flag TE (transmitter empty) del core local y activa el flag RF (receiver full) del core remoto.
    - Además, esta escritura puede generar una interrupción en el core remoto.
  - RRx (R): su lectura por el core local permite recibir el dato enviado por el core remoto.
    - De existir registros de estado SRx, esta lectura borra un flag RFx (receiver full) del core local y activa un flag TEx (transmitter empty) del core remoto.
    - Además, esta lectura puede generar una interrupción en el core remoto.



Microcontroladores multicore

unidades de mensajería (iii)

- El formato de los datos transferidos es definido por SW, así pueden:
  - Enviarse mensajes cortos (de anchura menor o igual al registro de mensajería)
  - Enviar la dirección y el tamaño de mensajes largos ubicados en memoria compartida (usando 2 registros de mensajería o enviando 2 mensajes consecutivos)
  - Sincronizar/señalizar hebras enviando mensajes vacíos.
  - Enviar comandos de un core a otro.
- Por ejemplo, es inmediato implementar un mailbox
  - Un core escribe el dato y el flag se activa automáticamente.
  - El otro core chequea (y eventualmente espera) la activación del flag. Cuando encuentra el flag activado, lee el dato y el flag se desactiva automáticamente

```
mail_t mail;
...
void main( void )
{
  ...
  mail = ...;
  TRA = mail;
  ...
}
```

CORE A

Core A envía

```
mail_t mail;
...
void main( void )
{
  ...
  while( !( SRB & (1<<FLAG_RF) ) );
  mail = RRB;
  ...
}
```

CORE B

Core B espera y lee

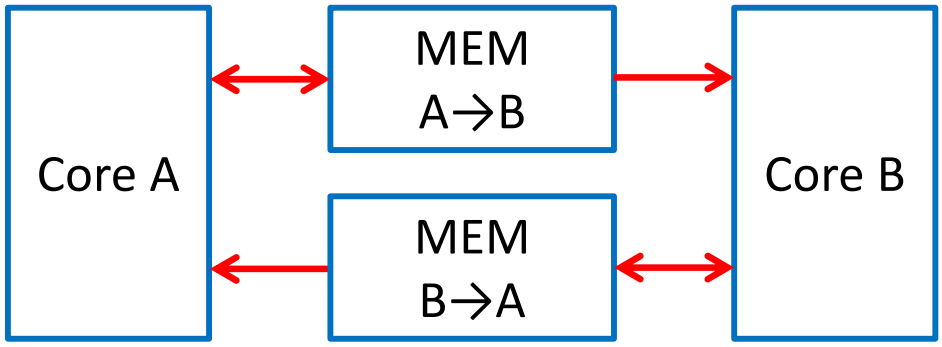
J.M. Mendías 2022

Microcontroladores multicore

memorias de mensajes

- Las memorias de mensajes son un caso de memorias compartidas
  - En las que cualquier core puede leer pero solo uno escribir.
  - No es necesario proteger su uso, pero sí sincronizar las hebras emisora y receptora

- Si el volumen de datos es alto, se suele usar DMA para mover datos:
  - El emisor inicia una transferencia por DMA desde su espacio local a un buffer ubicado en la memoria de mensajes.
  - Cuando la DMA finaliza, el emisor avisa al receptor de que hay datos disponibles.
  - El receptor inicia una transferencia por DMA desde el buffer ubicado en la memoria de mensajes y su espacio local.
  - Cuando la DMA finaliza, el receptor avisa al emisor para que envíe nuevos datos.



J.M. Mendías
2022

Microcontroladores multicore

interrupciones core-to-core

- En un sistema multicore cualquier core puede interrumpir a otro:
  - Ejecutando una instrucción de envío de evento (SEV) equivale a una SWI pero con efectos en el core remoto.
  - Escribiendo en ciertos bits del controlador de interrupciones dedicados a la generación de eventos remotos.
  - Abriendo un semáforo hardware.
  - Activando un flag hardware.
  - Enviando un mensaje a través de una unidad de mensajería.

- El mecanismo para que un core pueda ser interrumpido es el habitual:
  - Debe instalar una rutina de tratamiento asociada al evento.
  - Borrar las interrupciones pendientes.
  - Habilitar las interrupciones por la línea que corresponda.

