Fundamentos para la programación de un OS

Mg. Ing. Gonzalo E. Sanchez
Esp. Ing. Hanes N. Sciarrone
MSE - 2023

Implementación de Sistemas Operativos (I)

Fundamentos de OS

- ARM Assembler
- Cambio de contexto
- Service Calls (PendSV)
- ARM Architecture Procedure Call Standard

ARM Assembler

ARM ASM

- La codificación de un kernel para OS es dependiente del código ensamblador del procesador utilizado.
- Es imposible hacer cambios de contexto sin acceder a los registros del CPU.
- Para los alcances de esta materia, utilizaremos solamente 20 líneas (max).
- Se deja al lector ahondar en el set de instrucciones. Solo se explicarán las instrucciones necesarias.

ARM ASM

- Instrucción **PUSH:** Efectúa un push dentro del stack full-descending.
  - Utilización: **PUSH**{*cond*} *reglist*
  - *cond* es un código opcional de condición (sufijo de condición).
  - *reglist* es la lista de registros a ser insertados en la pila. No puede estar vacío, y debe estar encerrada entre llaves.
- Ejemplos de uso:
  - `PUSH {R4}`
  - `PUSH {R4-R7}`
  - `PUSH {R4,R6,R8}`

ARM ASM

- Instrucción **POP:** Efectúa un pop dentro del stack full-descending.
  - Utilización: **POP**{*cond*} *reglist*
  - *cond* es un código opcional de condición (sufijo de condición).
  - *reglist* es la lista de registros a ser insertados en la pila. No puede estar vacío, y debe estar encerrada entre llaves.
- Ejemplos de uso:
  - `POP {R4}`
  - `POP {R4-R7}`
  - `POP {R4,R6,R8}`

ARM ASM

- Tanto PUSH como POP actualizan automáticamente el Stack Pointer (MSP o PSP según corresponda).
- Para hacer push de registros y luego recuperarlos la sintaxis es:

```
PUSH {R4-R8}
POP {R4-R8}
```

- No es necesario invertir el orden de los registros.
- POP comienza a llenar los registros desde el último en *reglist* hacia el primero.

ARM ASM

- Los sufijos condicionales en instrucciones alteran el comportamiento de las mismas.
- Instrucciones que pueden ser condicionales requieren una instrucción IT previa (mandatorio).
- Si un condicional no se cumple, la instrucción:
  - No se ejecuta.
  - No escribe nada en el registro destino.
  - No afecta ningún flag de estado.
  - No genera ninguna excepción.

ARM ASM

- Una instrucción con un sufijo condicional solo se ejecuta si el flag de condición correspondiente está activo en APSR.

Suffix	Flags	Meaning
EQ	Z = 1	Equal
NE	Z = 0	Not equal
CS or HS	C = 1	Higher or same, unsigned
CC or LO	C = 0	Lower, unsigned
MI	N = 1	Negative
PL	N = 0	Positive or zero

ARM ASM

- Instrucción **MSR:** Mueve el contenido de un registro de propósito general al registro especial indicado.
  - Utilización: **MSR**{*cond*} *spec_reg, Rn*
  - *cond* es un código opcional de condición (sufijo de condición).
  - *spec_reg* es uno de los siguientes: APSR, MSP, PSP, PRIMASK, BASEPRI, BASEPRI_MAX, FAULTMASK, or CONTROL.
  - *Rn* es el registro de propósito general fuente del dato
- Esta instrucción depende del nivel de privilegio al momento de ejecución.

ARM ASM

- En modo no privilegiado, sólo puede utilizarse con APSR
- Ejemplos de uso:
  - `MSR MSP, R0`
  - `MSR APSR, R1`

ARM ASM

- Instrucción **MRS:** Mueve el contenido de un registro especial al registro de propósito general indicado.
  - Utilización: **MRS**{*cond*} *Rd, spec_reg*
  - *cond* es un código opcional de condición (sufijo de condición).
  - *spec_reg* es uno de los siguientes: APSR, MSP, PSP, PRIMASK, BASEPRI, BASEPRI_MAX, FAULTMASK, or CONTROL.
  - *Rd* es el registro de propósito general destino del dato
- Ejemplo de uso:
  - `MRS R0, MSP`

ARM ASM

- La utilización de MRS y MSR es importante a la hora de guardar el stack pointer entre cambios de contexto.
- Ejemplo de utilización para obtener el MSP y luego cargarlo desde R0:

```
MRS R0, MSP
//otro código
MSR MSP, R0
```

ARM ASM

- Instrucción **IT:** indica el inicio de un bloque IF-THEN
  - Utilización: **IT**{*x*{*y*{*z*}}} *cond*
  - *cond* es un código de condición.
  - *x, y, z* indican el flujo de ejecución para la segunda, tercer y cuarta instrucción luego de la instrucción IT.

ARM ASM

- Ejemplo de uso:

```
CMP R0, #9 ;
//Convert R0 hex value (0 to 15) into ASCII
//('0'-'9', 'A'-'F')
ITE GT ; Next 2 instructions are conditional
ADDGT R1, R0, #55 ; Convert 0xA -> 'A'
ADDLE R1, R0, #48 ; Convert 0x0 -> '0'
```

ARM ASM

- Instrucción **TST:** Testeo de bits
  - Utilización: **TST**{*cond*} *Rn, Operand2*
  - *cond* es un código opcional de condición (sufijo de condición).
  - *Rn* es un registro de propósito general conteniendo el operando 1.
  - *Operand2* es un operador flexible (registro o un literal).
- Efectúa un AND del operando 2 sobre el registro, pero el resultado se descarta y actualiza los flags.

Modelo del programador

- Ejemplo de uso:

  TST LR, 0x10 //Testea bit 4 del contenido actual de LR

- Si el bit[4] == 1, se actualiza el flag Z = 1.
- Las condiciones con sufijo EQ serán verdaderas.

ARM ASM

- Instrucciónes **B, BL** y **BX:** Instrucciones de branching.
  - Utilización:
    - **B**{*cond*} *label*
    - **BL**{*cond*} *label*
    - **BX**{*cond*} *Rm*
  - *cond* es un código opcional de condición (sufijo de condición).
  - *B* es un branch inmediato.
  - *BL* es un branch con link (actualiza LR)
  - BX es un branch indirecto (utiliza un registro).
  - *Rm* es el registro que contiene la dirección donde hacer el branch

ARM ASM

- Al utilizar BX, es mandatorio que el bit[0] = 1 para cortex M4.
- La dirección de memoria no debe ser impar. Una vez cargada en Rm se modifica seteando el bit[0] = 1.
- Recordar que el valor del bit[0] del PC siempre se carga en xPSR[24] (Thumb bit).
- Cuando se carga esta dirección en el PC, el bit[0] se utiliza para indicar que la instrucción en esa dirección es THUMB.

