C S4680-101_EMBEDDED SYSTEMS (SPRING 2021)

Grad Project (ACX 1): ACX Data Structures and Initialization

Background

A a general purpose operating system (OS) is essentially a program that provides **services** to other programs (or "processes" when in their running state) through resource management. By centralizing commonly required services an operating system provides an abstraction of the underlying hardware resources, provides for software re-use, and improves reliability and maintainability. Resources managed usually include:

- Processor (scheduling)
- Memory (stack, heap management)
- I/O devices (including storage)
- System utilities (e.g. inter-process and network communication)

There are many kinds of operating systems. Operating systems for general-purpose computer systems are often designed to support:

- multiple users
- multiple processes per user
- multiple threads ("light-weight" processes)

Sometimes these operating systems may support multiple processors as well. General purpose operating systems provide a rich set of services to user and kernel processes.

Operating systems designed for special-purpose computer systems (including embedded systems) may have different goals. For example, these might include:

- support for real-time (deterministic) processes/threads
- support for safety-critical systems (e.g. redundancy, fault-tolerance/recovery)
- support for very low-power systems
- support for very low-cost systems (small memories, inexpensive processors, etc.)

Once the goals of an operating system are established, its capabilities are reflected in its underlying data structures. For example, a general purpose OS may use data structures that provide maximum dynamic flexibility in process management. Lists, queues, and trees provide flexibility and performance at the cost of memory use. Virtual memory provides a very useful abstraction for programs while incurring a significant hardware and system software cost.

Embedded systems, on the other hand, may use statically defined quantities and simple data structures to reduce memory cost and/or to provide timing determinism. General purpose operating systems are assumed to be "preemptive" in their operation, meaning that processes and/or threads may be preempted at any point in their execution by the OS in order to place a higher priority process into execution. This is called rescheduling. Sometimes the term "operating system" is assumed to imply "preemptive scheduling". It is possible for a (simple) operating system to be non-preemptive--meaning that processes must cooperate in sharing of the CPU by invoking some OS function during their execution and avoiding busy-wait loops. This kind of "operating system" is sometimes called an "executive". We will be designing and implementing a simple multi-threading executive for our next project.

ACX - A Cooperative eXecutive

ACX is a very simple multi-threading cooperative executive. It supports the following:

- Up to 8 threads (at least 1)
- Independent thread stacks, allocated statically (that is, at compile-time rather than run-time)
- Stack overflow detection
- Three bits of thread control per thread, residing in three 8-bit variables (**disable**, **suspend**, **delay**)
- 16-bit delay counter per thread

- Global system tick counter (32 bit)
- System timer with 1 msec resolution

Kernel State

In order for ACX to support threads and thread scheduling it needs some basic data structures. These will include variables for:

- thread stacks
- thread stack control (saved SP and base SP values)
- thread delay counters
- thread state variables (disable, suspend, delay)
- current thread ID
- global system tick counter

There will also be a couple of useful typedefs and #defines.
The **ACX API** provides brief descriptions of ACX functions and data.

Problem

For this assignment you will:

