Threads - Roll your own

How do we actually switch between execution contexts in UNIX?

Hannes Rabo
Katariina Martikainen
Charlotta Spik

KTH Kista, Sweden 27.11.2018

Thread or process?

- processes run in separate memory spaces
- threads share a memory space
- a thread is an entity within a process → process can have multiple threads
- a thread contains set of structures the system will use to save the thread ***context*** until it is re-scheduled

What is context?

- The information necessary for keeping track of where we are executing

- Thread context
    - thread's set of machine registers
    - the kernel stack (system calls etc here)
    - a thread environment block (holds info of the thread’s state)
    - a user stack in the address space of the thread's process (executable user code here)

ucontext_t - What do we find in it?

ucontext_t is a struct containing all the information about a context

Most importantly (for this assignment) contains

- **uc_link** - The link to the context to resume after termination
- **uc_stack** - Setting the stack connected to this context

Other things such as registers are also saved here

ucontext_t - How do we find it?

- **getcontext(&context_struct)**
    - Get a current context the program is running in and store it in the struct **context_struct**

- **makecontext(&context, func, param)**
    - Make a new context. Starts executing from the function specified in **func** with parameters specified in **param**

Making the switch

A call to **swapcontext(&context_one, &context_two)** will:

- Write current state (including stack, registers, instruction pointer etc.) to **context_one**
- Previously saved **context_two** becomes the current context

As the instruction pointer is overwritten, the execution will probably start at a new location!

To Yield or not to Yield

Two primary ways we could handle switching

- Threads yield to release the resources (explicit function call)
- Timeout that interrupts at any moment

You will explore this topic later!

Why would we use this?

1. Do not write your own thread libraries if you do not know what you are doing
2. The primary performance gain is from **avoiding kernel mode**
3. We are avoiding real parallelism with all connected problems

Conclusion: Combined with hardware threads, this is similar to how libraries are implemented

Exam question 1

1.6 context [2 points*]

By the help of the library procedure `getcontext()`, a process can save its own so called `context`. We could build a library that allowed us to create new executing threads and manually switch between these by calling a scheduler.

Why would we want to build such a library, are there any advantages? What would the disadvantages be?

Exam question 1

1.6 context [2 points*]

By the help of the library procedure `getcontext()`, a process can save its own so called `context`. We could build a library that allowed us to create new executing threads and manually switch between these by calling a scheduler.

Why would we want to build such a library, are there any advantages? What would the disadvantages be?

Answer: We would have a thread library were the treads are handled by the process itself. A switch between two threads would take less time compared to if we let the operating system do the switch (as is done in the pthread library). We would avoid many synchronization problems since we know that only one thread executes at any given moment.

A disadvantage would be that we would not be able to utilize a processor with several cores. We would also be completely blocked if a thread needs to do an I/O operation.