ECSE 426 – Fall 2015
Microprocessor Systems

Dept. Electrical and Computer Engineering

McGill University

Outline

- Interrupt prioritization
- Exception Handling

- OS Support
  - Supervisor Calls
  - Pending Service Call
  - Exclusive Access

Interrupt Prioritization

- Each interrupt source has an 8-bit interrupt priority value
- Bits divided into pre-empting and non-pre-empting “sub-priority” levels
  - Sub-priority levels only apply if the pre-empting priority levels are the same
  - Software programmable PRIGROUP register field of the NVIC
    - chooses how many of the 8-bits are used for “group-priority” and how many are used for “sub-priority”
  - Group priority is the pre-empting priority

- In the STM32F10x 16 levels (4-bit) of priority are implemented:

PRIGROUP (3 Bits)	Binary Point (group.sub)		Preempting Priority (Group Priority)		Sub-Priority	
			Bits	Levels	Bits	Levels
011	**4.0**	**gggg**	**4**	**16**	**0**	**0**
100	**3.1**	**gggs**	**3**	**8**	**1**	**2**
101	**2.2**	**ggss**	**2**	**4**	**2**	**4**
110	**1.3**	**gsss**	**1**	**2**	**3**	**8**
111	**0.4**	**ssss**	**0**	**0**	**4**	**16**

Interrupt Prioritization

- Lower numbers are higher priority
- Hardware interrupt number is the lowest level of prioritization
  - IRQ3 is higher priority than IRQ4 if the priority registers are programmed the same

- In STM32F10x 16 levels (4-bit) of priority implemented:

PRIGROUP (3 Bits)	Binary Point (group.sub)		Preempting Priority (Group Priority)		Sub-Priority	
			Bits	Levels	Bits	Levels
011	4.0	gggg	4	16	0	0
100	3.1	gggs	3	8	1	2
101	2.2	ggss	2	4	2	4
110	1.3	gsss	1	2	3	8
111	0.4	ssss	0	0	4	16

Exception handlers in C

- C compilers follow AAPCS (ARM Architecture Procedure Call Standard)
- C function can modify R0 to R3, R12, R14 (LSR) and PSR
  - caller saved registers
  - Code that calls a subroutine must save them to memory if it needs them after the function call

- If C function needs to use R4-R11, it should save these registers onto stack memory and restore them before exiting
  - Callee saved registers

- Typically R0-R3 are input parameters, and R0 is return result (+ R1 for 64 bit result)

