Lecture 05 ARM System call

Dr. Tushar, Mosaddek Hossain Kamal
Professor

Computer Science and Engineering, University of Dhaka,

BSc Third Year, Semester 2 (Sep – Feb), Academic Year: 2024

CSE3201: Operating Systems

Oct 29, 2024

Outline

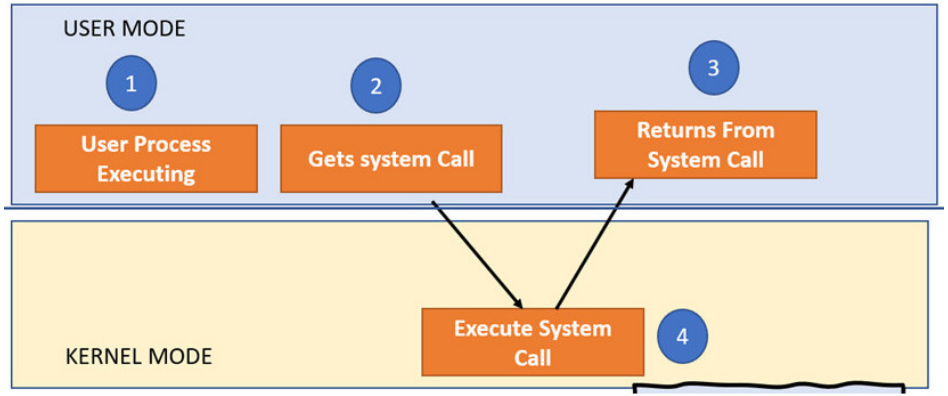
Learning Outcome

Learning Outcome

- Understand the Cortex-M4 Processor Major Components
  - NVIC, SysTick, FPU, Special Registers, and so on
  - Real-time clock
- **Understand the use of Cortex-M4 lower level programming:**
  - System call, Task and Memory management
  - thread, Synchronization tools
  - Exception handling
- Understanding ARM Memory map and address space
  - Flash memory, SRAM
  - Peripheral address space
  - Kernel stack frame and process/task stack frame

User and Kernel Mode Execution

- Simplistically, execution state consists of
  - Registers, Processor Mode, PC, and SP
- User application and the kernel have their own execution state
- System call mechanism safely transfers from user-execution to kernel-execution and back
- It is a way that an application requests a service from the kernel of the operating system
- Let an user program wants to read data from files or send data to the outside world need kernel service.

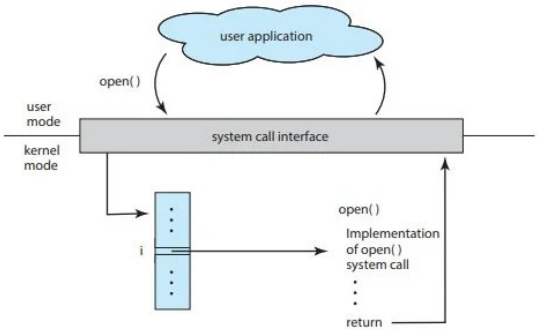


System Call Mechanism in principle

- Processor Mode: user mode –unprivileged, kernel mode – privileged mode
  - Switched from user mode to kernel mode
  - Switched back when returning to user mode
- SP (Stack Pointer)
  - User-level SP (PSP) is saved, and Kernel SP (MSP)is initialized (or loaded)
  - User-level SP (PSP) is restored when returning to user mode
- PC (Program counter)
  - User level PC is saved and PC set to kernel entry point
  - User PC restored when returning to user mode
  - Kernel entry via designated entry point must be strictly enforced

System Call Mechanism in principle

- Registers
  - Set at the user level to indicate system a call type and its arguments
    - a convention between application and the kernel
  - Some registers are preserved at user-level or kernel-level to restart user-level execution
    - Depends on language calling convention etc.
  - Result of system call placed in registers when returning to user-level
    - Another convention

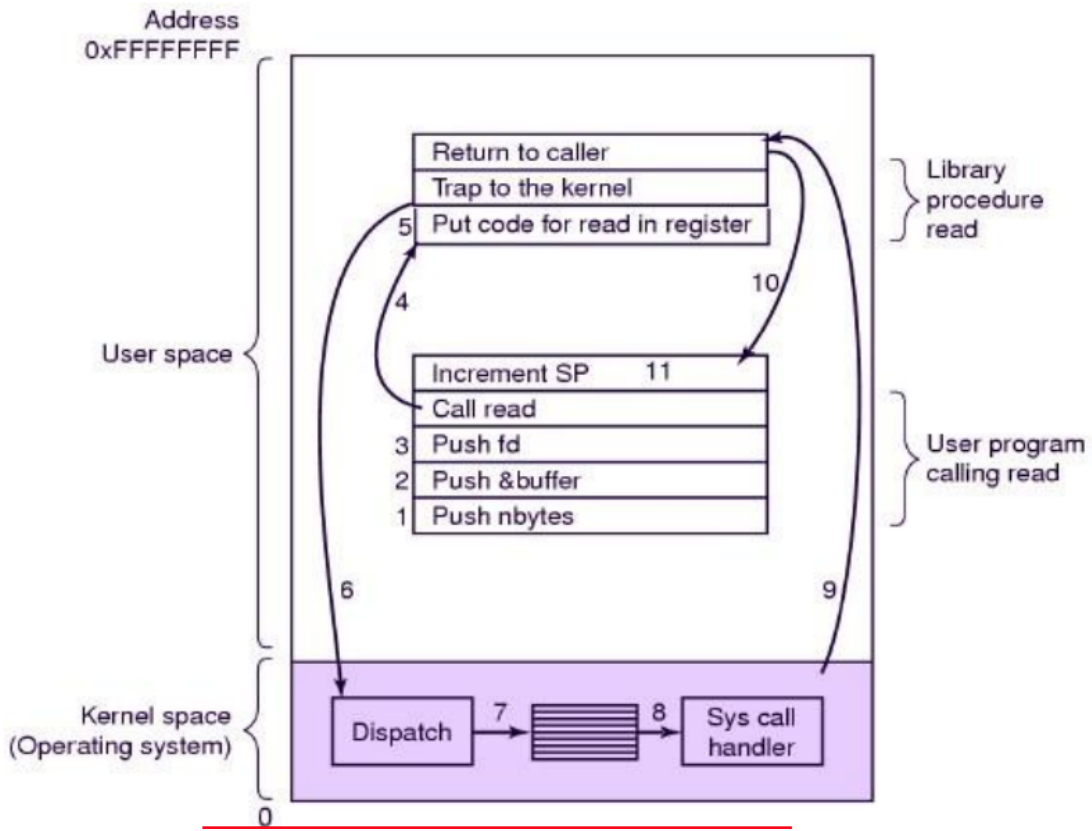


Why do we need system call

Why not simply jump into kernel via a function call???

