Lecture 13 — Thread Implementation

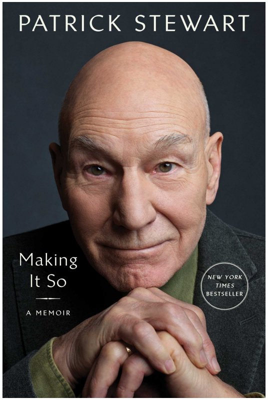
Jeff Zarnett & Mike Cooper-Stachowsky
jzarnett@uwaterloo.ca, mstachowsky@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

April 7, 2024

Make it So

With the thread theory behind us, now we need to actually work with some.



The model we'll use is a bit simpler than the UNIX thread (but we'll get there).

How To Get Started

A thread runs code and has some function that it starts with…
    Like `main()` in C, but it can be named anything.

There must be a way of keeping track of where we are in the execution.

We also need a stack for each thread (so a stack pointer).

Starting a Thread

Thread functions all have the same prototype:

```
void * function_name( void * argument ) {
  /* Interpret the argument(s) as needed here */

  /* Many threads never return, so they just loop forever as needed */
}
```

Why `void *` for input and output?

Anything Goes

The OS designer can't know what input and output an arbitrary thread will have.

So let the programmer choose! `void *` means it can be anything.

I’ll Show Myself Out

Threads rarely exit in a real-time operating system.
Generally it’s a continuous-operation system doing specific things.

You can make the return type `void` in your OS, but it’s not 100% correct.

More correct: exit from a thread is a system call, and tells the OS this thread cannot run ever again.

Starting the New Thread

If we’ve defined the starting function, can we just... call it?

No; that would run it in the *current* thread.

We need to run it as a new thread instead.

It’s like Delegation

HR Team: Asks to attend training on “Delegation Skills”
Me: Delegates the training to a subordinate
Outstanding move

In addition to the code to run, we need a new stack and the thread context.

Context

Context is the data a thread needs while running.

Obvious example: keep track of progress in the code!

But also: register data and stack pointer.

In a bigger OS: maybe permissions, open files, etc.

Own Stack

Remember: the stack is local storage for each thread...
  And also the execution trace to how we got here.

If we had one shared stack for everyone, all threads write to it.

Overwrite each other's data, overlapping execution traces...

Where to Thread Stacks Come From?

So if a thread needs a stack, where does it come from?

The OS can just designate an area of memory as the stack for a given thread.

You could divide the OS stack into different pieces, one for each thread.

Stack Pool

If each thread gets a piece of the OS stack, can they overwrite each other if one of the stacks gets too big?

Yes (unless we have virtual memory or similar). So on the Cortex M... yes.

This sort of stack overflow can corrupt data, and can’t be prevented.
  We just say it’s the application programmer’s problem.

Minimal Thread Control Block

We covered the idea of the process control block fairly thoroughly.

A minimal thread control block (TCB) has:

- Identifier
- Instruction pointer
- Stack pointer

We may or may not have the starting point of the thread in there.

Context is Not Data

Context is not entirely the same as the data.

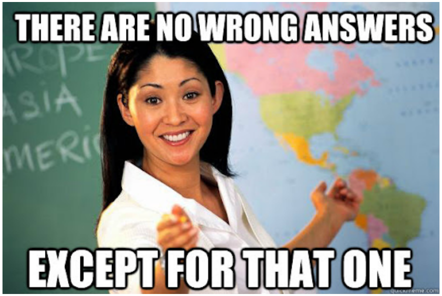
Context is how we know what we’re doing and where we are.

Data is things that a thread needs to do its job.

Corruption

If our context gets corrupted we'll crash (hard fault) very quickly.

If the data is corrupted, the output is wrong or garbage...
    We may get a hard fault, but not necessarily.



Take Turns

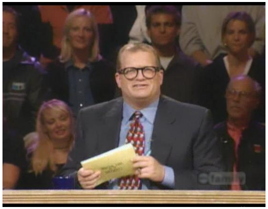
For threads to take turns on the CPU, it means we have to save and load their context information.

When thread A’s context is loaded, it can run.

When its turn is over, save the context, and load that of thread B.

… Now generalize that to arbitrarily many threads.

Whose Turn is it Anyway?

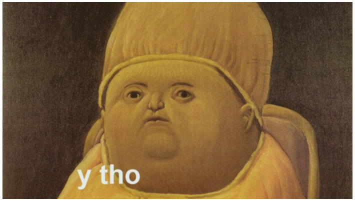


That’s a scheduling decision, so we’ll come back to that soon.

Setting up a Thread

1. Create the TCB.
2. Designate some area of memory to be the stack.
3. Set up the stack to contain the context of the thread.
4. The Kernel must:
   - Set xPSR to 1«24 (trust us)
   - Set PC to the thread's function pointer
   - Put input argument into `R0`
   - Put the stack 16x4 bytes down from xPSR
   - Store in order: `xPSR, PC, LR, R12, R3, R2, R1, R0, R11` through R4

That Seems Complicated



The goal is to make it seem to the chip like the thread has already run.

We are setting up the thread context on the stack just like it would be saved.

If you forgot something (likely) you'll get a hard fault.

What was I doing again?

We have been circling around the idea of the context switch a lot so far.

It may make conceptual sense but maybe unclear on how to implement it.
- This is what labs 2 and 3 are for.

Context switch is a very hard part of OS development...
- And any bug in it will be a huge problem!

Managing this in a RTOS

If there are many TCBs to manage, do they go in a list or array? Design decision!

The OS needs its own stack pointer (use `MSP`).

It also needs to keep track of how many threads exist, what’s running now, and how much stack space is available to allocate.

Basic Functionality

The RTOS needs to be able to:

- Initialize itself
- Create a new thread
- Start running a thread
- Switch between threads

Function Prototypes

```
kernelInit( ... ); // Initialize kernel data and chip settings

osCreateThread( ... ); // Make a new thread and register it with the kernel

kernelStart( ... ); // Run the first thread

osSched( ... ); // Run scheduler (choose new thread to run)

osYield( ... ); // Voluntarily give up the CPU, tell the scheduler to run
```

Initialization function: set kernel variables, set interrupt priority, create TCBs, create stack pointers.

Create Thread: get a new stack (fail if no space), initialize the thread stack, update OS variables to track new thread.

Abut Scheduling

Scheduling is so complex it gets its own section of the course (next one!).

Key question: which thread gets to run when?

In the labs, we use Round Robin until Lab 5.

Terminating a Thread

In our simple OS a thread that just... ends... will cause a hard fault.

It can be done! Remove the thread from scheduling; deallocate its resources.

That has to happen in a bigger OS (UNIX); otherwise we’re leaking memory.

To Boldly Go...

SET A COURSE THROUGH THAT SOLAR FLARE
I WANT TO MAKE A DRAMATIC OPENING
imgflip.com

Well then, let’s get on to what threads are like in UNIX.