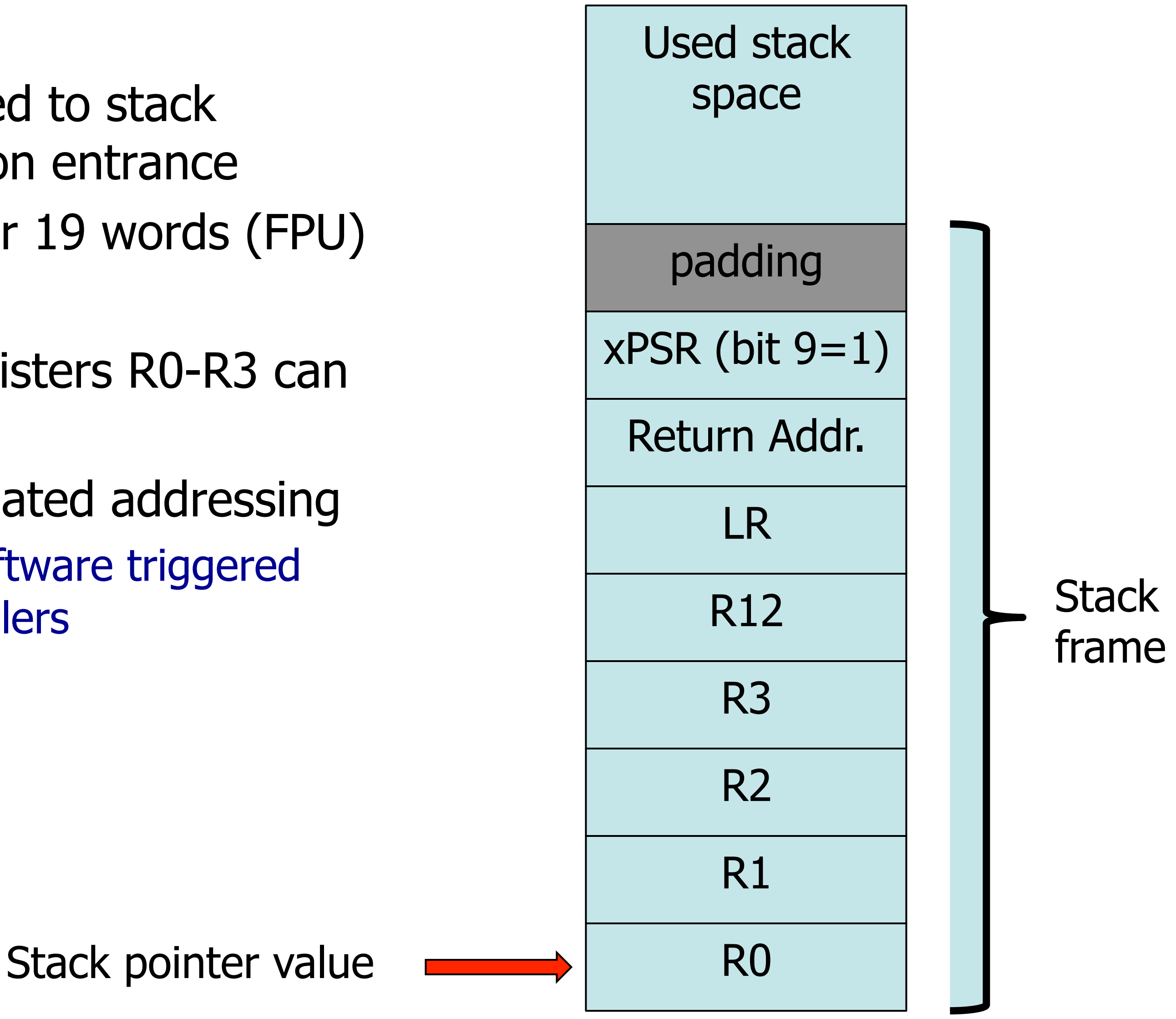
Exception handlers in C

- Exception mechanism needs to save R0-R3, R12, LR and PSR at exception entrance automatically
- Restore them at exception exit under control of processor’s hardware
- Value of return address (PC) is not stored in LR as in normal C function calls
  - Exception mechanism puts an EXC_RETURN code in LR at exception entry
  - Value of return sequence also needs to be saved
- 8 registers need to be saved (+ floating point)

Stack Frame

- Block of data pushed to stack memory at exception entrance
- 8 words (no FPU) or 19 words (FPU)

- General purpose registers R0-R3 can be easily accessed
- Use stack pointer related addressing
- Pass information to software triggered interrupts or SVC handlers



EXC_RETURN

- When processor enters exception handler or interrupt service routine
  - Value of link register (LR) updated to a code called EXC_RETURN
  - Used to trigger exception return mechanism when loaded into program counter (PC)

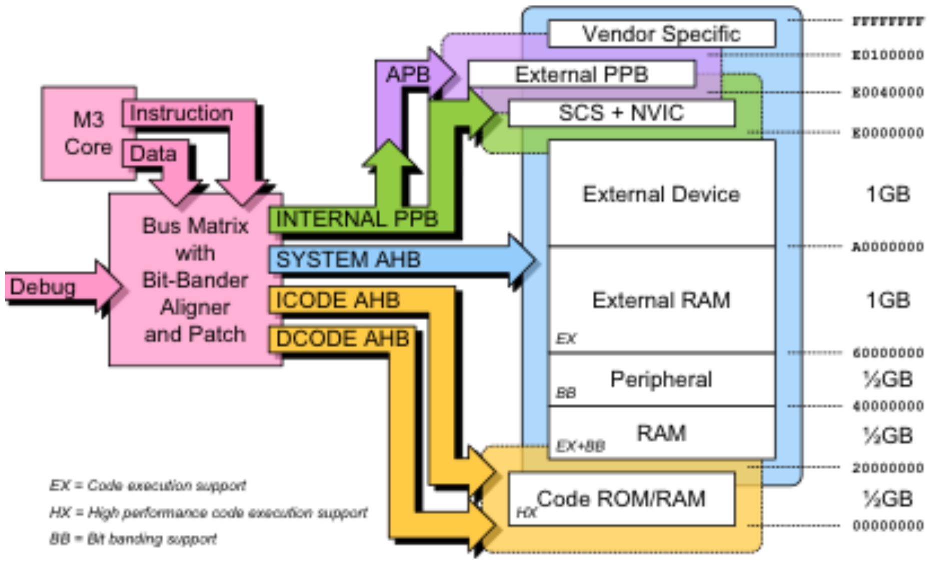
  - Some bits provide extra information about the exception sequence
  - 4: Stack frame type = 1 (8 words) or 0 (26 words)
  - 3: Return mode = 1 (Thread) or 0 (Handler)
  - 2: Return stack = 1 (Process stack) or 0 (Main stack)