- Function call do not
  - Change from user mode to kernel mode
  - and eventually back again
- Restrict possible entry points to secure locations

Steps to Making a System Call



There are 11 steps in making the system call read(fd, buffer, nbytes)

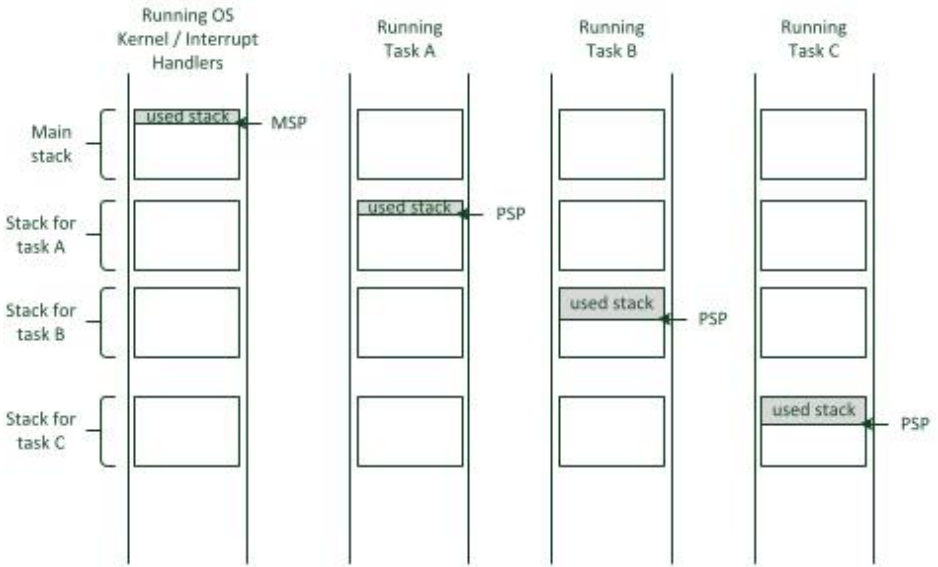
System Call : SVC and PendSV

ARM OS support feature

- Cortex-M processor design with OS support in mind
- Several features are available to make OS implementation easier
  - Shadowed stack pointer: MSP used for OS kernel; PSP for application
  - SysTick timer: a simple timer included inside the processor
  - SVC and PendSV exception: required for OS context switching
  - Unprivileged execution level allows basic security restricts access rights of the application tasks
  - Exclusive accesses: the exclusive load and store useful for semaphore as mutual exclusion
- Low interrupt latency: context switching overhead is low

Shadowed stack pointer

- MSP: Main Stack Pointer. It is used in thread mode when the control bit [1] SPSEL is '0.'
- MSP is always used in handler mode
- PSP: Processor/process stack pointer PSP is used in thread mode when the control bit is set to '1.'
- PUSH and POP operation uses SP (R13) as the currently selected stack pointer
- Processor can access MSP and PSP using MRS and MSR instructions
- In privileged mode, you can MSP for all operations ignoring PSP
- Usually OS like RTOS uses MSP for handler mode and part of the kernel



Shadowed Stack: Benefit and Initialization

Shadowed Stack: Benefit

- If application stack corrupts, Kernel stack and another task stack intact and thus improve system reliability
- The stack space needed in each task to cover maximum stack usage and one level of the stack frame
  - Contains 9-word in Cortex-M3 and 27 words in Cortex-M4 (including FPU registers)
  - the MPU usage raises memory fault for stack overflow
- When the system power-up it initializes the MSP; however, OS can reinitialize the main stack
- it then assigns (MSR) stacks PSPs for each task to initialize
- the write the control register to set SPSEL

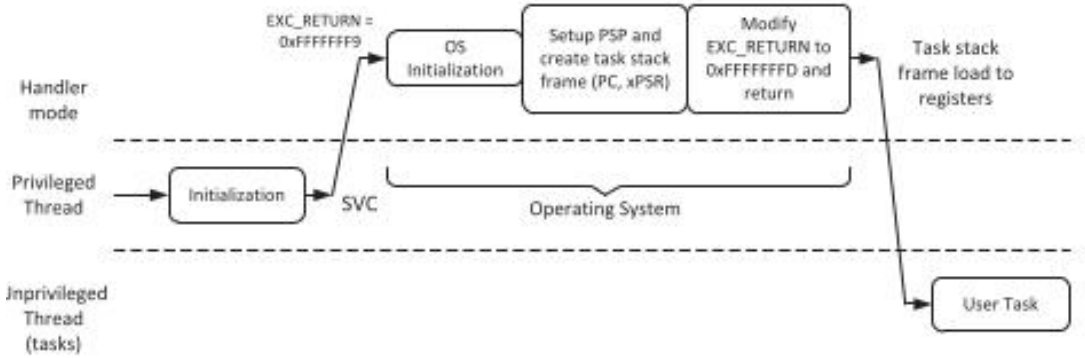
```
LDR R0,=PSP_TOP; PSP_TOP is a constant defines the top address of stack
MSR PSP, R0; Set PSP to the top of a process stack
MRS R0, CONTROL; Read current CONTROL
ORRS R0, R0, #0x2 ; Set SPSEL
MSR CONTROL, R0 ; write to CONTROL
ISB      ; Execute and ISB after updating CONTROL,
         ; this is an architectural recommendation
```