Microcontroladores multicore

control de acceso a recursos

- En una MCU, el acceso a los recursos (memoria y dispositivos) puede estar supervisado por un dispositivo hardware dedicado.
  - MMU (Memory Management Units), RDC (Resource Domain Controllers)...
  - Permiten restringir con precisión el acceso de los cores a los recursos
  - Impiden accesos no permitidos generando excepciones cuando los detectan

- Permiten definir regiones de memoria y atribuir cada una con:
  - Tipo: normal o de dispositivo (en ella se mapean registros de dispositivos).
  - Clase: compartida o local a cierto core.
  - Modo Acceso: lectura/escritura, solo lectura, solo de acceso a datos (no ejecución)
  - Nivel de privilegio/seguridad requerido: el core debe estar en un cierto estado para acceder al recurso
  - Semáforo HW asociado: para que el acceso al recurso en caso de ser compartido requiera su cierre previo.
    - Separación de datos, instrucciones y pilas

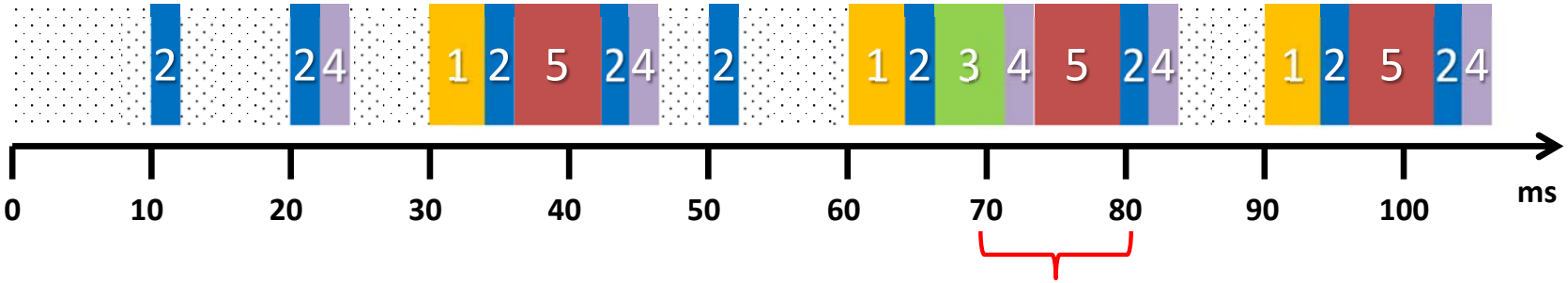
Planificador cooperativo

problemas de underrun

- Sistema monoprocesador:

Resolución del tick: **10 ms**
La prioridad de las tareas es: **T1 > T2 > T3 > T4 > T5**

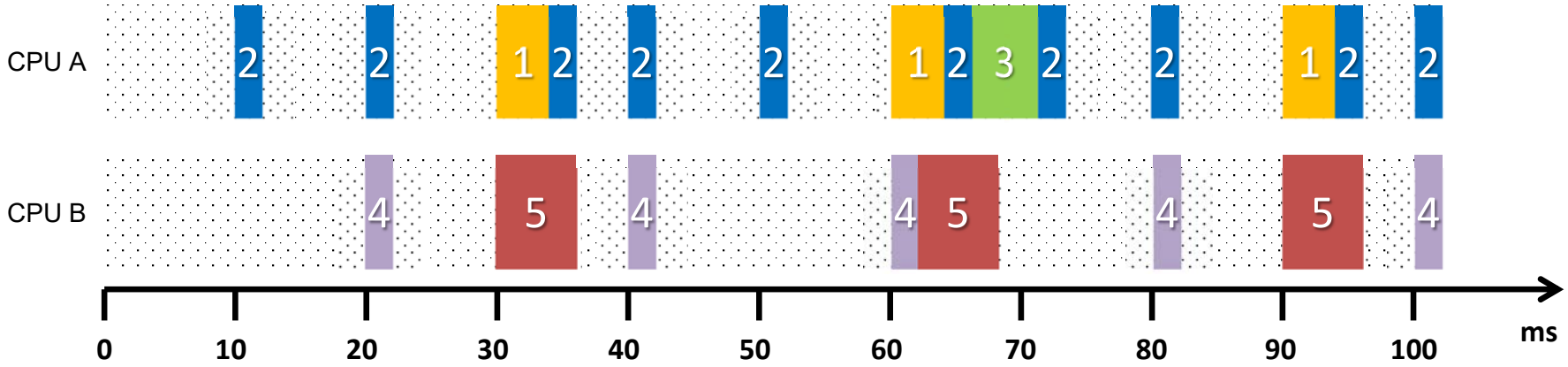
tarea 1: $t_1 = 30$ ms, $c_2 = 4$ ms
tarea 2: $t_2 = 10$ ms, $c_2 = 2$ ms
tarea 3: $t_3 = 60$ ms, $c_3 = 5$ ms
tarea 4: $t_3 = 20$ ms, $c_3 = 2$ ms
tarea 5: $t_3 = 30$ ms, $c_3 = 6$ ms