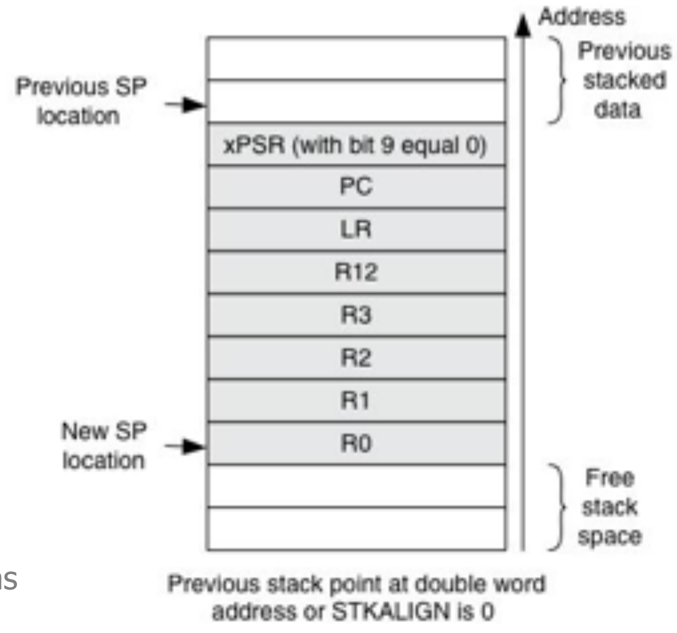
Exception sequence

- Exception occurs
  - need to push registers into stack (form stack frame)
  - perform vector fetch
  - and start exception handler instruction fetch

- Multiple bus interfaces
  - reduce interrupt latency: do stacking and flash memory access in parallel
  - Stacking: data bus
  - Vector fetch + instruction fetch: instruction bus

Cortex Interrupts





<u>Entry</u>

Processor state automatically saved to stack over data bus (SYSTEM)

{R0-R3, R12, LR, PC, xPSR}

In parallel, ISR pre-fetched on the instruction bus (ICODE)

ISR ready to start executing as soon as stack PUSH complete

<u>Exit</u>

Processor state is automatically restored from the stack

In parallel, interrupted instruction is prefetched for execution upon POP

Stack POP can be interrupted -> new ISR immediately executed

Interrupt latency

- Delay from start of interrupt request to start of interrupt handler execution
- If memory system has zero latency and bus system design allows vector fetch and stacking at same time, latency is only 12 clock cycles for Cortex-M
- Latency can be higher due to wait states in memory system
  - If processor is carrying out a memory transfer (including buffered write operations), transfer must be completed before exception sequence starts
- Cortex-M strives to minimize latency
  - Many operations handled by hardware, e.g. nested interrupt handling
  - No need to use software code to determine which interrupt to service
  - No need to use code to locate starting addresses of ISRs