Stack Pointer and Stack Frame

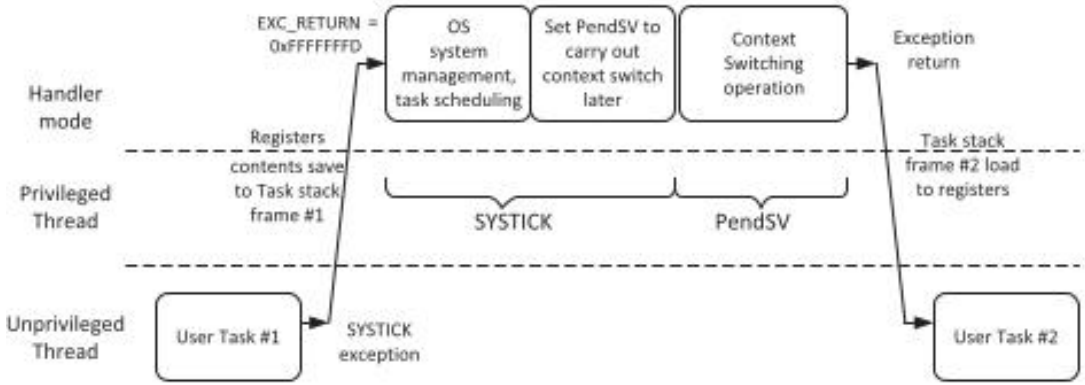
- In Handler Mode OS update and program PSP directly and use an exception sequence to run the task
- Initialize a task: IN thread mode OS first use SVC exception
- In SVC handler OS create process/task stack frame in the process stack
- Trigger an exception return
- When the task stack frame is loaded – application task started
- SysTick handler load the task
- Inside context switch:
  - save the current status of the registers of the current task stack frame
  - save the current PSP value
  - Set the PSP to the last SP value for the next task
  - restore the last values for the next tasks
  - use exception return to switch to the next task

Initialize a task

Initialize a Task after start of OS



Context switch



The Context switching is carried out in PendSV

SVC and PendSV Exception

SVC (Supervisor call) usually known as System call

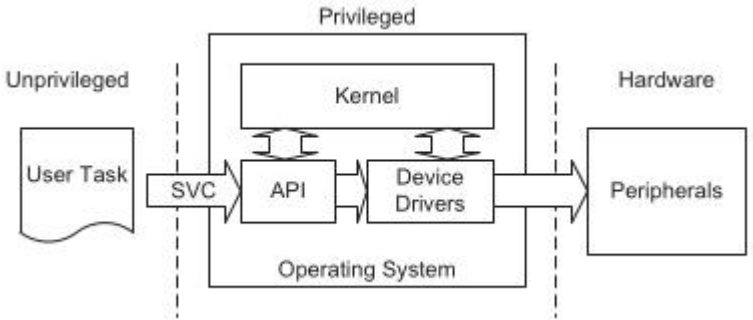
- Exception number 11
- Programmable priority level
- SVC is triggered by SVC instruction
- NVIC vs. SVC
  - Interrupts are imprecise
  - Interrupt executes several instructions before invoke
  - On the other hand, SVC is precise
  - Handler must run after the SVC instruction except higher priority interrupt is invoked
  - In many application systems, SVC is used to access system resources
  - application tasks are running in unprivileged mode
  - some hardware can be accessed or set up in the privileged mode
  - the only way the application can access the resources vis service from OS
  - This creates a convenient environment not to worry about the detail of the hardware used

SVC and PendSVC

SVC and PendSV

- SVC allows application development independently of the OS
  - Application does not need to know the exact address of the OS service function
  - the actual hardware level execution
  - the application needs to know only the service number and parameters
- SVC instruction generates SVC
- the immediate value required for SVC service number
- and parameters, for example:

```
SVC #0x3; Call SVC function 3
SVC 0x3; Traditional syntax acceptable in ARM
```



SVC System call

- In Keil C, SVC instruction can be generated using '__svc' function
- in GCC and some other toolchains, this can be generated using inline assembly
- when SVC handler executed, you can determine the immediate data value
  - SVC reading the stacked PC
  - then reading the instruction from that address and masking out the unneeded bits
  - SVC can use PSP or MSP by reading stacked PC value
  - , therefore, we need to find out which stack before reading the stacked PC
  - This (PSP or MSP) can be determined from the link register; see code below

```
SVC_Handler
        TST LR, #4; Test bit 2 of EXC_RETURN
        ITE EQ;
        MRSEQ   R0, MSP; if 0, stacking used MSP, copy to R0
        MRSNE   R0, PSP; if 1, stacking used PSP, copy to R0
        LDR     R0, [R0, #24]; Get stacked PC from the stack frame
                        ; (stacked PC = address of instruction after SVC)
        LDRB    R0, [R0, #-2]; Get first byte of the SVC instruction
                        ; now the SVC number is in R0
        ...
```

SVC in gcc

SVC in C

- SVC handler separated into two parts
  - First extract the starting address of the stackframe
  - passes it to the second part
  - this needs assembly to check the value of LR (EXEC_RETURN)
  - the second part extracted the stacked PC from stackframe and SVC number from the program code

Example: ARM keil:

```
__asm void SVC_Handler(void)
{
TST LR, #4
ITE EQ; Test bit 2 of EXC_RETURN
MRSEQ R0, MSP; if 0, stacking used MSP, copy to R0
MRSNE R0, PSP; if 1, stacking used PSP, copy to R0
B __cpp(SVC_Handler_C)
ALIGN 4
}
```

