C S4680-101_EMBEDDED SYSTEMS (SPRING 2021)

Grad Project (ACX 2): x_yield - Thread Rescheduling

Introduction

For this assignment you will write the kernel initialization function **x_yield** whose prototype is:

```
void x_yield(void);
```

A thread may be in one of three states: RUNNING, READY or BLOCKED. There is only one thread in the RUNNING state, and a thread is considered READY if it is not blocked by one of the three blocking conditions: disabled, suspended or delayed. The **x_yield** function is called by a RUNNING thread to yield the processor to the next thread that is in the READY state. If no other threads are in the READY state, then control is returned to the calling thread. The **x_yield** function carries out the rescheduling function of ACX, and uses a simple **round-robin** scheduling policy.

Round-robin scheduling simply uses a thread's ID to check thread status for a READY condition, beginning with the next thread in sequential order after the calling thread, where the sequence is carried out modulo-N and N is the number of threads. For example, if there N = 8 (8 threads defined), and a thread with ID = 3 calls **x_yield**, then threads 4, 5, 6, 7, 0, 1, and 2 would be checked for READY, in that order. The first READY thread would be given control. If no other threads are READY, then control would be returned to thread 3.

The **x_yield** function can be called from any function called by a thread, **however:**

- **if function nesting is too deep, or too much local variable memory is allocated, it is possible to overflow the thread's stack area**. The x_yield function will check for stack overflow by checking the value of a "canary" placed at the end of a thread's stack region.
- **x_yield** must not be called from within an ATOMIC block (that is, within a block in which interrupts are disabled). x_yield performs atomic accesses to some system state and re-enabled interrupts during its execution. This means that the block will no longer provide atomic access for the thread and the resulting behavior may not be as intended.

The **x_yield** function may not be called from an interrupt handler.

Requirements for the x_yield Function

In general, the **x_yield** function must:

- Save the state of the calling thread on its stack (what must be saved?)
- Save the calling thread's stack pointer in the stack control array
- Check for stack overflows (canary still intact?)
- Determine next thread to place into execution (scheduling)
- Retrieve the next thread's stack pointer and update the SP register
- Restore the next thread's state from its stack
- return to the next thread (picking up immediately after where it last called x_yield or x_delay).

Details

Part 1

- Write the **x_yield** function in assembly language in the file "acx_asm.s".
- Your function should save and restore only the "callee-save" registers. When entering the function you should push them onto the calling thread's stack. When exiting the function you should pop them from the stack in reverse order (this thread will often be a different thread's stack). As a first step in development, write a 'main' function that calls your x_yield function where it only implements the register save and restore parts of the function. This includes using the current thread ID stored in x_thread_id to save/restore the value of SP from the correct place in your stack control array. Verify that you can call it and return back to the calling thread correctly using the simulator, and

that register values have been preserved. Also verify that you correctly save and restore SP to/from the appropriate stack control structure based on the thread ID (which will be zero in this case, but will not always be zero). IMPORTANT NOTE: Whenever you read or write SP you will need to do it atomically--that is, the access must not be interrupted since it involves two register locations. You may use the cli (disable global interrupts) and sei (enable global interrupts) instructions around accesses to any registers or variables that must be atomic. We won't require saving the global interrupt status prior to disabling interrupts because we require that interrupts be enabled when any thread rescheduling function is called (x_yield, x_delay).

- Determine how you will check for stack overflow. The canary values should be located in that last available stack memory address for each thread. This address should be at a fixed offset relative to the stack base address of another thread. For example if stacks are allocated to threads in ascending memory address order with ascending thread IDs, then "thread7.canary" is located at "thread6.spbase + 1", "thread6.canary" would be located at "thread5.spbase + 1", etc. Thread 0 canary would then be located at the first byte of the stack memory area. If a thread's canary does not match its initialized value you should enter an infinite error loop that asserts a special blinking signal. **How could you check to make sure your canary test works? Do this before continuing.**

Part 2

- Below the **x_schedule** entry point you will write the scheduling part of **x_yield**. For this part of the function you may use any registers in your code (the compiler makes sure that caller-save registers are saved, and the "save" section of the x_yield code should save all of the callee-save registers). The **x_schedule** label is used as a function entry-point (with no prior saving of callee-save registers) by the **x_new** ACX function. This will be explained later. The AVR C ABI states that R1 must be 0 before returning to another thread and that R0 is a scratch register and may be left in any state.
- Scheduling will be "round-robin", starting with the next thread ID (mod NUM_THREADS) after the one that called **x_yield**. Scheduling is done by:
  - bitwise OR the status variables x_disable_status, x_suspend_status and x_delay_status
  - Scan the result for the next bit whose value is zero, starting with the bit number that is one after that corresponding to the current thread ID (mod NUM_THREADS). You will have to scan the bits in a circular fashion and will find it handy to use the x_thread_mask.
    Alternatively you may use a **bit2mask** function (you'll have to write it) that returns a bit mask corresponding to a particular ID (between 0 and 7 inclusive). This function may be implemented in C or assembly code (though assembly can be more efficient--e.g. you may put a lookup table in code memory and use the **lpm** instruction to access it).
  - Maintain the the value of the current bit being checked as the thread ID of the (possible) next thread, and the thread mask as the corresponding bit mask.
- When the next READY thread is found, use its ID to lookup its saved SP value.
  - Save the next thread ID in x_thread_id, and the mask in x_thread_mask
  - initialize SP with the next thread's SP
  - proceed to the **x_restore** assembly code label where registers are popped and the return instruction is executed to go to the next thread.
- If there are no READY threads, then the kernel will (for now) keep scanning. An interrupt handler may change the state of a thread, thereby enabling it. Later we will use the condition of having no READY threads to put the chip into a SLEEP state to save power. In this state an interrupt can be used to "jolt" it out of SLEEP and rescheduling can continue.

Simple Test

A simple test of the **x_yield** function uses the following while loop at the end of your main method:

```
while(1){
    x_yield();
}
```

This should cause the kernel to save and restore T0's state (since it is the only thread that is READY). If this works, then you (might) have it working correctly. **Use the simulator to verify saving, scheduling and restoring thread state.**

Submission

Upload your **acx_asm.S** file and **demonstrate that your program works correctly in class**.

Submission status

Submission status	Submitted for grading
Grading status	Not graded
Due date	Tuesday, April 13, 2021, 11:55 PM
Time remaining	Assignment was submitted 6 days 18 hours late
Last modified	Tuesday, April 20, 2021, 6:33 PM
File submissions	ACX02.zip April 15 2021, 2:05 PM
Submission comments	Comments (0)

◄ Screencast: ACX x_init function

Jump to...

Screencast: ACX x_yield function ►