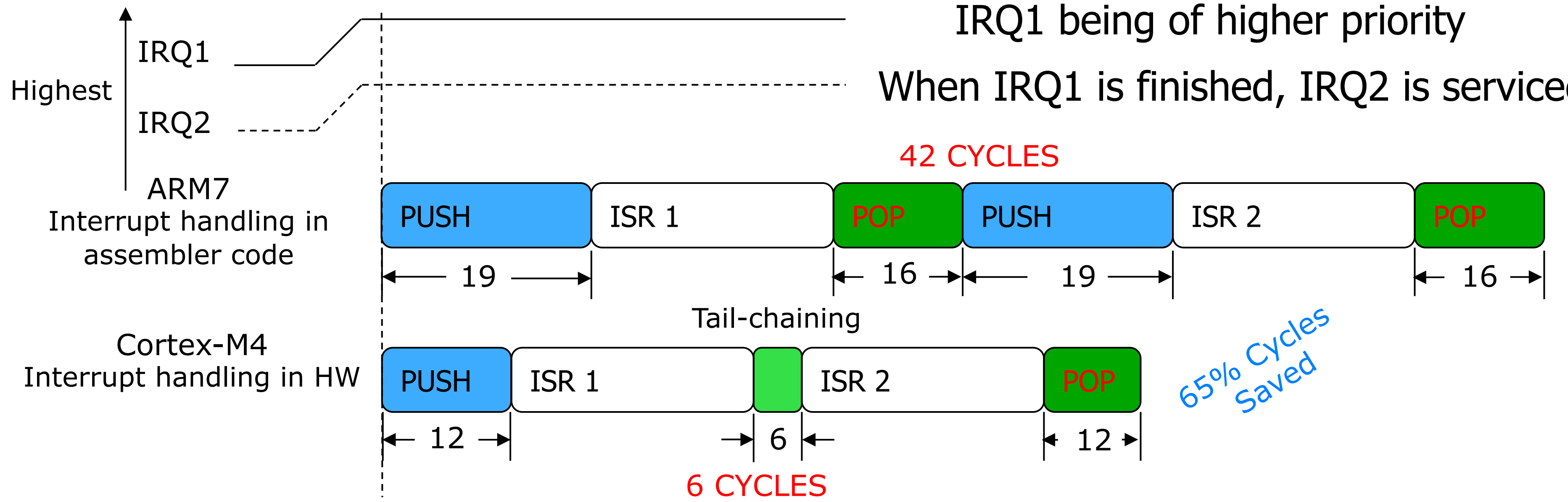
Tail chaining

- When exception occurs, but processor is handling another exception of same or higher priority, exception enters pending state
- Processor finishes executing current exception handler, proceeds to process pending request
- Processor skips the steps of restoring registers from stack (unstacking) then pushing them back onto stack
- Timing gap between the two exceptions is considerably reduced
  - only 6 cycles for memory system with no-wait state
- Tail chaining optimization makes system more energy efficient
  - reduces number of stack accesses and hence amount of memory transfer

Fast Interrupts: Tail Chaining

Case of 2 simultaneous interrupts, IRQ1 being of higher priority

When IRQ1 is finished, IRQ2 is serviced



In Cortex-M, ISR2 has only a 6-cycle delay. ISR2 has been tail-chained

ICODE (vector) and SYSTEM (stack) used in parallel

OS Support

- Shadowed stack pointer
  - Two stack pointers available
  - MSP used for OS kernel and interrupt handlers. PSP used by application tasks
- Systick timer
  - Simple timer included inside processor.
  - Embedded OS can be used on a wide range of Cortex-M microcontrollers
- Supervisor Call (SVC) and Pendable Service Call (PendSV)
  - Allow context switching and other essential embedded OS operations
- Unprivileged execution level
  - Basic security model that restricts access rights of some application tasks
- Exclusive access
  - Exclusive load and store instructions are useful for semaphore and mutual exclusive (MUTEX) operations in the OS