Cambio de contexto

Cambio de contexto

- El cambio de contexto es la base de un OS.
- Comprender la mecánica de cambio de contexto es lo que ayuda al desarrollo de nuevas funcionalidades.
- Hacer debug puede llegar a tornarse complicado.

Cambio de contexto

- Ejemplo: OS corriendo 2 tareas. Cambio de contexto inicial y entre tareas.
- **PASO 1:** Determinación de stack para cada tarea_

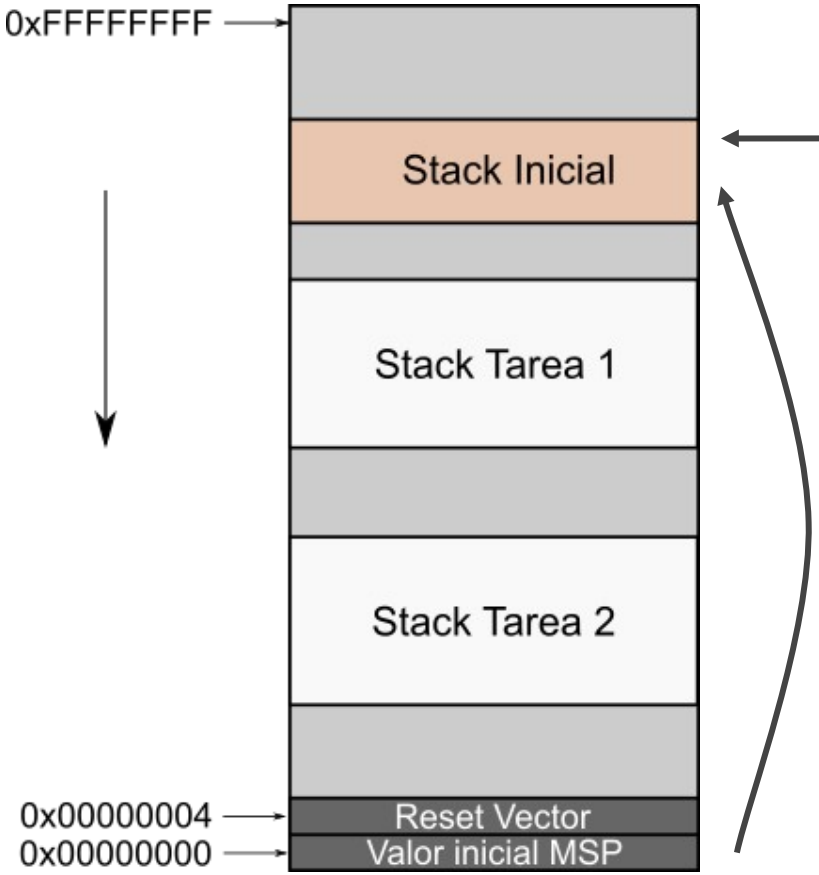
```
#define STACK_SIZE 256

uint32_t stack1[STACK_SIZE/4];
uint32_t stack2[STACK_SIZE/4];

void tarea1(void);

void tarea2(void);
```

Cambio de contexto



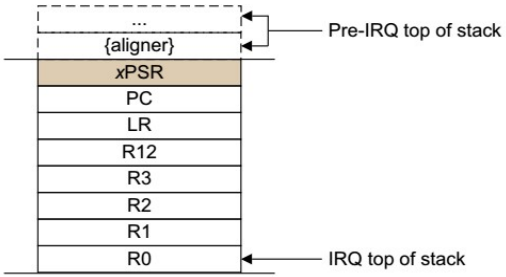
Dirección de memoria cargada en MSP

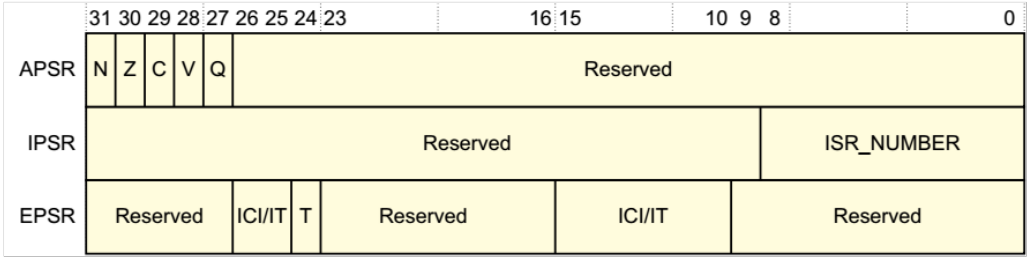
- El stack inicial no tiene un tamaño definido.
- Solamente tiene punto de entrada.
- Punto de entrada almacenado en la dirección 0x00000000 (FLASH).