la instancia de tarea 2 correspondiente a este slot no ha podido ejecutarse

- Sistema multiprocesador:

CPU A: **T1 > T2 > T3** CPU B: **T4 > T5**



Planificador cooperativo

extensión multiprocesador

- El esquema más simple es el de reloj compartido
  - Las tareas de la aplicación se reparten estáticamente entre los procesadores.
    - En todo momento sólo existe una única tarea en ejecución en cada procesador
  - Cada procesador ejecuta localmente un planificador cooperativo
    - Encargado de planificar y despachar las tareas que le han sido asignadas
  - Un procesador actuará como maestro y el resto como esclavos:
    - El maestro es el encargado de sincronizar todos los planificadores compartiendo su tick.
    - El mecanismo de distribución de tick dependerá de la arquitectura hardware.

```
void master_scheduler( void )
{
   ... borra interrupción pendiente por timer...

  master_sendTick();
  scheduler();
};
```

El **planificador maestro** es la RTI por fin de cuenta del temporizador

planificador cooperativo convencional

```
void slave_scheduler( void )
{
  ... borra interrupción pendiente por recepción del tick ...

  scheduler();
};
```

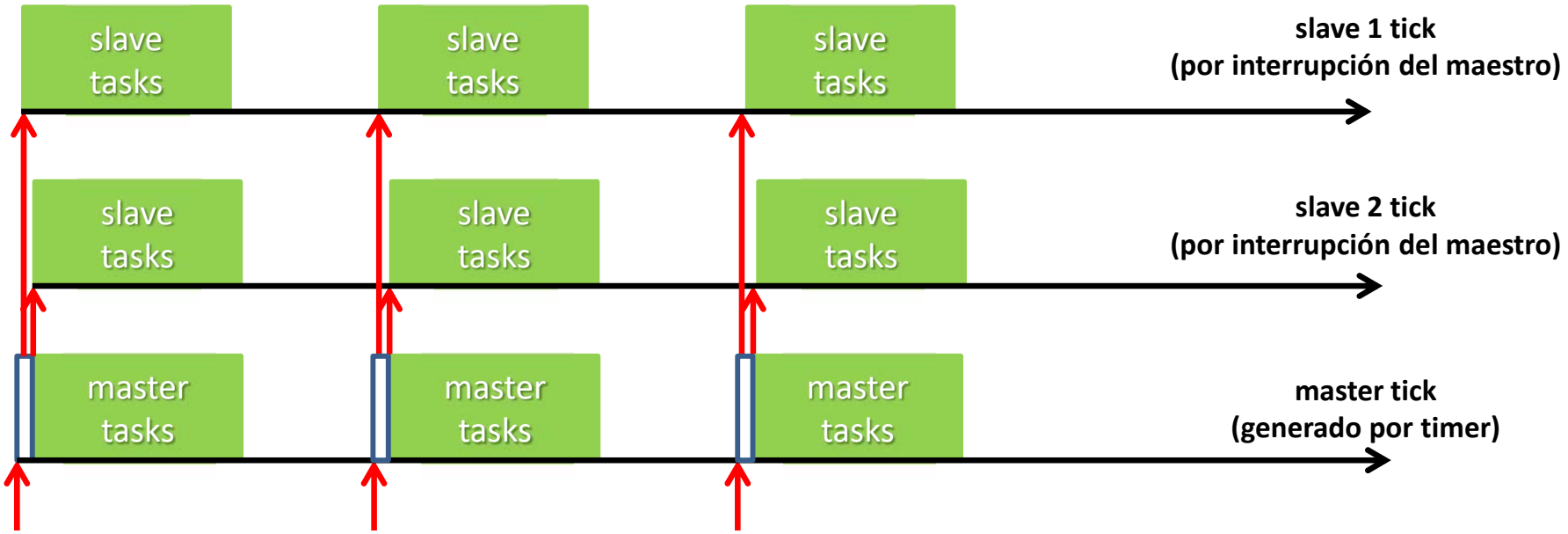
El **planificador esclavo** es la RTI recepción del tick

