Task Switch in TachyOS (ARMv7m)

> Context switch exploits exception exit / entry behavior on target architecture.

Start initial task

1. exception entry stack is prepared for new task

prepared exception entry

Registers	Initial Value	Stack Top Pointer (Task Context)
R4	0	*
R5 ~ R11	0	
xPSR	THUMB (1<<24)	
Return Address	address of runtime init routine for task	
R0 - R4, LR	DONT CARE	

2. enter handler mode (or supervisor mode) using `SVC`
3. set `PSP` with stack top address from the initial task
4. pop `PSP` by amount of saved context `R4-R11` (size of 32 bytes)

> Note : task context, which is not actually used, is inserted into the stack only for maximization of code reuse. By doing this, we don't have to differentiate the way of stack preparation between `start_task()` and `switch_task()`

Switching Task

1. When exception has occurred, the hardware pushes exception entry context`R0 - R3, R12, R14(LR), Return Address, xPSR` into process stack `PSP` ahead of jump to any exception handler

2. if switch is required before returning from exception handler, additional exception entry context is inserted into process stack (just beneath original exception entry frame) to mimick hardware to jump switch function on exception return.

   1. decrement `PSP` with size of exception entry stack
   2. fill function parameters for switch function (which is standard C function) into `R0 - R4` within the prepared exception entry stack
   3. set return address for the exception entry stack as a function entry of switch function (standard C function)
   4. set xPSR as thumb mode (assuming ARM interworking not used here)

3. Return from exception

4. handler will restore callee-saved registers

> Note : core registers R4-R11 will be preserved by callee, that mean the exception handler implemented as pure C function which complies AAPCS should restore the values of callee-saved-registers ahead of return

5. then H/W will jump to the switch function with popping up the exception entry context from process stack PSP, which is pretty much same to calling general C function with argugment
6. switch function push callee saved registers into stack PSP

Switch function (Cortex-M4)

```
void tch_port_switch(uwaddr_t nth,uwaddr_t cth)
{
    asm volatile(
#if FEATURE_FLOAT > 0
            "vpush {s16-s31}\n"
#endif
            "push {r4-r11}\n"        ///< save callee-saved registers
            "str sp,[%0]\n"          ///< save stack pointer within TCB

            "ldr sp,[%1]\n"          ///< load stack pointer value from TCB
of next task
            "pop {r4-r11}\n"         ///< restore callee-saved registers
#if FEATURE_FLOAT > 0
            "vpop {s16-s31}\n"
#endif
            "ldr r0,=%2\n"           ///< set svc handler argument
            "svc #0" : : "r"(&((tch_thread_kheader*) cth)->ctx),"r"(&
((tch_thread_kheader*) nth)->ctx),"i"(SV_EXIT_FROM_SWITCH) :
"r4","r5","r6","r8","r9","r10", "r11", "lr" );
}
```

Exception entry stack frame on switch

Fields	Role	Origin
R0 ~ R3	function arguments	fake exception entry
R4 ~ R11	local variables	from original exception entry (callee saved registers)
R12	Intra-Procedure-call scratch	from fake exception entry
R14	Link Register	from fake exception entry
Return	return address	from fake exception entry
xPSR	instruction mode for return address (ARM / THUMB)	from fake exception entry

Task Switch in FreeRTOS

Start Initial Task

1. create idle task with lowest priority level
2. create timer task
3. setup H/W specific configuration (H/W floating point / interrupt / priority / etc.)
4. start initial task

Simplified callgraph of scheduler initialization

```
vTaskStartScheduler()
-> vApplicationGetIdleTaskMemory()
-> xTaskCreateStatic(prvIdleTask)
    ...
    -> pxPortInitialiseStack()
    ...
-> xTimerCreateTimerTask()
    ...
    -> pxPortInitialiseStack()
    ...
-> xPortStartScheduler()
    -> vPortSetupTimerInterrupt()
    -> vPortEnableVFP()
    -> prvPortStartFirstTask()
        -> SVCall()
```

pxPortInitialiseStack()

> create exception entry stack for redirection of exception return (to task routine), implementation is basically deteremined by hardware behavior and context switcing mechanism

Stack Layout

Field Name	Value	Note
xPSR	THUMB	exception entry context
Return Address	task routine address	exception entry context
LR	task return address (@prvTaskExitError)	exception entry context
R1 - R3, R12	0	
R0	task parameter	

Field Name	Value	Note
LR	0xfffffffd	exception entry will return to thread mode with PSP
R4 - R11	0	

prvPortStartFirstTask()

> starting initial task for scheduler. this function call never return to its original caller, because exception entry is be swapped to start task.