Cambio de contexto

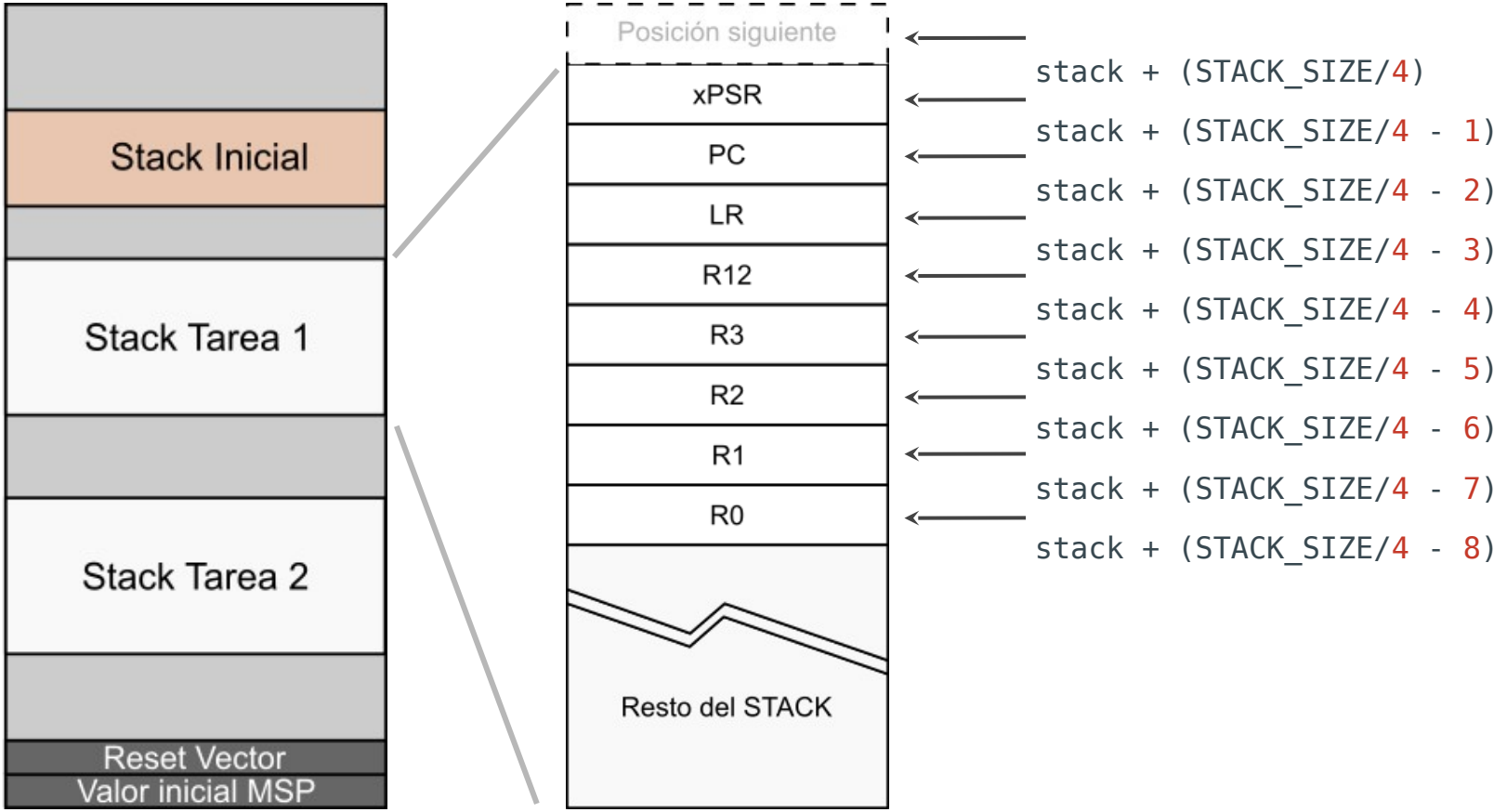
- **PASO 2:** Creación de Stack frame para cada tarea.

```
stack[STACK_SIZE/4-1] = 1 << 24;                    /* xPSR.T = 1
*/
stack[STACK_SIZE/4-2] = (uint32_t)entry_point;      /* PC */
stack[STACK_SIZE/4-3] = (uint32_t)task_return_hook; /* LR */
stack[STACK_SIZE/4-8] = (uint32_t)arg;              /* R0 */
stack[STACK_SIZE/4-9] = 0xFFFFFFF9;                 /* LR IRQ */
```





Cambio de contexto



Cambio de contexto

- Hay registros que no están siendo inicializados.
- R12 es el llamado registro “scratch”.
- Es utilizado por el compilador de C para almacenar valores intermedios en distintas operaciones.
- R3 - R1 son de propósito general.
- El stack frame generado es para el primer ingreso.
- Los valores de R12, R3-R1 no son utilizados en el primer ingreso.

Cambio de contexto

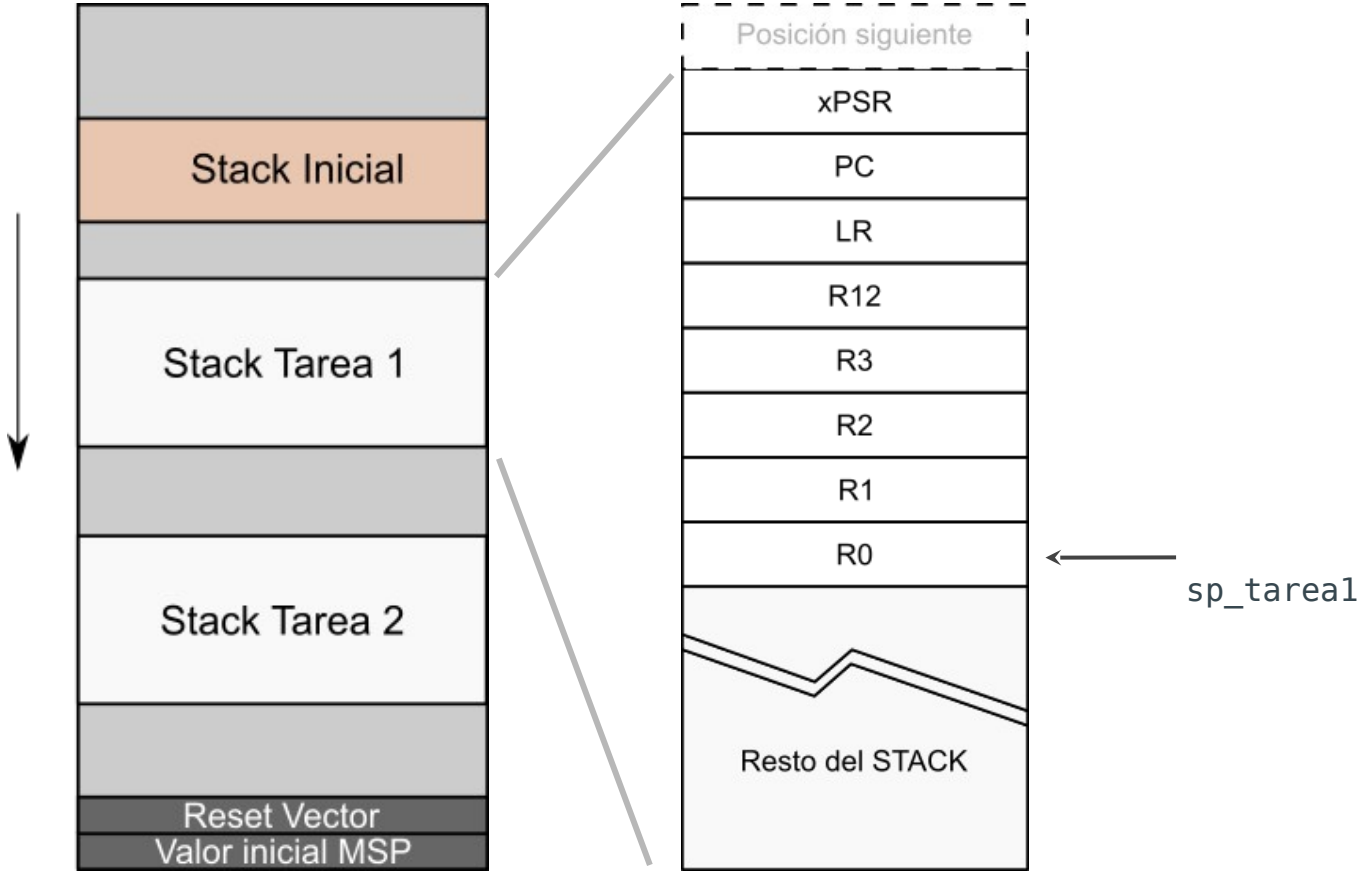
- **PASO 3:** Determinación del valor del Stack Pointer para cada tarea.

```
#define AUTO_STACKING_SIZE    8

uint32_t sp_tarea1, sp_tarea2;

sp_tarea1 = (uint32_t)(stack1+STACK_SIZE/4 - AUTO_STACKING_SIZE);
sp_tarea2 = (uint32_t)(stack2+STACK_SIZE/4 - AUTO_STACKING_SIZE);
```