planificador cooperativo convencional

Planificador cooperativo

MCU multicore

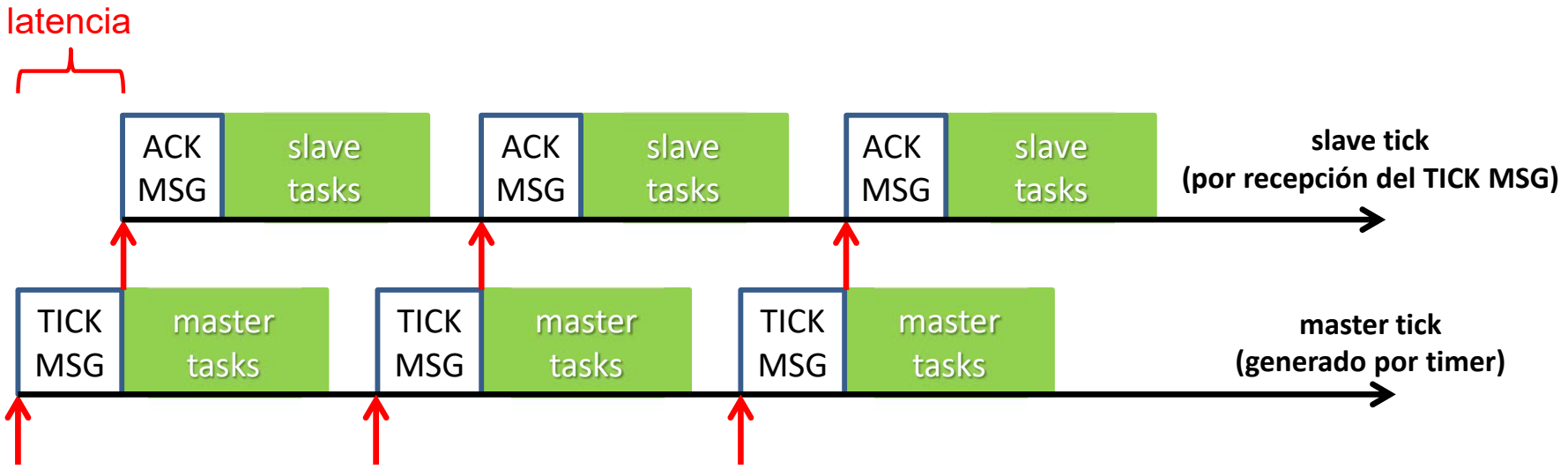
- Aplica un esquema de interrupción periódica directa
  - El maestro es interrumpido periódicamente por un timer para:
    - Generar explícitamente una interrupción en cada esclavo que sirve para distribuir el reloj.
    - Ejecutar aquellas tareas que tiene asignadas y que haya activado su planificador local.
  - Todos los esclavos son interrumpidos periódicamente por el maestro para:
    - Ejecutar aquellas tareas que tiene asignadas y que haya activado su planificador local.
    - Los esclavos no realizan ACK a menos que se quiera que el maestro detecte bloqueos.
  - La comunicación entre tareas intra/inter core se hace a través de memoria.



Planificador cooperativo

doble MCU distribuidas

- Aplica un esquema de envío periódico de mensajes
  - El maestro envía periódicamente un mensaje TICK al esclavo
    - Lo hace disparado por la interrupción periódica de un timer.
    - El mensaje TICK sirve para distribuir el reloj y opcionalmente enviar datos al esclavo.
    - La latencia del tick dependerá de la longitud del mensaje y la velocidad de transmisión.
    - Si el mensaje es muy largo se reparte entre mensajes consecutivos.
  - El esclavo envía periódicamente un mensaje ACK en respuesta al master
    - Lo hace disparado por la interrupción de recepción del mensaje TICK
    - El mensaje ACK sirve como mecanismo de confirmación de ausencia de errores en el nodo o en la red, así como para enviar datos al maestro.