SVC Service number in Assembly

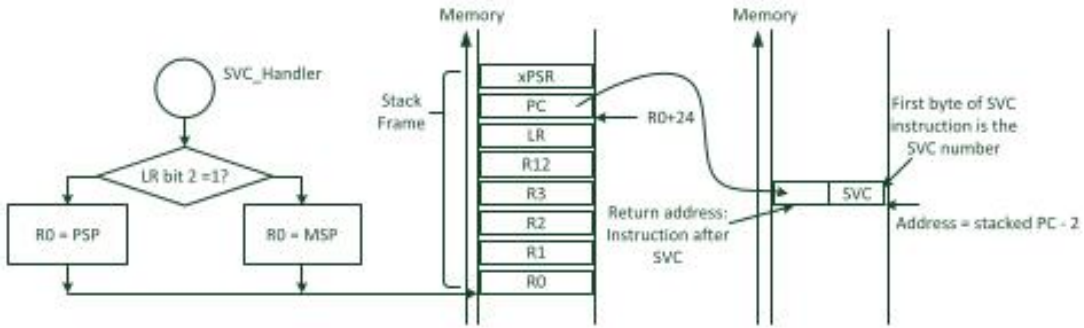


Figure 1: SVC service number in Assembly

- You need to check the LR register to get the address of the stack frame.
- Passes it to the second part as input parameter
- You need assembly here to get the value of LR
- the second part extract PC from stack frame, number from the program code
- and other values of the stacked registers

SVC in C : second part

```
void SVC_Handler_C(unsigned int * svc_args)
 {
uint8_t svc_number;
uint32_t stacked_r0, stacked_r1, stacked_r2, stacked_r3;
svc_number = ((char *) svc_args[6])[-2]; //Memory[(Stacked PC)-2]
stacked_r0 = svc_args[0];
stacked_r1 = svc_args[1];
stacked_r2 = svc_args[2];
stacked_r3 = svc_args[3];
// ... other processing
...
// Return result (e.g. sum of first two arguments)
svc_args[0] = stacked_r0 + stacked_r1;
return;
}
```

The values are in a stack frame because any higher priority exception or interrupt may change the registers' values if it arrives simultaneously.

- The advantage to passing stack frame address enables us to extract any information from stack frame, including stacked registers
- this is essential to pass values and return values for the OS services
- if a higher priority interrupt then the higher priority interrupt execute first

Example of SVC: passing parameter and return values

```
#include <stdio.h>
// Define SVC functions
int __svc(0x00) svc_service_add(int x, int y); // Service #0 : Add
int __svc(0x01) svc_service_sub(int x, int y); // Service #1 : Sub
int __svc(0x02) svc_service_incr(int x); // Service #2 : Incr
void SVC_Handler_main(unsigned int * svc_args);
// Function declarations
int main(void)
{
int x, y, z;
x = 3; y = 5;
z = svc_service_add(x, y);
printf ("3+5 = %d \n", z);
x = 9; y = 2;
z = svc_service_sub(x, y);
printf ("9-2 = %d \n", z);
x = 3;
z = svc_service_incr(x);
printf ("3++ = %d \n", z);
while(1);
}
```

Example of SVC: passing parameter and return values

```
// SVC handler - Assembly wrapper to extract
//stack frame starting address
__asm void SVC_Handler(void)
{
TST LR, #4 ; Test bit 2 of EXC_RETURN
ITE EQ
MRSEQ R0, MSP; if 0, stacking used MSP, copy to R0
MRSNE R0, PSP; if 1, stacking used PSP, copy to R0
B __cpp(SVC_Handler_C)
ALIGN 4
}
// SVC handler - main code to handle processing
// Input parameter is stack frame starting address
// obtained from assembly wrapper.
void SVC_Handler_main(unsigned int * svc_args)
{
// Stack frame contains:
// r0, r1, r2, r3, r12, r14, the return address and xPSR
// - Stacked R0 = svc_args[0];
// - Stacked R1 = svc_args[1];
// - Stacked R2 = svc_args[2];
// - Stacked R3 = svc_args[3];

--- next slide ---
```

Example of SVC: passing parameter and return values

—- Continue from the previous slide —

```
// - Stacked R12 = svc_args[4]
// - Stacked LR = svc_args[5]
// - Stacked PC = svc_args[6]
// - Stacked xPSR = svc_args[7]
unsigned int svc_number;
svc_number = ((char *)svc_args[6])[-2];
switch(svc_number)
{
case 0: svc_args[0] = svc_args[0] + svc_args[1];
break;
case 1: svc_args[0] = svc_args[0] - svc_args[1];
break;
case 2: svc_args[0] = svc_args[0] + 1;
break;
default: // Unknown SVC request
break;
}
return;
}
```

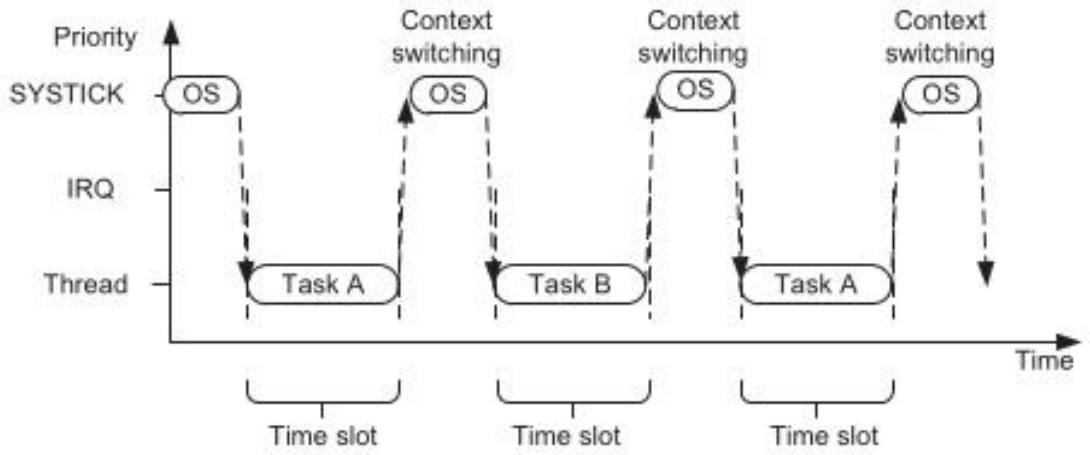
Rationale: you cannot use SVC with it in SVC – it raises the HardFault exception. Note that ARM7TDMI has SWI instruction for similar functionality.