Cambio de contexto



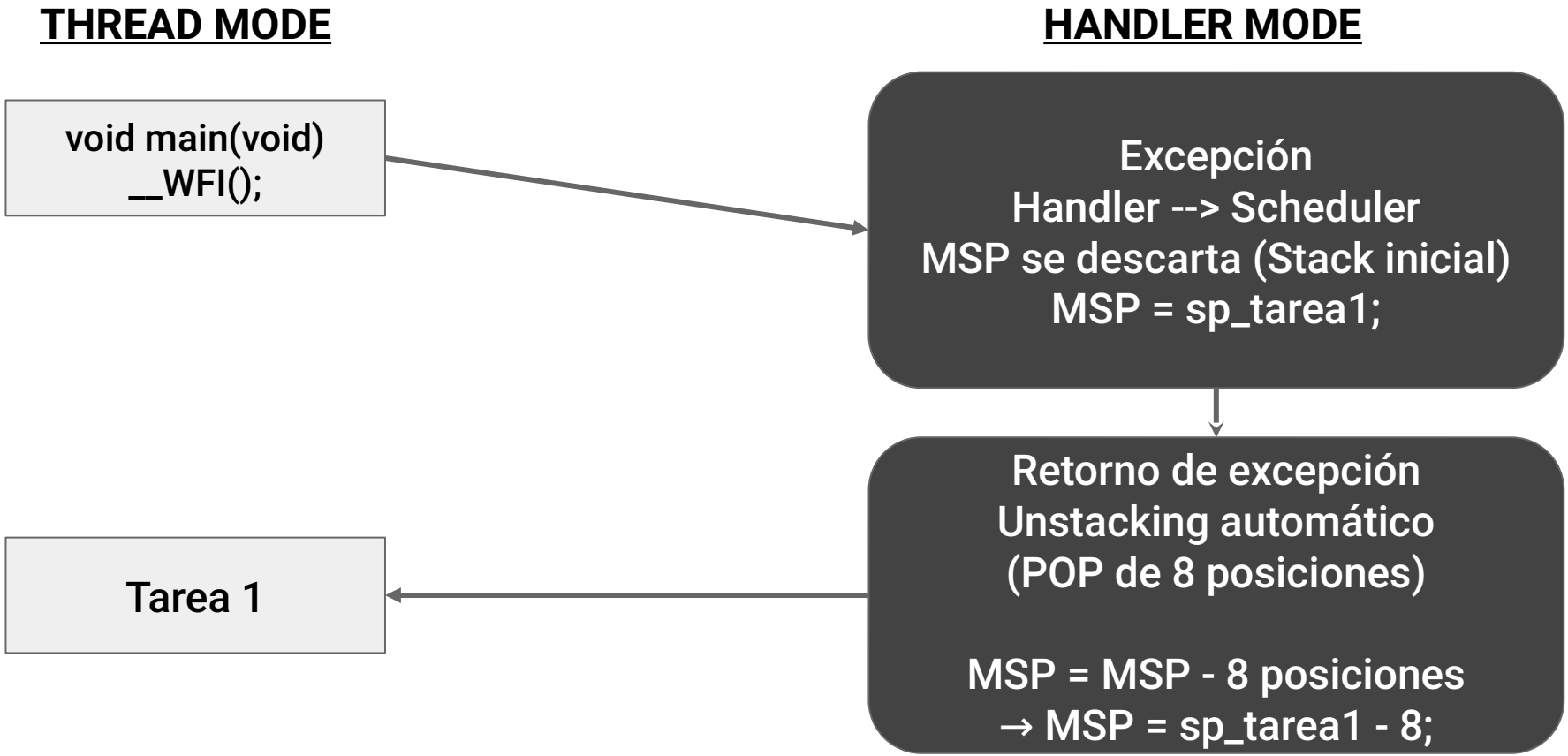
Cambio de contexto

- Es muy importante no confundir el stack pointer definido con el registro MSP.
- El stack pointer definido (sp_tarea1, sp_tarea2, ….) es una variable en RAM.
- Análoga a la definición de un vector que se accede con aritmética de punteros, avanzando o retrocediendo.
- Guarda el valor del registro MSP cuando la tarea es expropiada.
- Se carga ese valor guardado en MSP cuando se retorna a la tarea anteriormente expropiada.

Cambio de contexto

- **PASO 4:** Primer ingreso del programa al Scheduler.
- NOTA: Este ejemplo de 2 tareas es sobre un OS expropiativo con política de scheduling Round-Robin.
- La secuencia es la siguiente:
  - El programa principal efectúa configuraciones y queda a la espera del scheduler (instrucción WFI)
  - Ocurre una excepción, cuyo handler contiene el scheduler.
  - Scheduler hace cambio de contexto a *tarea1 (stack1)*.
  - Se retorna de la excepción.