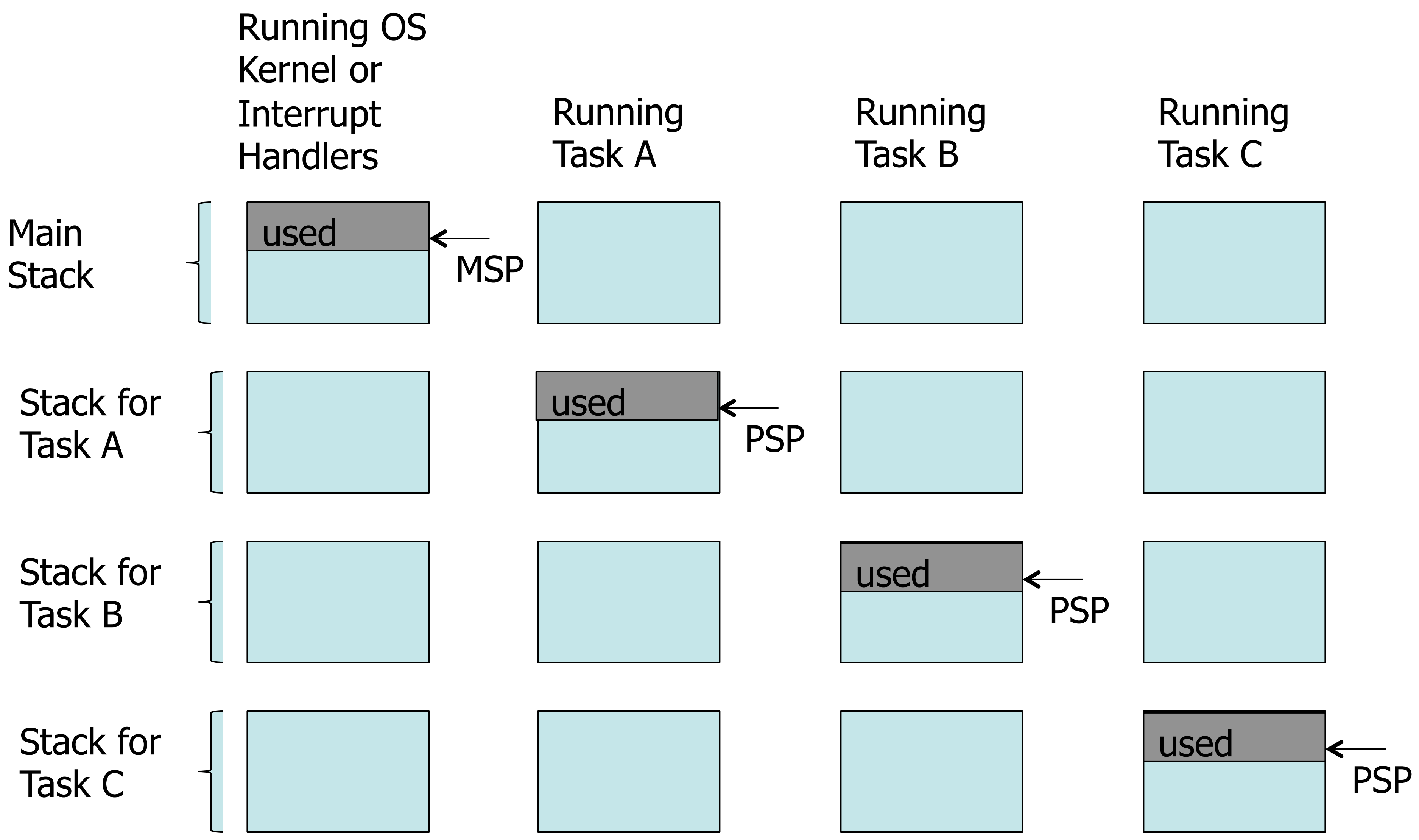
Shadowed Stack Pointer

- Main Stack Pointer (MSP)
  - Default stack pointer
  - Used in Thread mode when CONTROL bit[1] (SPSEL) is 0
  - Always used in Handler mode

- Processor Stack Pointer (PSP)
  - Used in Thread mode when CONTROL bit[1] (SPSEL) is 1

- In systems with an embedded OS or RTOS
  - Exception handlers (including OS kernel) use MSP
  - Application tasks use PSP
  - Each application task has its own stack space
  - Context switching in OS updates PSP each time context is switched

Shadowed Stack Pointer



Shadowed Stack Pointer

- If application task has a problem that leads to stack corruption, stack of OS and other tasks likely remains intact
- Stack space for each task only needs to cover maximum stack usage plus one level of stack frame
- OS can use Memory Protection Unit (MPU) to define the stack region which an application task can use
  - If task has a stack overflow problem, MPU can trigger a MemManage fault exception
  - Prevent task from overwriting memory regions outside its allocated stack space