Control Register For SPSEL (RW) and nPRIV (RW)

Bits	Function
Bits 31:3	Reserved
Bit 2	**FPCA:** Indicates whether floating-point context currently active: 0: No floating-point context active 1: Floating-point context active. The Cortex-M4 uses this bit to determine whether to preserve floating-point state when processing an exception.
Bit 1	**SPSEL:** Active stack pointer selection. Selects the current stack: 0: MSP is the current stack pointer 1: PSP is the current stack pointer. In Handler mode this bit reads as zero and ignores writes. The Cortex-M4 updates this bit automatically on exception return.
Bit 0	**nPRIV:** Thread mode privilege level. Defines the Thread mode privilege level. 0: Privileged 1: Unprivileged.

PendSV – Pending Service Exception

- This important and key exception for supporting OS operation
- Using the characteristic of PendSV, it is use for contex switching
- By setting lowest priority it can be excuteted after all high-priority handler
- Following example shows the context switch for two tasks



PendSV context switching

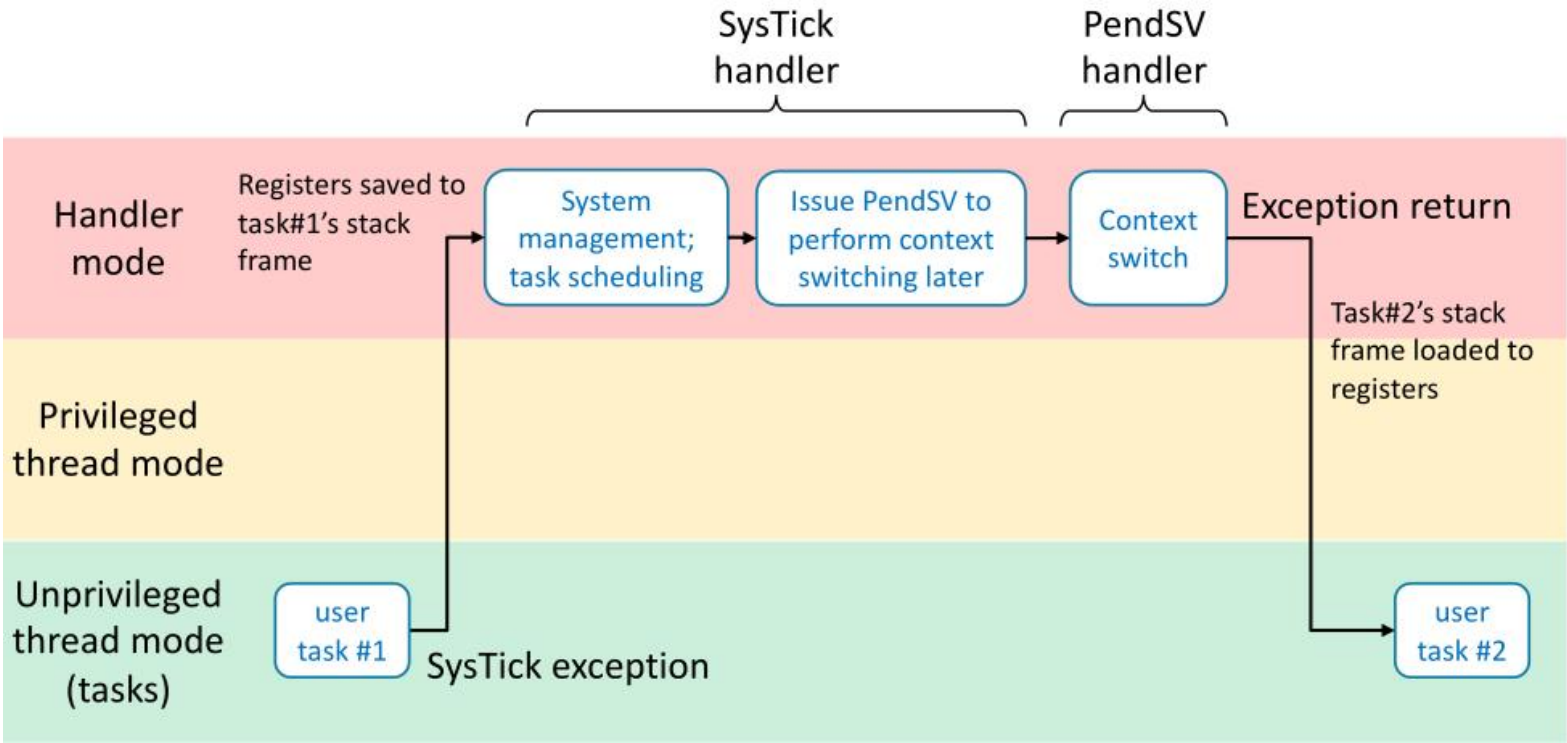
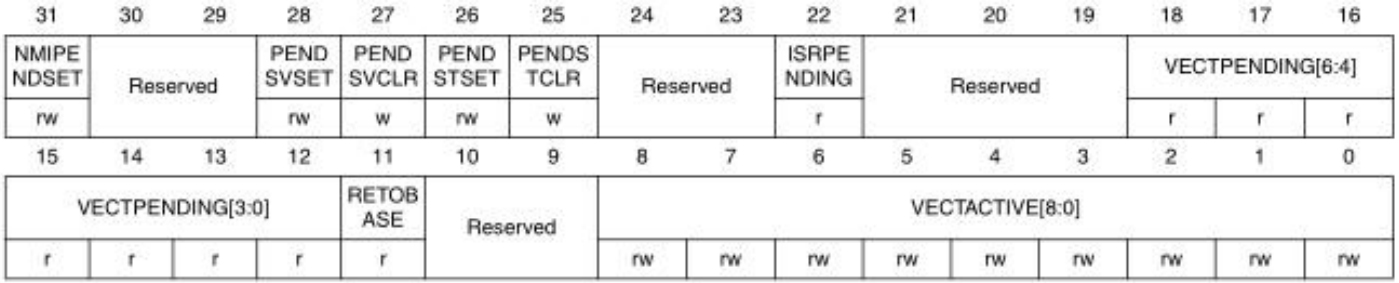


Figure 2: PendSV in action for context switching steps

PendSV Set and Clear

- Triggered by setting a flag in Interrupt Control and State Register (ICSR) in SCB
- Not as precise as SVC
- Can be set inside a higher-priority ISR and executed when the ISR finishes