Cambio de contexto



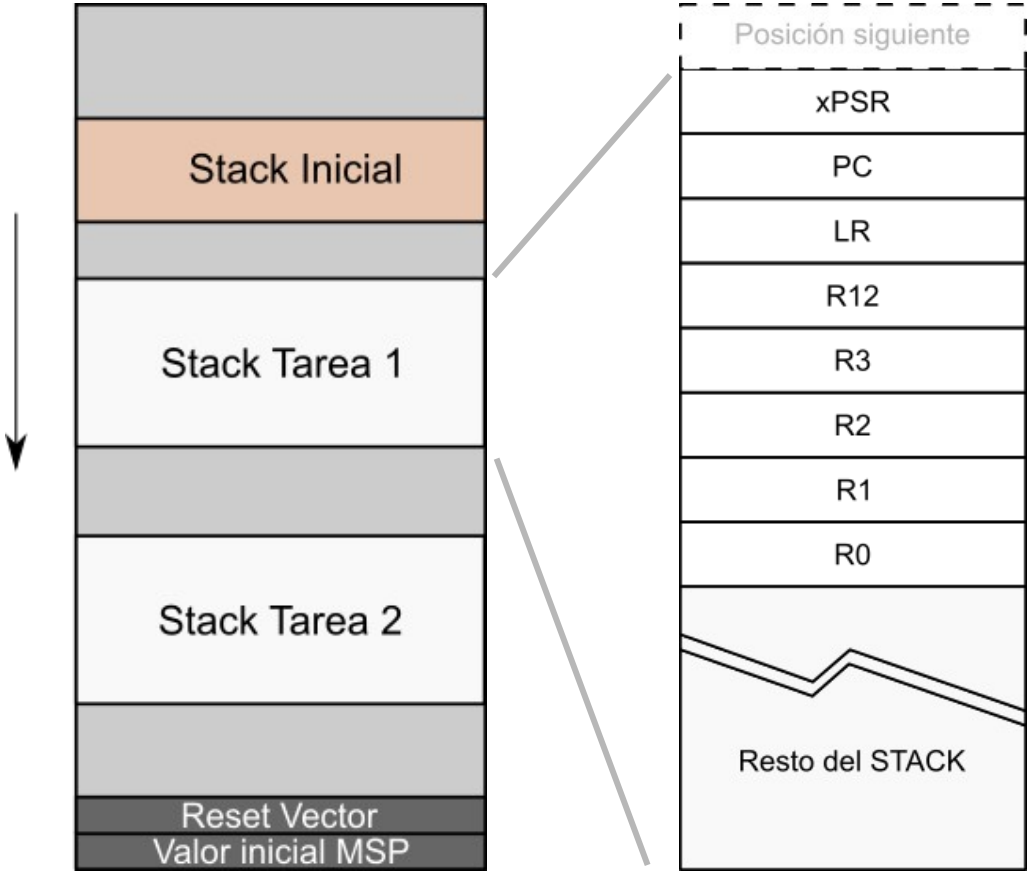
Cambio de contexto

- **IMPORTANTE:** En este punto MSP contiene la dirección del elemento *posterior* al último accesible del array stack1.

```
MSP = stack1 + STACK_SIZE/4;
```

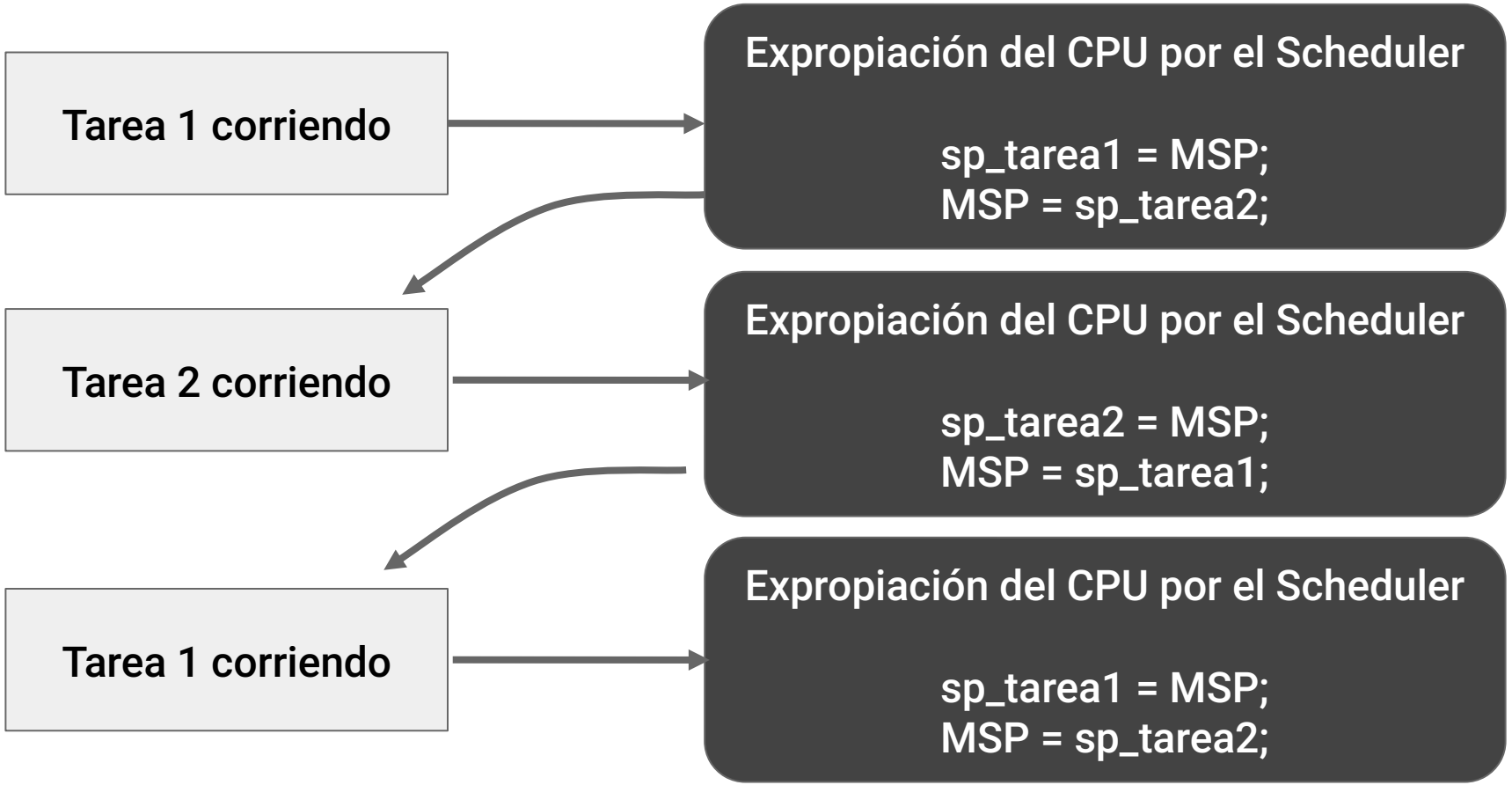
- Esto es consecuencia del mecanismo de PUSH, que equivale a decrementar MSP en una posición y luego almacenar el valor.
- Si se hiciese un PUSH en este momento, se almacenaría en la dirección `stack1 + STACK_SIZE/4 - 1`

Cambio de contexto



- **IMPORTANTE:** Valores al primer ingreso de ejecución de tarea1 (modo trhead).

