# C S4680-101_EMBEDDED SYSTEMS (SPRING 2021)



## Grad Project (ACX 3): x_new - Create a new thread

### Introduction

Once the kernel is initialized we need a way to associate functions with threads. The **x_new** function is used to assign a code (function) pointer to a particular thread ID. After the call to **x_init** from the main function there is only one thread in operation--Thread 0--running the main function code. The prototype for the basic **x_new** function is:

```
void x_new(byte threadID, PTHREAD newthread, byte isEnabled)
```

- **threadID** is the ID of the thread to which '**newthread**' will be assigned (0 through 7 : these should be #defined in acx.h)
- **newthread** is a function pointer that takes no parameters and returns nothing. We may later change this to accept parameters.
- **PTHREAD** that may be defined as:

```
typedef (void *PTHREAD)(void);
```

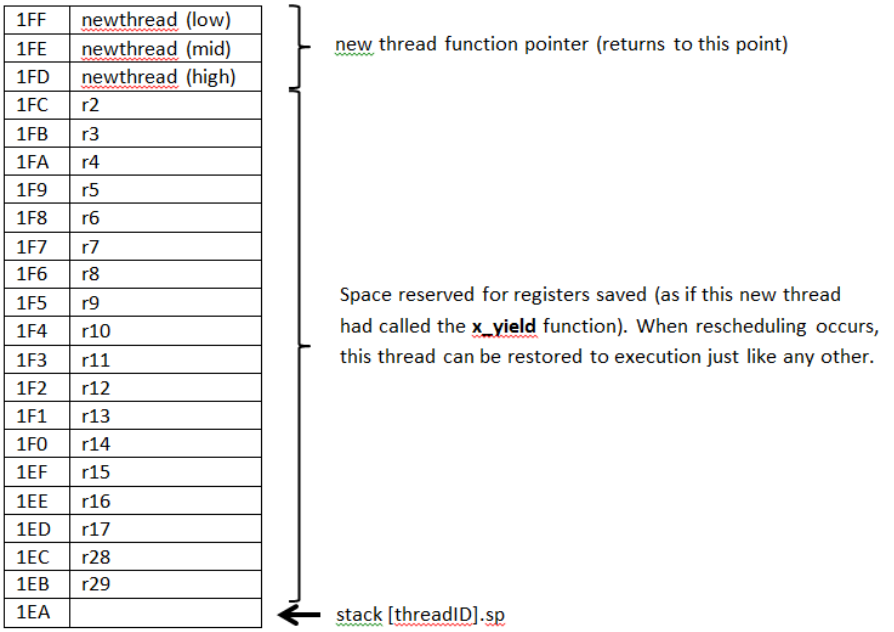
- **isEnabled** is the initial status of the thread--1 means enabled, 0 means disabled.

### Function Description

When called, **x_new** must initialize the stack of the specified thread by copying the function pointer onto its stack (as if it were a return address). The thread's stack pointer must then be updated to a value **that allows space for all the registers that are normally saved when x_yield is called** (**x_yield** is the primary rescheduling function of the kernel). For example, if the value of the new thread's sp is initialized to 0x1FF, then the **newthread** address is copied to 1FF (low byte), 1FE (mid byte), and 1FD(high byte) . Then the thread's saved stack pointer (e.g., stack[Tx_ID].pstack  where x is 0 through 7) is decremented by:

- 3 (for return address), plus
- 18 (for callee-saved registers)

This would leave pstack for this thread with a value of 0x1FF-0x15 = 0x1EA.
In visual terms, the stack would look like the following:




## Calling x_new() to Replace the Current Thread

If the **x_new** function is called with a thread ID that is not the ID of the running (calling) thread, then, after carrying out its function of setting up the specified thread's stack, **x_new** simply returns to the calling thread. The newly initialized thread will be scheduled to run when the next rescheduling call (**x_new** or **x_delay**) is made, and it the next READY thread.

If the **x_new** function is called with the calling thread's own ID, then the calling thread is replaced by the new thread. In this case, the **x_new** function does not return to the caller, but instead must jump to the rescheduling part of the **x_yield** function (skipping the register save part). This is the **x_schedule** entry point ) **which must be made global using the .global assembler directive**. This allows the scheduler to find the next READY thread, and if the new thread is the next ready thread, it will be restored to execution in the usual way. To implement this you will simply call **x_schedule** as if it were a regular C function (when it is in fact just an alternate entry point into the **x_yield** function).

## Writing a Function to Operate as a Thread

A function that operates as a thread will typically have the following form:

```
void threadFunction(void) {
   init_thread();        // any needed thread initialization
   while(1){             // do-forever
      do_thread_stuff(); // must call x_yield or x_delay somewhere to allow other threads to run
   }
}
```

The thread is coded as an infinite loop with at least one kernel rescheduling call that is invoked within the loop. This allows other threads to execute. If a rescheduling call is not made, this is the only thread that will run.

**Note:** The **x_delay** function provides a way to delay a thread by a specified number of "system ticks" (tick interval depends on TIMER0 initialization--chosen based on system requirements) while allowing other threads to run.

## Testing

Create a project TestACX that contains the following main.c function:

```
void testThread(void);
int main(void)
{
   volatile int j = 0;
   x_init();
   x_new(1, testThread, true);  // create thread, ID=1
// x_new(0, testThread, true);  // replace current thread
   while(1){
      j++;
      x_yield();
   }
}
//------------------------
// A test thread
//------------------------
void testThread(void)
{
   volatile int i = 0;
   while(1){
      i++;
      x_yield();
   }
}
```

Your first task is to make sure that x_new sets up the stack for thread ID 1 correctly. This can be done with the simulator.

When you run this code in the simulator, each call to x_yield should cause the kernel to switch to the other thread, picking up just after its (previous) call to x_yield. Verify that this is indeed what happens. You should also verify that the local variables are correctly incrementing starting with 0.

**After you have verified correct switching between the "main" thread and "testThread", uncomment the line that creates (and replaces) thread 0**. When this is done, execution should never reach the while loop in main, but instead two threads will be running, each with its own local variable 'i' but executing the same code. You can verify that you are correctly switching between threads by checking to make sure the thread ID is changing between 0 and 1 for each call to **x_yield**.

## Submission

Zip your project and submit in a file called acx.zip.

## Submission status

| | |
|---|---|
| **Submission status** | Submitted for grading |
| **Grading status** | Not graded |
| **Due date** | Saturday, April 17, 2021, 11:55 PM |
| **Time remaining** | Assignment was submitted 2 days 18 hours late |
| **Last modified** | Tuesday, April 20, 2021, 6:33 PM |
| **File submissions** | ACX03.zip April 20 2021, 12:03 PM |
| **Submission comments** | Comments (0) |

◄ Screencast: ACX x_yield function

Jump to...

Grad Project (ACX 4): x_delay - Delay a thread by specified number of "system ticks" ►