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
NMIPENDSET	Reserved		PENDSVSET	PENDSVCLR	PENDSTSET	PENDSTCLR	Reserved		ISRPENDING	Reserved			VECTPENDING[6:4]		
rw			rw	w	rw	w			r				r	r	r

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
VECTPENDING[3:0]				RETOBASE	Reserved		VECTACTIVE[8:0]								
r	r	r	r	r			rw	rw	rw	rw	rw	rw	rw	rw	rw

```
Bit 28 PENDSVSET: PendSV set-pending bit.
Write:
0: No effect
1: Change PendSV exception state to pending.
Read:
0: PendSV exception is not pending
1: PendSV exception is pending
Writing 1 to this bit is the only way to set the PendSV exception state to pending.
Bit 27 PENDSVCLR: PendSV clear-pending bit. This bit is write-only. On a read,
value is unknown.
0: No effect
1: Removes the pending state from the PendSV exception.
```

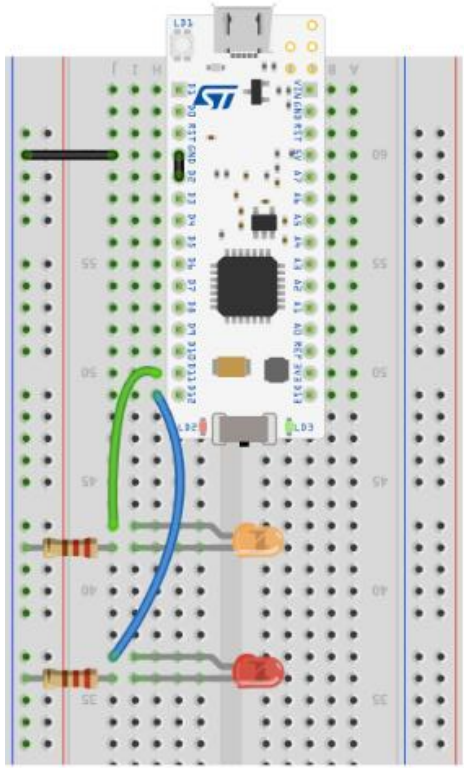
System Control Block (SCB) Registers: '0xE000E010'

```
typedef struct
{
volatile uint32_t CPUID;  // CPUID Base Register 0x0
volatile uint32_t ICSR;   // Interrupt Control and State Register 0x4
volatile uint32_t VTOR;   // Vector Table Offset Register 0x8
volatile uint32_t AIRCR;  // Application Interrupt and Reset Control Register 0xC
volatile uint32_t SCR;    // System Control Register   0x10
volatile uint32_t CCR;    // Configuration and Control Register 0x14
volatile uint32_t SHPR1;  // System Handler Priority Register 1 0x18
volatile uint32_t SHPR2;  // System Handler Priority Register 2 0x1C
volatile uint32_t SHPR3;  // System Handler Priority Register 3  0x20
volatile uint32_t SHCSR;  // System Handler Control and State Register  0x24
volatile uint32_t CFSR;   // Configurable Fault Status Register combined of MemManage
//Fault Status Register, BusFault Status Register, UsageFault Status Register 0x28
volatile uint32_t HFSR;   // HardFault Status Register 0x2C
uint32_t RESERVED;        // 0x30
volatile uint32_t MMAR;   // MemManage Fault Address Register  0x34
volatile uint32_t BFAR;   // BusFault Address Register  0x38
volatile uint32_t AFSR;   // Auxiliary Fault Status Register 0x3C
uint32_t RESERVED1[18];   // 0x40
volatile uint32_t CPACR;  // Coprocessor access control register 0x88
} SCB_TypeDef;
```

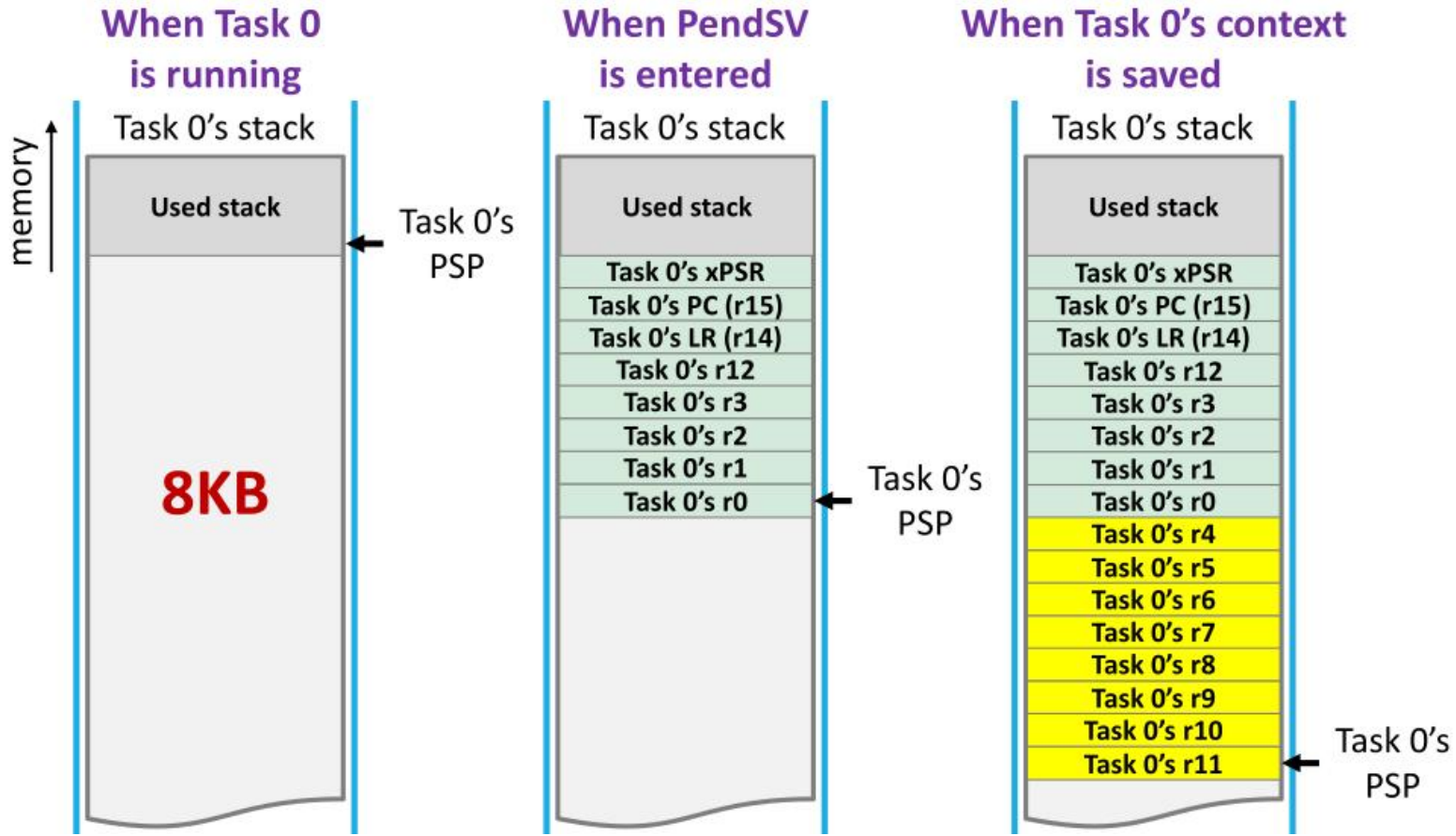
PendSV and context switch – Demonstration