Cambio de contexto

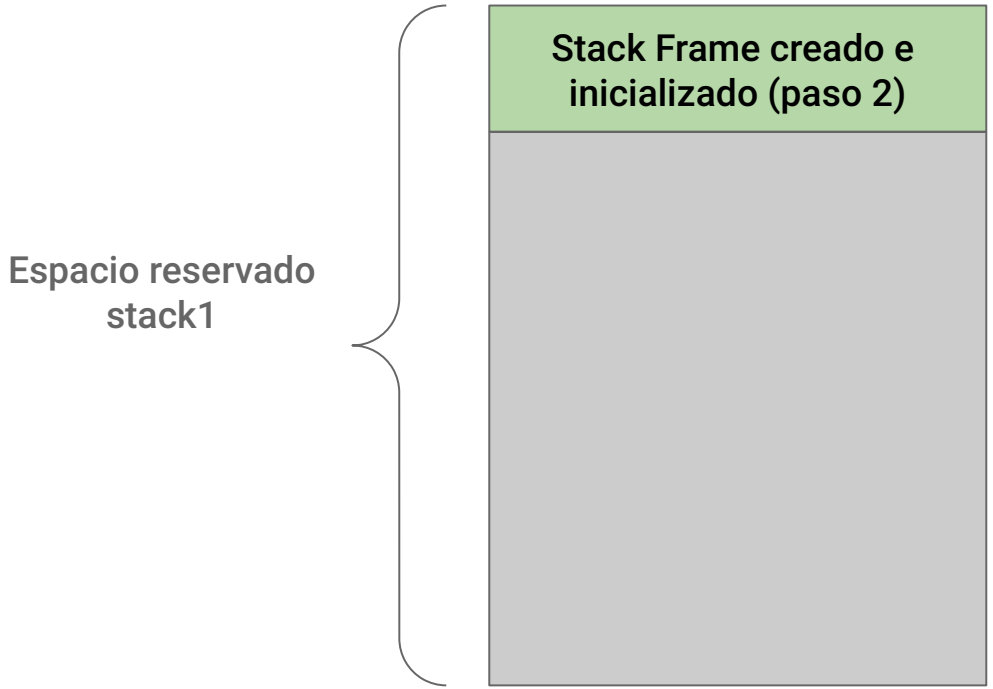


Cambio de contexto

- La variable que guarda el estado actual del stack pointer es de suma importancia.
- Dentro de la tarea al llamar a distintas funciones el MSP se modifica.
- **Ejemplo**: dentro de la *tarea1* se llama a la función *foo()*.
- Dentro de *foo()* se llama a la función *bar()*.
- Existe un solo registro LR (un solo nivel de profundidad de anidamiento).
- La dirección de retorno para volver a tarea1 debe apilarse.

Cambio de contexto

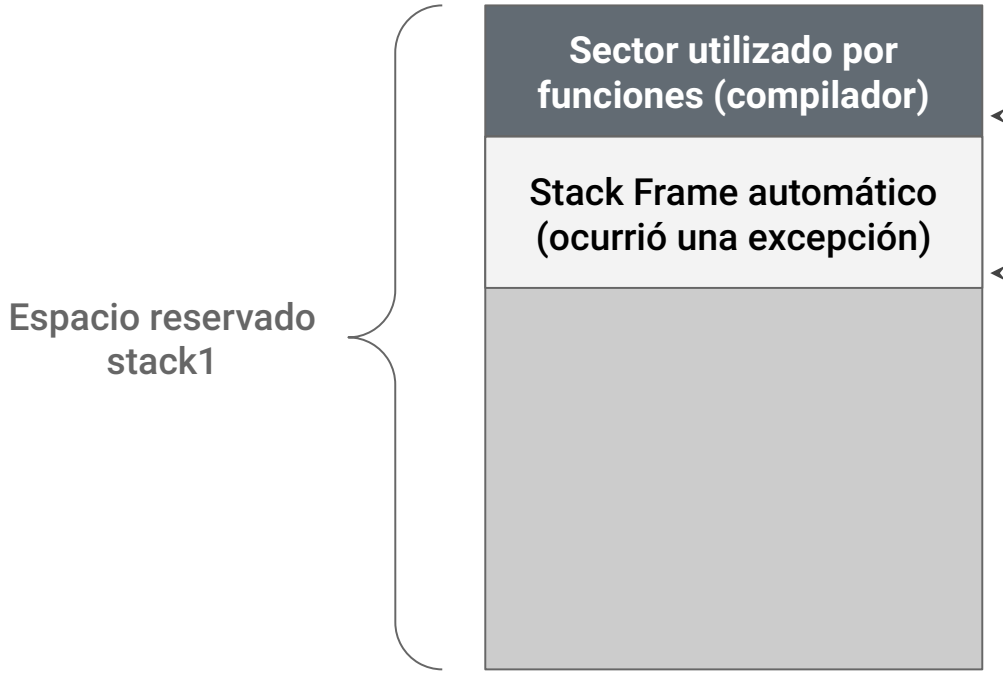
Estado del stack reservado antes de la primera llamada a la tarea1.



- Recordar: Al ingresar a la tarea por primera vez el Stack Frame inicializado se consume (pop).
- Al ocurrir una excepción se vuelve a escribir automáticamente.

Cambio de contexto

Estado del stack reservado durante la primera llamada a la tarea1 y llamadas consecutivas.




```
sp_tarea1 = MSP;
```

- Recordar: una vez que se vuelva a esta tarea, se hace POP del Stack Frame automáticamente al retornar de excepción.

Cambio de contexto

HANDS ON

Service Calls

(PendSV)

Service Calls

- El core Cortex-M está pensado para soportar implementaciones de OS.
- Para ello, existen distintas herramientas a disposición.
- Una muy importante son las llamadas Service Calls.
- Nos enfocaremos solamente en la excepción Pended Service Call (PendSV)