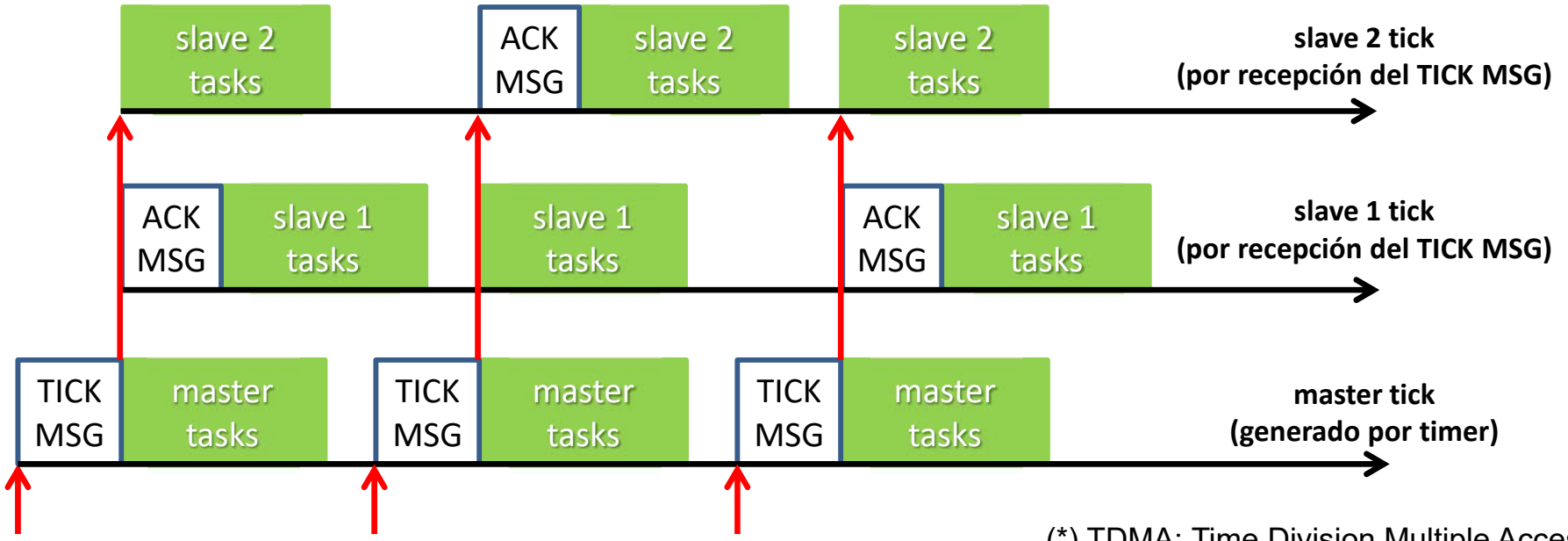


J.M. Mendías 2022

Planificador cooperativo

múltiples MCU distribuidas

- Aplica un esquema TDMA* de envío periódico de mensajes
  - El maestro difunde periódicamente un mensaje TICK a todos los esclavos
    - Lo hace disparado por la interrupción periódica de un timer .
    - El mensaje TICK incluye el ID de un esclavo y opcionalmente datos para dicho esclavo.
    - Rotatoriamente el master manda los ID de todos los esclavos de la red.
  - El esclavo aludido envía un mensaje ACK al master
    - Lo hace disparado por la interrupción de recepción del mensaje TICK.
    - El mensaje ACK incluye el ID de un esclavo y opcionalmente datos para el maestro.



(*) TDMA: Time Division Multiple Access

Planificador cooperativo

múltiples MCU distribuidas

- En entornos distribuidos deben usarse enlaces físicos de baja latencia:
  - RS-232: transmisión full-duplex (2 hilos)
    - conexión punto a punto: máximo 2 nodos
    - mensajes de 1 byte
    - longitud/velocidad enlace: hasta 30 m (115 Kb/s)
  - RS-485: transmisión diferencial full-duplex (2 pares de hilos)
    - conexión multipunto, hasta 32 nodos
    - mensajes de 1 byte
    - longitud/velocidad del enlace: 15 m (10 Mb/s), hasta 1 Km (90 Kb/s)
  - CAN: transmisión diferencial half-duplex (1 par de hilos)
    - conexión multipunto: hasta 32 nodos
    - mensajes de hasta 8 bytes
    - longitud/velocidad: 40 m (1 Mb/s) - 10 Km (5 Kb/s)
  - IIC: transmisión síncrona half-duplex (2 hilos)
    - conexión multipunto: nominalmente hasta 127 nodos, en la practica no muchos
    - mensajes de longitud no definida
    - longitud/velocidad: 1 m (400 Kb/s)