Context Switching

- OS needs to switch between different tasks
- Context switching usually carried out in the PendSV exception handler
- Can be triggered by a periodic SysTick exception
- Inside context switching operation
  - Save current status of registers in the current task
  - Save the current PSP value
  - Set the PSP value to the last SP value for the next task
  - Restore the last values for the next task
  - Use exception return to switch to the task

- Context switching is carried out in PendSV
  - Typically at lowest priority level
  - Prevents context switching from happening in the middle of an interrupt handler

Supervisor Calls: SVC & PendSV

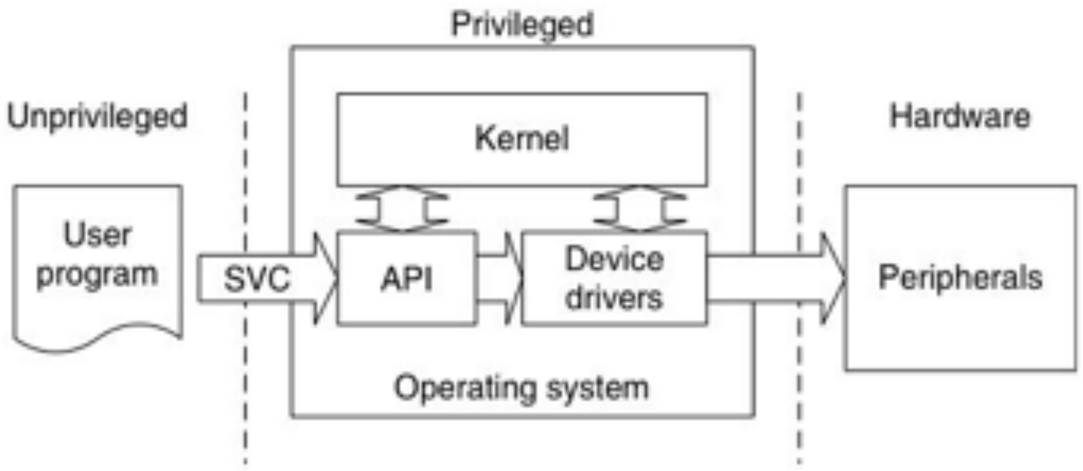
- SVC (Supervisor Call) and PendSV (Pendable Service Call)
  - Important for OS designs

- SVC Instruction
  - Keil/ARM: `__svc`
  - Portable; hardware abstraction
  - Can write to NVIC using a software trigger interrupt register, but several instructions might execute while the interrupt is pending
  - With __svc, the SVC handler executes immediately (except when another higher priority exception arrives)

- Can be used as API to allow application tasks to access system resources

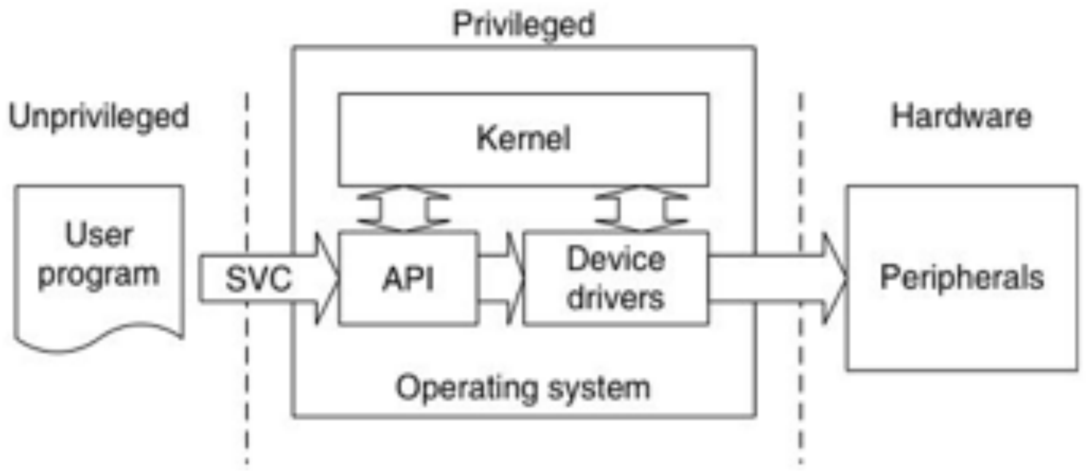
Supervisor Calls: SVC

- Systems with high reliability requirements
  - Some hardware set up to be privileged access only
  - Embedded system is more robust and secure