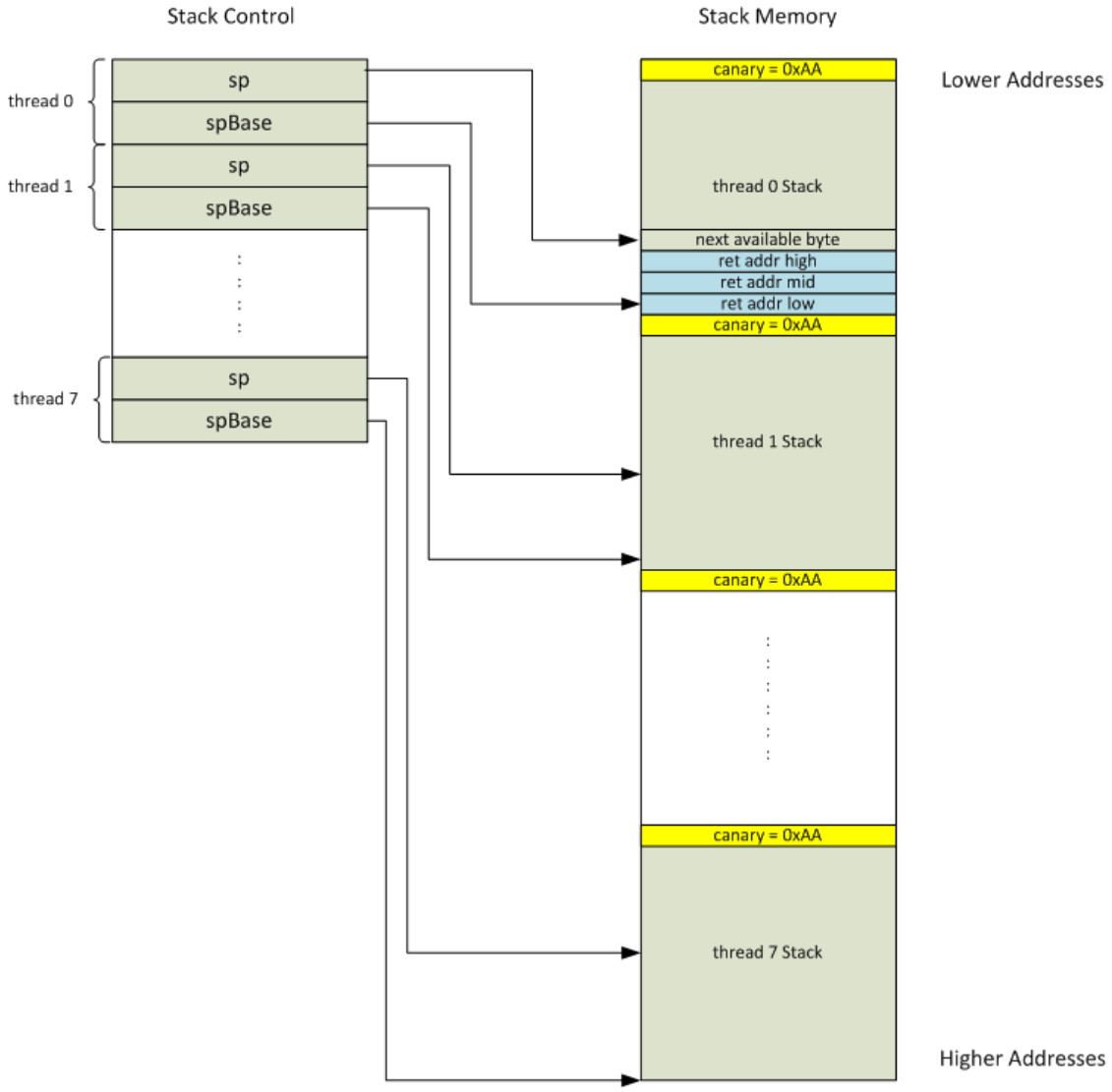
1. Define ACX constants, types and data structures
2. Implement the **x_init** function which:

- initializes thread stacks and stack control structures, and initializes "canary" values for stack overflow detection
- initializes kernel status variables
- initializes the system timer and thread delay counters. You will use 8-bit Timer0 in CTC mode, where TOP is set for a count representing 1 msec. You will provide a Timer0 ISR (for compare match) in **acx.c** that will (for now) do nothing.

Details

Download the starting files from here: ACX Start Files

- The **acx.h** file is provided. It defines constants, a few types used for stack and thread control and function prototypes.
- Thread stack sizes are defined statically in **acx.h**.
- In file **acx.c** declare and initialize all kernel state variables that can be initialized statically (outside a function). This includes status bytes, delay counters (an array of uint16_t), and global timer/counter variable, current thread ID (x_thread_id):
  - disable status = 0xFE (leave Thread 0 enabled)
  - suspend status - 0x00
  - delay status = 0x00
  - delay counters (an array of uint16_t) and global timer are cleared to zero
  - x_thread_id = 0 (current thread)
  - x_thread_mask = 0x01 (bit 0 set corresponds to thread 0)
- In file **acx.c** stub functions for each ACX system call are provided (except **x_yield** and **x_schedule** which will be implemented in **acx_asm.S**).
- Write the **x_init** function to initialize kernel data structures and place canary values in thread stacks. The canary values should be placed at the very last byte of each thread's stack (lowest addressable byte of the stack). Here is a diagram to show the relative locations of the stacks:



Note that the canary values are placed at the last available stack location (lowest address) for each thread's stack. It will be simple to check these using the stack base pointers in the stack control structures.

- Before returning, the the **x_init** function should copy the call stack (return address, etc.) into the Thread 0 stack area, and set SP to point to this stack area so that when **x_init** returns, the calling function will become Thread 0.

To test your code you may run it in the simulator and inspect memory locations for correct values.

Submission

Upload a zip file **ACX.zip** containing your **acx.h** and **acx.c** and **acx_asm.S** files

Submission status

Submission status	Submitted for grading
Grading status	Not graded
Due date	Thursday, April 8, 2021, 11:55 PM
Time remaining	Assignment was submitted 15 mins 26 secs early

Last modified	Thursday, April 8, 2021, 11:39 PM
File submissions	ACX01.zip April 8 2021, 11:39 PM
Submission comments	Comments (0)

Edit submission | Remove submission

You can still make changes to your submission.

◀ Embedded Systems Review Quiz

Jump to...

Screencast: Implementing ACX - Introduction ▶