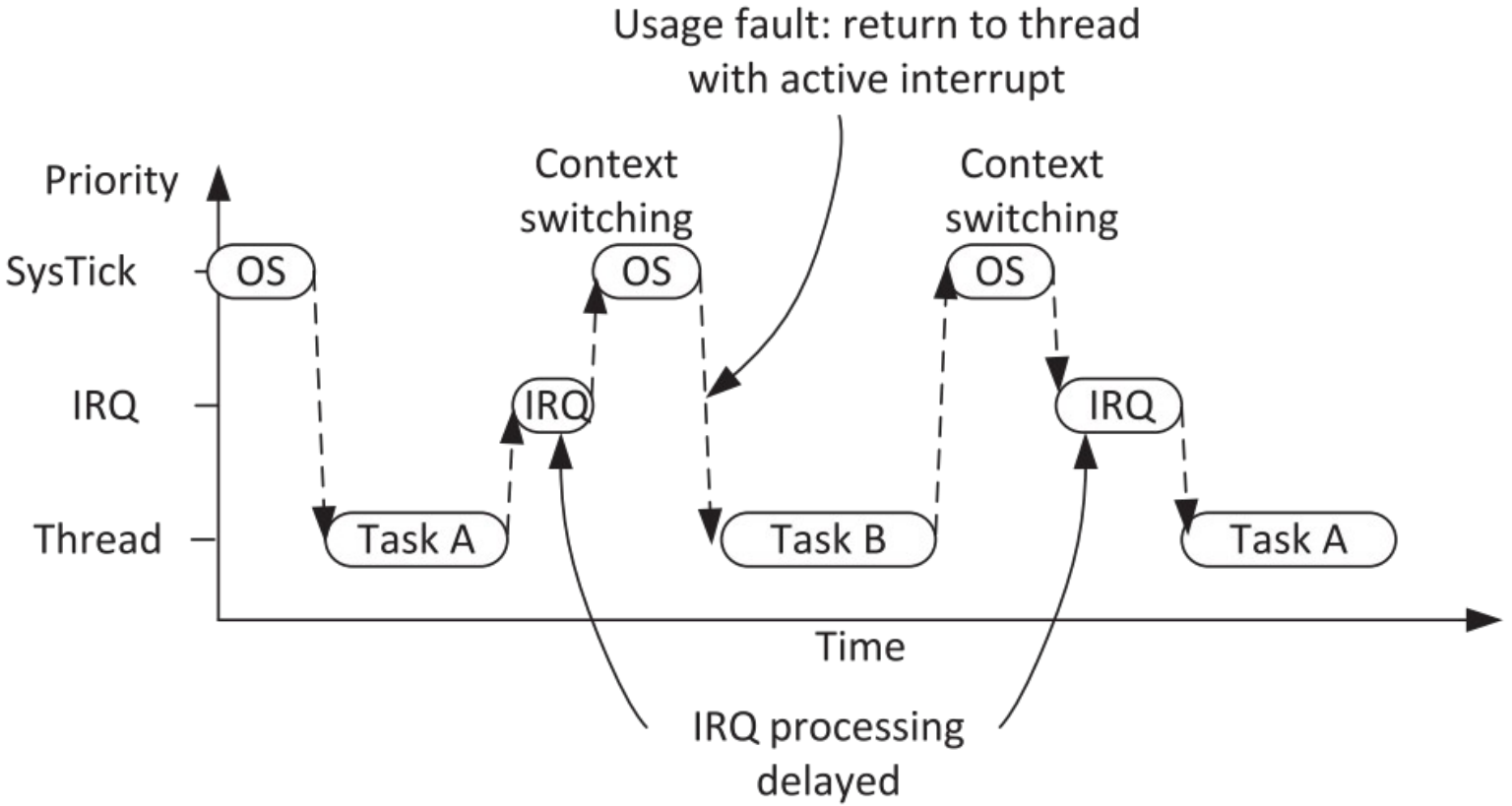
PendSV

- PendSV es una de las excepciones importantes que soportan la implementación de un OS en Cortex-M
- Se lanza seteando el bit correspondiente de excepción pendiente en el registro ICSR.
- Prioridad programable.
- Setear la prioridad más baja posible se hace que sea ejecutada luego de todas las demás excepciones.
- No es precisa, puede ser seteada como pendiente dentro de otra interrupción.

PendSV

- La solución de que el cambio de contexto sea ejecutado dentro del SysTick tiene grandes limitaciones.
- Suponga que una IRQ sucede justo antes de la excepción del SysTick.
- La excepción del SysTick hace una expropiación del IRQ handler.
- El handler del IRQ queda sin atender (delayed).
- Por defecto, el diseño de Cortex-M no permite volver a modo Thread con un servicio de interrupción activo.

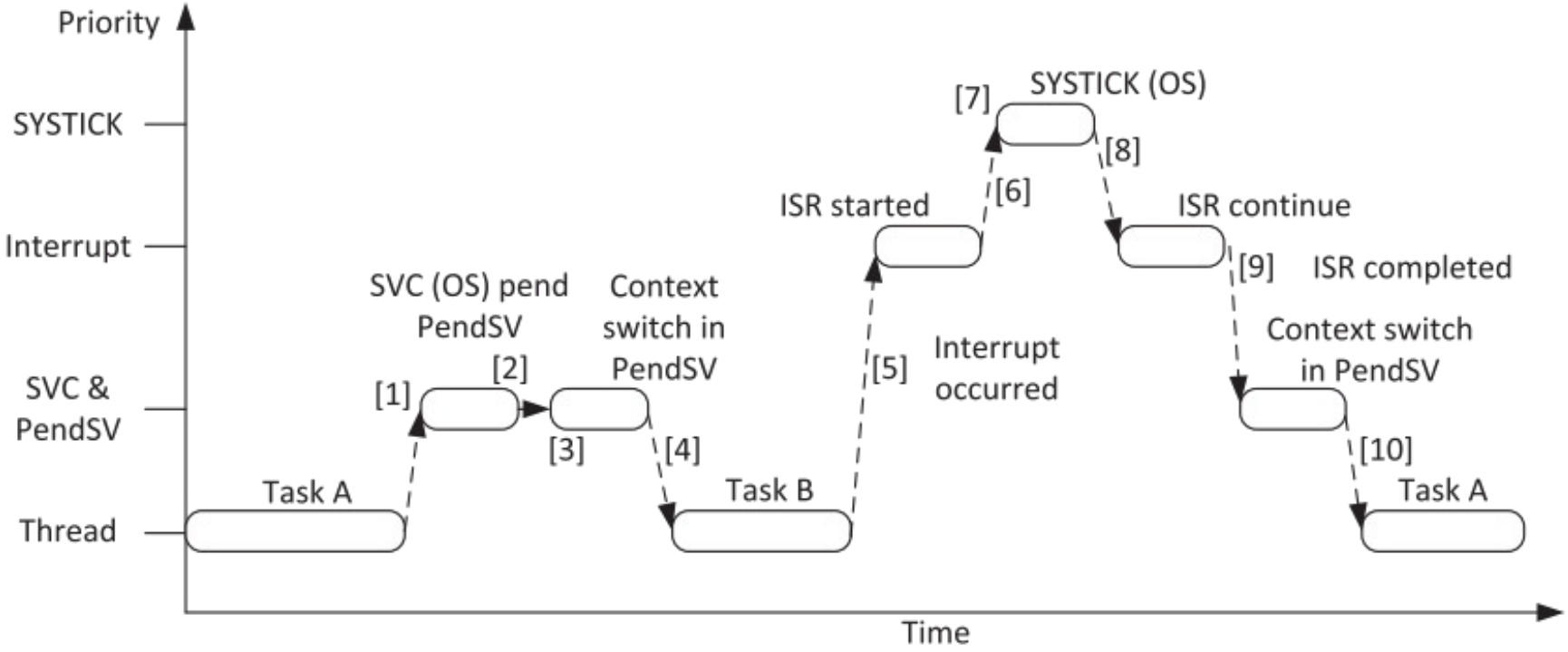
PendSV


Usage fault: return to thread
with active interrupt
Priority
SysTick
IRQ
Thread
OS
Context
switching
OS
Context
switching
OS
IRQ
IRQ
Task A
Task B
Task A
Time
IRQ processing
delayed