Supervisor Calls: SVC

- Application tasks
  - run in unprivileged access level
  - can only access protected hardware resources via services from OS
  - Cannot gain unauthorized access to critical hardware
  - Do not need to know programming details of underlying hardware
  - Can be developed independently of OS (don't need to know exact address of OS service functions – only the SVC service number and parameters)
  - Hardware level programming handled by device drivers



SVC Handling

- SVC instruction (`__svc`)
  - Needs immediate value (`SVC #0x3; Call SVC function 3`)
  - SVC exception handler extracts parameter and knows which action to perform
- Procedure
  - Determine if the calling program was using main stack or process stack?
  - Read the program counter value from appropriate stack
  - Read instruction from that address (masking out unnecessary bits)

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

Source : J. Yiu, “The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors”

SVC Handling

- In C, we break it into two parts
- Can’t check the value of LR (EXC_RETURN) in C
- Use assembly inline (`__asm`)

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

Source : J. Yiu, “The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors”

SVC Handling

```
void SVC_Handler_C(unsigned int * svc_args)
{
  uint8_t svc_number;
  unint32_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 elements)
  svc_args[0] = stacked_r0 + stacked_r1;
  return;
}
```

Source : J. Yiu, "The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors"

SVC Handling

- Passing the address of the stack frame allows the C handler to extract any information it needs
- Essential if you want to pass parameters to an SVC service and get a return value

- A higher priority interrupt could be executed first and change the values of R0, R1, etc.
- Using the stack frame ensures your SVC handler gets the correct input parameters

PendSV

- Pended Service Call (exception type 14)
  - Programmable priority level
  - Triggered by writing to Interrupt Control and Status Register (ICSR)
  - Unlike SVC, it is not precise. Pending status can be set in a higher priority exception handler
- Execution of OS kernel (and context switching) triggered by
  - SVC call from an application task (e.g. task is stalled because it is waiting for data or an event)
  - Periodic SysTick exception
- PendSV is lowest priority exception
  - Context switching delayed until all other IRQ handlers have finished
  - OS can set pending status of PendSV & carry out context switching in PendSV exception handler

Exclusive Access

- Multi-tasking system: tasks need to share limited resources
  - For example, one console output
- Tasks need to “lock” a resource and then “free” it after use
  - Usually based on software variables
  - If lock variable is set, other tasks can see that the resource is locked

- Lock variable is called a semaphore
  - If only one resource is available, also called Mutual Exclusive (MUTEX)
  - In general, semaphores can support multiple tokens
    - e.g. one for each channel of a communication stack
    - Semaphore implemented as a token counter
    - Each task decrements the semaphore when it needs the resource

Exclusive Access (2)

- Decrement of counter is not atomic
  - One instruction to read variable
  - One instruction to decrement it
  - One instruction to write back to memory

- Context switching may occur between read and write
  - Simplest approach: disable context switching when handling semaphores
    - Can increase latency and only works for single processor designs
    - Multi-processor: tasks on different processors can try to decrement semaphore variable at same time

Exclusive Access: Local Monitor

- Cortex-M4 supports feature called exclusive access
  - Semaphores are read and written using exclusive load and exclusive store
  - If during store, access cannot be guaranteed to be exclusive, exclusive store fails
- Processor has small hardware unit called the local monitor
  - Normally in Open Access state
  - Exclusive load: switches to Exclusive Access state
  - Exclusive store: only executes if local monitor in Exclusive Access state
  - STREX fails if
    - CLREX has been executed (switching local monitor back to Open)
    - Context switch has occurred (interrupt)
    - No LDREX was executed earlier
    - External hardware returns an exclusive fail status
  - Multiprocessor systems need a global exclusive access monitor

Summary

- Exception handling
  - Exception return code is inserted into link register
    - Provides key information about the exception
  - Handling is built around the stack frame

- Cortex M4 and the Discovery board provide multiple resources to support operating system functionality
  - Shadowed stack: different stacks for different tasks
  - Supervisor calls: OS provides hardware access services
  - Pending service call: context switching
  - Exclusive access: local monitor and exclusive load/store