The following program demonstrates very simple preemptive multitasking OS

- The program consists of three concurrent tasks:
- task0() blinks the on-board green LED every 500ms
- task1() blinks the yellow LED every 1000ms
- task2() blinks the red LED every 1500ms
- Task priority not implemented



Context Switch – Demo



Demo Sample Code (1)

Ref. RTOS context-switch slides

```
#include "stm32l4xx.h"
#define LED0 (1<<3)
#define LED1 (1<<5)
#define LED2 (1<<4)
/* Macro for word accesses */
#define HW32_REG(ADDRESS) (*((volatile unsigned long *)(ADDRESS)))
#define stop_cpu() while(1);
void LED_initialize(void); // Initialize LED
void task0(void); // Toggle LED0
void task1(void); // Toggle LED1
void task2(void); // Toggle LED2
volatile uint32_t systick_count=0;
// Stack for each task (8Kbytes each - 1024 x 8 bytes)
long long task0_stack[1024], task1_stack[1024], task2_stack[1024];
// Data use by OS
uint32_t curr_task = 0; // Current task
uint32_t next_task = 1; // Next task
uint32_t PSP_array[3]; // Process Stack Pointer (PSP) for each task
```

Demo Sample Code (2)

```
int main(void) {
LED_initialize();
// Create stack frame for task0
PSP_array[0] = ((uint32_t) task0_stack) + sizeof(task0_stack) - 16*4;
HW32_REG((PSP_array[0] + (14*4))) = (uint32_t) task0; // initial Program Counter
HW32_REG((PSP_array[0] + (15*4))) = 0x01000000; // initial xPSR
// Create stack frame for task1
PSP_array[1] = ((uint32_t) task1_stack) + sizeof(task1_stack) - 16*4;
HW32_REG((PSP_array[1] + (14*4))) = (uint32_t) task1; // initial Program Counter
HW32_REG((PSP_array[1] + (15*4))) = 0x01000000; // initial xPSR
// Create stack frame for task2
PSP_array[2] = ((uint32_t) task2_stack) + sizeof(task2_stack) - 16*4;
HW32_REG((PSP_array[2] + (14*4))) = (uint32_t) task2; // initial Program Counter
HW32_REG((PSP_array[2] + (15*4))) = 0x01000000; // initial xPSR
// Start first task
curr_task = 0; // Switch current task to task #0
__set_PSP((PSP_array[curr_task] + 16*4)); // Set PSP to top of task 0 stack
NVIC_SetPriority(PendSV_IRQn, 0xFF); // Set PendSV to lowest possible priority
NVIC_SetPriority(SysTick_IRQn, 0xFF); // Set SysTick to lowest possible priority
SysTick_Config(4000); // 1000 Hz SysTick interrupt on default 4MHz core clock
__set_CONTROL(0x3); // Switch to use Process Stack, unprivileged state
__ISB(); // Execute ISB after changing CONTROL (architectural recommendation)
task0(); // Jump to task 0
stop_cpu(); // Should not be here
```

Demo code (3)

GPIO for 3 LEDs are configured as output using CMSIS without STM32 HAL API

```
void LED_initialize(void) {
RCC->AHB2ENR |= RCC_AHB2ENR_GPIOBEN; // Enable Port B clock
// Set pin 3, 4, 5 as general-purpose output mode (pull-push)
GPIOB->MODER |= (GPIO_MODER_MODER3_0 |
GPIO_MODER_MODER4_0 |
GPIO_MODER_MODER5_0);
GPIOB->MODER &= ~((GPIO_MODER_MODER3_1 |
GPIO_MODER_MODER4_1 |
GPIO_MODER_MODER5_1));
GPIOB->OSPEEDR |= (GPIO_OSPEEDER_OSPEEDR3 | // 100MHz operations
GPIO_OSPEEDER_OSPEEDR4 |
GPIO_OSPEEDER_OSPEEDR5);
GPIOB->PUPDR = 0; // No pull up, no pull down
return;
}
```

Demo Code (4)

Each task declars its own local variable

```
void task0(void) { // Toggle LED #0
while (1) {
uint32_t ts = systick_count;
while (systick_count - ts < 250); // wait 250ms
GPIOB->ODR ^= LED0;
};
}
void task1(void) { // Toggle LED #1
while (1) {
uint32_t ts = systick_count;
while (systick_count - ts < 500); // wait 500ms
GPIOB->ODR ^= LED1;
};
}
void task2(void) { // Toggle LED #2
while (1) {
uint32_t ts = systick_count;
while (systick_count - ts < 1000); // wait 1000ms
GPIOB->ODR ^= LED2;
};
}
```