Code Snippet

```
    __asm volatile(
                    " ldr r0, =0xE000ED08   \n" /* Use the NVIC offset
register to locate the stack. */
                    " ldr r0, [r0]          \n"
                    " ldr r0, [r0]          \n"
                    " msr msp, r0           \n" /* Set the msp back to the
start of the stack. */
                    " mov r0, #0            \n" /* Clear the bit that
indicates the FPU is in use, see comment above. */
                    " msr control, r0       \n"
                    " cpsie i               \n" /* Globally enable
interrupts. */
                    " cpsie f               \n"
                    " dsb                   \n"
                    " isb                   \n"
                    " svc 0                 \n" /* System call to start
first task. */
                    " nop                   \n"
                );
```

1. before start first task in thread mode, roll up stack pointer to the start of stack address
2. all bits are cleared in CONTROL register, which put execution context as below
   - SP = SP_Main
   - FP extension not active
   - privileged access allowed
3. enable interrupt and fault exception
4. enter supervisor call to start task

vPortSVCHandler() (naked function)

```
    __asm volatile (
                    "   ldr r3, pxCurrentTCBConst2      \n" /* Restore the
context. */
                    "   ldr r1, [r3]                    \n" /* Use
pxCurrentTCBConst to get the pxCurrentTCB address. */
```

```
                    "   ldr r0, [r1]                    \n" /* The first
item in pxCurrentTCB is the task top of stack. */
                    "   ldmia r0!, {r4-r11, r14}        \n" /* Pop the
registers that are not automatically saved on exception entry and the
critical nesting count. */
                    "   msr psp, r0                     \n" /* Restore the
task stack pointer. */
                    "   isb                             \n"
                    "   mov r0, #0                      \n"
                    "   msr basepri, r0                 \n"
                    "   bx r14                          \n"
                    "                                   \n"
                    "   .align 4                        \n"
                    "pxCurrentTCBConst2: .word pxCurrentTCB             \n"
                );
```

1. load TCB of first task and pop context information `R4-R11, R14` from the prepared stack frame of the task

> Note : R14 has crucial role in exception return. SPSEL bit is wrtten by hardware based on the value in R14 (LR). this trick is used because SPSEL is not allowed to be written in handler mode (see EXC_RETURN table)

2. set `PSP` value with stack top of the task
3. returning from exception will direct execution location into task routine, which is prepared by `pxPortInitialiseStack()` call

EXC_RETURN table (No FP extention)

EXC_RETURN	Return-Mode	Return-Stack
0xFFFFFFF1	Handler Mode	MSP
0xFFFFFFF9	Thread Mode	MSP
0xFFFFFFFD	Thread Mode	PSP

Switching Task

> FreeRTOS uses PendSV call for context switch. the core registers saved by callee are according to AAPCS. that mean any restored context (or context to be saved) could be corrupted epilogue (or prologue) on function exit (or entry).

1. so switch function should be written in pure assembly or at least naked function.
2. switch function should not preempt another interrupt handler because it makes to keep core registers from corruption very difficult. (even when all the interrupt handlers are written as naked function.)

```
#define portYIELD()
\
{
\
    /* Set a PendSV to request a context switch. */
\
    portNVIC_INT_CTRL_REG = portNVIC_PENDSVSET_BIT;
\

\
    /* Barriers are normally not required but do ensure the code is
completely  \
    within the specified behaviour for the architecture. */
\
    __asm volatile( "dsb" ::: "memory" );
\
    __asm volatile( "isb" );
\
}

#define portEND_SWITCHING_ISR( xSwitchRequired ) if( xSwitchRequired !=
pdFALSE ) portYIELD()
#define portYIELD_FROM_ISR( x ) portEND_SWITCHING_ISR( x )
```