Planificador cooperativo

ejemplo: dos MCU conectadas por UART1 (RS-232)

- Solo requiere modificar ligeramente el planificador
  - Para tener 2 versiones del mismo: una para el maestro y otra para el esclavo
  - El resto es idéntico a un sistema monoprocesador.

```
void master_scheduler( void )
{
  static boolean init = TRUE;

  I_ISPC = BIT_TIMER0;          ····· borra interrupción pendiente del timer

  if( init )                    ┐
    init = FALSE;               │
  else                          │ en la primera llamada al planificador maestro, no se lee
    ackMsg = URXH1;             ┘ el mensaje ACK porque el esclavo aún no lo ha mandado ····· Lee el mensaje ACK (enviado por el esclavo en anterior time-slot)
  UTXH1 = tickMsg;              ····· Envía el mensaje TICK (para el actual time-slot del esclavo)
  scheduler();                  ····· planificador cooperativo convencional
};
```

```
void slave_scheduler( void )
{
  I_ISPC = BIT_URXD1;           ····· borra interrupción pendiente por recepción por UART1

  tickData = URXH1;             ····· Lee el mensaje TICK (enviado por el maestro en el actual time-slot)
  UTXH1 = ackMsg;               ····· Envía el mensaje ACK (para el siguiente time-slot del maestro)
  scheduler();                  ····· planificador cooperativo convencional
};
```

Planificador cooperativo

programa maestro

```
...
void masterScheduler( void ) __attributte__ ((interrupt ("IRQ")));
void timer0_open_tick( void (*isr)(), unit16 tps );

char ackMsg, tickMsg;          ..... Mensajes intercambiados master-slave

void main( void )
{
  sys_init();

  scheduler_init();
  create_task( masterTask1, 3 );
  ...

  masterTask1_init();
  ...

  sw_delay_ms( INIT_DELAY );   ..... Espera que el esclavo arranque
  ch = URXH1;                  ..... Elimina eventuales bytes recibidos pendientes de leer
  UTXH1 = 0x0;
  while( !(UTRSTAT1 & (1<<0)) );   } Rendevouz inicial maestro-esclavo
  ch = URXH1;
  timer0_open_tick( master_scheduler, TICKSxSEC );   ..... El planificador del maestro se instala como RTI del timer0
  while( 1 )
  {
    sleep();
    dispacher();
  }
}
```

Planificador cooperativo

programa esclavo

```
...
void slaveScheduler( void ) __attributte__ ((interrupt ("IRQ")));
void uart1_openRX( void (*isr)() );

char ackMsg, tickMsg;

void main( void )
{
  sys_init();

  scheduler_init();
  create_task( slaveTask1, 2 );
  ...

  slaveTask1_init();
  ...

  ch = URXH1;
  while( !(UTRSTAT1 & (1<<0)) );
  ch = URXH1;
  UTXH1 = 0x0;
  uart1_openRX( slaveScheduler );
  while( 1 )
  {
    sleep();
    dispacher();
  }
}
```

Mensajes intercambiados master-slave

Elimina eventuales bytes recibidos pendientes de leer

Rendevouz inicial maestro-esclavo

El planificador del maestro se instala como RTI por recepción de datos desde la UART1

Acerca de *Creative Commons*

- Licencia CC (Creative Commons)
  - Ofrece algunos derechos a terceras personas bajo ciertas condiciones. Este documento tiene establecidas las siguientes:

Reconocimiento (*Attribution*):
En cualquier explotación de la obra autorizada por la licencia hará falta reconocer la autoría.

No comercial (*Non commercial*):
La explotación de la obra queda limitada a usos no comerciales.

Compartir igual (*Share alike*):
La explotación autorizada incluye la creación de obras derivadas siempre que mantengan la misma licencia al ser divulgadas.

Más información: https://creativecommons.org/licenses/by-nc-sa/4.0/