Demo: Code (5)

- SysTick handler schedules tasks in a round-robin fashion
- Signal PendSV interrupt to do actual context switching

```
void SysTick_Handler(void) { // Triggered at 1KHz
systick_count++;
// Simple task round robin scheduler
next_task = 0;
switch (curr_task) {
case 0: next_task = 1; break;
case 1: next_task = 2; break;
case 2: next_task = 0; break;
default: next_task = 0; stop_cpu(); break; // Should not be here
}
if (curr_task != next_task) { // Context switching needed
SCB->ICSR |= SCB_ICSR_PENDSVSET_Msk; // Set PendSV to pending
}
return;
}
```

Demo: Code (6)

- Context switching code runs inside PendSV handler
- Assume no floating-point support (otherwise, more registers need to be saved)

```
void __attribute__((naked)) PendSV_Handler(void) {
// ------------------------
// Save current context
asm("MRS R0, PSP"); // Get current process stack pointer value
asm("STMDB R0!, {R4-R11}"); // Save R4 to R11 in task stack (8 regs)
asm("LDR R1, =curr_task");
asm("LDR R2, [R1]"); // Get current task ID
asm("LDR R3, =PSP_array");
asm("STR R0, [R3, R2, LSL #2]"); // Save PSP value into PSP_array
// ------------------------
// Load next context
asm("LDR R4, =next_task");
asm("LDR R4, [R4]"); // Get next task ID
asm("STR R4, [R1]"); // Set curr_task = next_task
asm("LDR R0, [R3, R4, LSL #2]"); // Load PSP value from PSP_array
asm("LDMIA R0!, {R4-R11}"); // Load R4 to R11 from task stack (8 regs)
asm("MSR PSP, R0"); // Set PSP to next task
asm("BX LR"); // Return from exception
}
```

Task Scheduling and Interrupt

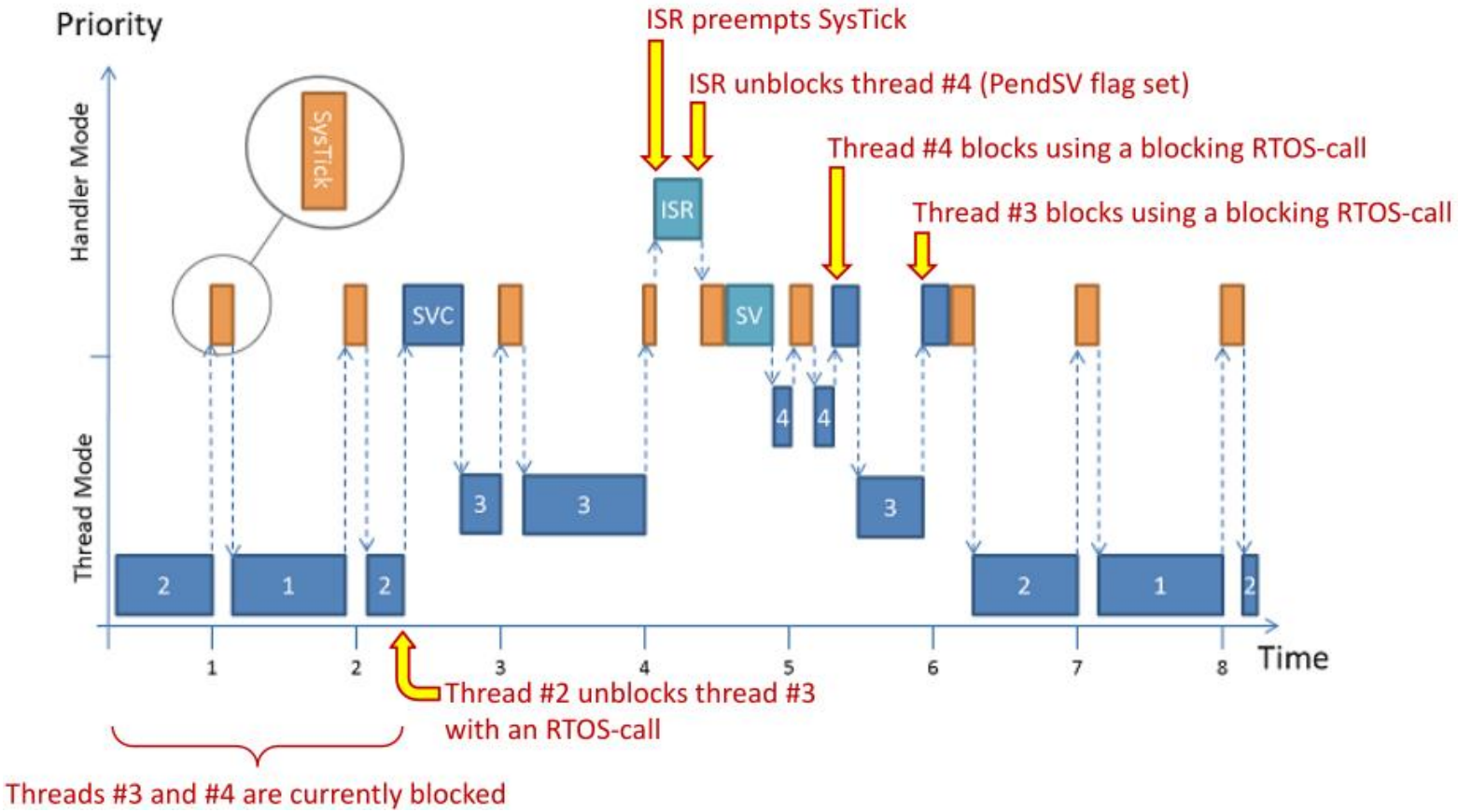


Figure 3: Example of Task scheduling and Interrupt

Code in ISR Context

- No blocking code in ISR, e.g.,
  - The wait() function
  - Infinite while loop
  - Other blocking calls
- No printf(), malloc() or new in ISR
  - These functions are not re-entrant
  - Their behaviors are undefined, data can be corrupted, or the program can crash, when called from an ISR