PendSV

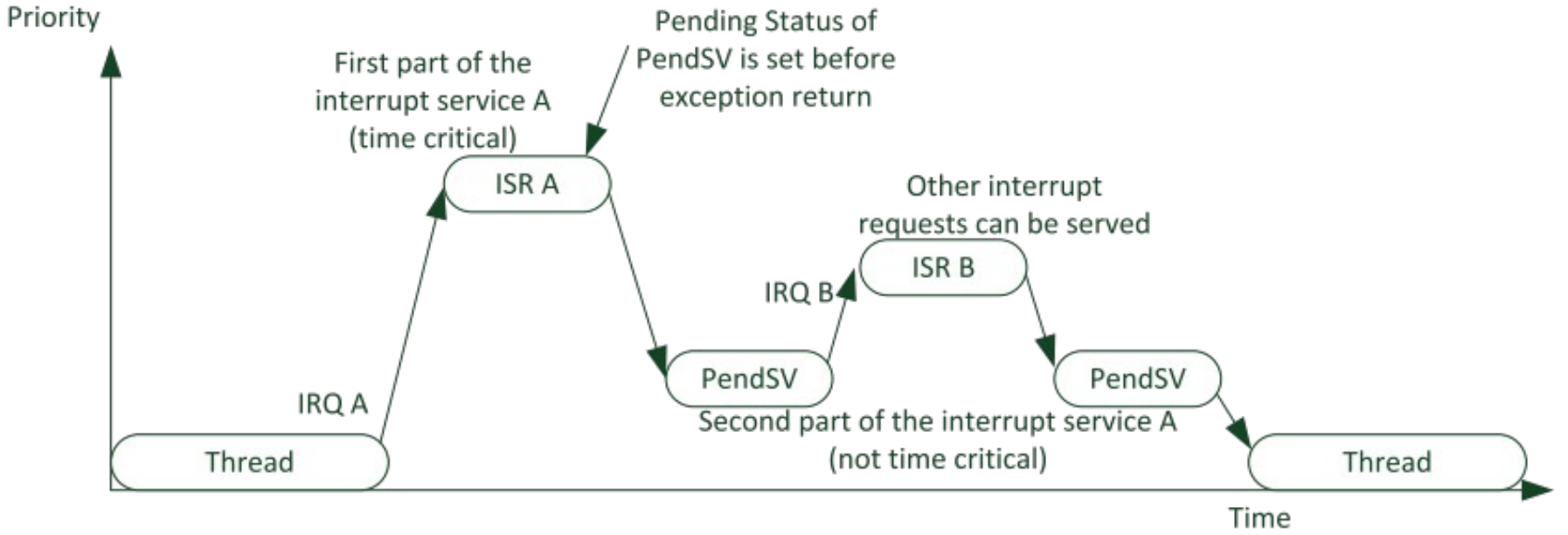
- Para solucionar esto, simplemente se puede evitar el cambio de contexto si existe una IRQ pendiente.
- No es recomendable: afecta la performance del OS al negar cambios de contexto en caso de varias IRQ cercanas a Systick.
- La solución es utilizar PendSV.

PendSV


Priority
SYSTICK
Interrupt
SVC &
PendSV
Thread
Task A
[1]
SVC (OS) pend
PendSV
[2]
[3]
Context
switch in
PendSV
[4]
Task B
[5]
Interrupt
occurred
ISR started
[6]
[7]
SYSTICK (OS)
[8]
ISR continue
[9]
ISR completed
Context switch
in PendSV
[10]
Task A
Time

PendSV

- Existe la posibilidad de que un IRQ Handler expropie el CPU a PendSV. La implementación debe contemplar eso.



ARM Architecture Procedure Call Standard

(AAPCS)

AAPCS

- Para facilitar la implementación del OS, se busca que la mayor parte esté escrita en lenguaje C.
- Solamente lo indispensable se escribe en ASM.
- El scheduling y el manejo previo de memoria para el cambio de contexto suele hacerse dentro de PendSV.
- Al combinar C y ASM es necesario conocer los estándares que utiliza el compilador de C.

AAPCS

Regis-ter	Syn-onym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8		Variable-register 8.
r10	v7		Variable-register 7.
r9		v6 SB TR	Platform register. The meaning of this register is defined by the platform standard.
r8	v5		Variable-register 5.
r7	v4		Variable-register 4.
r6	v3		Variable-register 3.
r5	v2		Variable-register 2.
r4	v1		Variable-register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.

AAPCS

- R0-R3 se utilizan para pasar argumentos a las funciones.
- R0-R1 se utilizan para almacenar valores de retorno de las funciones.
- El registro R12 se utiliza como scratch para las funciones y también por el linker.
- R4-R8, R10 y R11 se utilizan para alojar variables locales.
- En el caso de ser necesaria variables de 64bits (uint64_t, long int, double) se utilizan registros consecutivos.

AAPCS - Restricciones

- AAPCS implica algunas restricciones universales para el manejo del stack:
  - Stack-limit < SP <= stack-base. El stack pointer tiene que estar dentro de la extensión del stack.
  - SP mod 4 = 0. Todo el tiempo el stack debe estar alineado a un word de 32bits (no se permite apuntar a valores de byte o halfword).
  - Un proceso solo puede guardar datos en el stack dentro del intervalo cerrado determinado por [SP; stack base -1].
- Para una interface pública se requiere que SP mod 8 = 0 (alineación double word. Implica padding en ocasiones).

AAPCS - Retorno de resultados

- Tipos de datos fundamentales menores de 4 bytes se extienden (según signo) a 4 bytes y retornan en R0.
- Tipos de datos fundamentales double-word (64bits) retornan en R0-R1.
- Existen más reglas, pero para la extensión de esta materia solamente veremos estas dos.

AAPCS - Pasaje de Argumentos

- El pasaje de argumentos es un poco más laborioso.
- Se utiliza tanto los registros R0-R3 como el stack.
- Para argumentos de extensión menor a 4 words, se ubican los datos en orden comenzando desde R0 hasta R3.

AAPCS

```
uint32_t foo(uint32_t a, uint32_t b, uint64_t c)  {
// R0 = a; R1 = b;  R3 = HIGH(c); R4 = LOW(c);

   /*****    codigo de la funcion  *****/

return bar;
//  R0 = bar;
}
```

Gracias.