Context Switching

Jan Koniarik
433337@mail.muni.cz

Faculty of Informatics, Masaryk University

May 13, 2022

Goals

- Provide a good basic mental model of context switching.
- Show that the complexity in lies the specifics of the hardware platform.

The problem

In the simplest terms, a context switch is an act of:

1. Stopping execution of actual job
2. Storing the state of the job
3. Restoring the state of new job
4. Resuming execution of the new job

Motivation

- Context switching allows us to implement jobs as a reasonable segment of code that can be stopped at any point.
- The alternative is decomposing everything into tiny segments of code that are called sequentially - high cognitive load.

Assumptions

During this lecture, we assume single-core processors without a big operating system - an embedded device.
The knowledge is transferable. We want to keep the explanation simple.

Basic context switching

Basic context switching

To explain basic context switching, we need to:

1. Define a simplified model of a processor and necessary concepts.
2. Show how context switching works on that model.

Having the basic terms properly defined is necessary, as that greatly simplifies the explanation.

Model

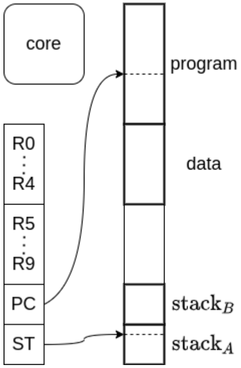
The model of processor is inspired by ARM platform and provides a specific simplified example. It is used only for the explanation of the basic context switching.

Processor

Processor consists of interconnected:

core	executes instruction
memory	stores code, stacks, and data
registers	local memory:

- R0..R9 general purpose
- PC instruction address
- ST stack address

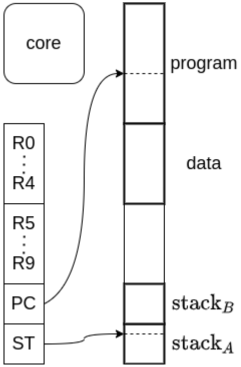


Program

We assume that the program has two forms:

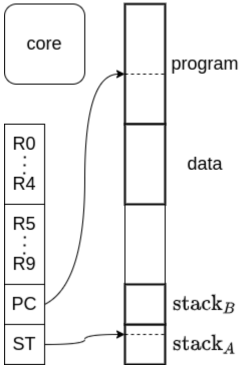
1. source code
2. compiled code

Register PC contains the address of the next instruction of the program that should be executed.



Stack

A part of memory where the program stores local data for functions. The processor stores an address at the end of a stack in ST register.



Function call

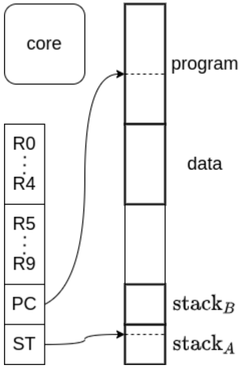
From the perspective of instructions, the call of function has a form of jump instruction - PC register is changed to the target address:

jump example:

```
bl 801440c <abort>
```

Key questions:

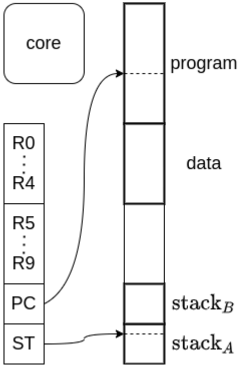
- What happens with the registers?
- How are the arguments passed?



Function call - registers

Let's define a convention about how registers should be handled between jumps.
When caller routine calls a callee routine:

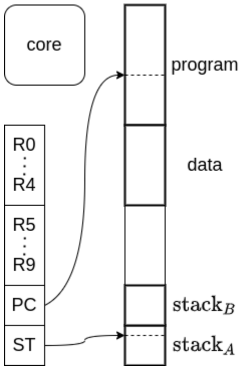
R0-R4	can be corrupted by the call
R5-R9	has to be preserved during the call



Function call - arguments

There are multiple ways for the code to pass data between caller and called:

- Data can be pushed on a stack.
- Data can be stored in R0-R4 registers.



Interrupts

An interrupt is an event that can happen at any time. It is serviced by the processor that executes a designated function - interrupt handler.

Interrupts

Once processor detects interrupt:

- P finishes active instruction
- P simulates function call behavior - R0-R4 registers are stored on the stack
- P executes interrupt handler
- H takes care of the rest of the registers if necessary (it acts as called function)
- H executes interrupt-related logic
- H clears it’s data from stack
- H returns from code
- P Restores last state on stack into R0-R4 registers

P - processor, H - handler

Interrupts

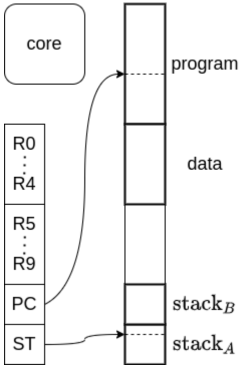
For our purposes, there are two important sources of interrupts:

Timer	peripheral that can cause events periodically or after a delay. The peripheral is driven by the same clock as the processor - cycle-level resolution.
Software	caused internally by the code - originates from within

Threads

Thread works in familiar manner. There are two key properties:

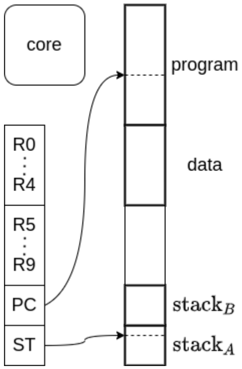
1. We work with a single processor - threads are sequentially interleaved.
2. Each thread requires a separate stack in the memory.



Context switching

Let A be an active thread and B thread that shall be used. A context switch is a process which:

1. Stops thread A
2. Stores the state of A: registers, stack address
3. Restores the state of B
4. Restores execution of B



Sources of context switching

We assume two sources of a requirement for context switch:

1. Explicit yield from thread
2. Initiated by scheduler (i.e. periodic timer)

Both can be handled in the same way - they raise a software interrupt used for a context switch.

How it works

Assume thread A is running and that software interrupt for context switch is raised:

- P stores registers R0-R4 at the end of stack_A
- P executes handler
- H stores registers R5-R9 at the end of stack_A
- H stores values of ST,PC for thread A
- H asks scheduler for next thread - B
- H restores values of ST,PC of thread B
- H restores registers R5-R9 from stack_B
- H returns
- P restores registers R0-R4 from stack_B
- P resumes execution of thread B

P - processor, H - handler

New thread

What if thread B is new and never executed a single instruction?
When a thread is created, RTOS executes a special procedure that prepares the stack in a way that:

- it looks like the thread was executed
- has no effect

Complexity

How fast is this?
The execution time of a context switch consists of three steps:

1. Store state of an active thread
2. Select and switch to a new thread
3. Restore the state of the new thread

Steps 1) and 3) can be fast as they boil down to a few instructions.
The complexity is mainly affected by step 2).

Complexity
Select and switch

The complexity of selection and switching depends on the exact algorithm used and the necessary steps of the switching process. For a simple scheduler with a low amount of tasks, that can be done in a few instructions. (This is purposefully vague, it could give hints to the project)

Remarks

Do not forget that we used a simplified processor. This is only a general approach. You always have to think about:

1. Each nuance of your hardware platform:
    - 1.1 What registers are handled by the interrupts
    - 1.2 How the hardware works with stack
    - 1.3 What forms the state of a thread (only registers? Something in memory?)
2. Each nuance of the rest of RTOS
    - What forms the state of a thread? For example, some kernels have a global symbol that always points to the control structure of actual thread.

ARM

ARM

1. ARM architecture is modern 32bit architecture used frequently in industry.
2. Manufacturer provides relevant documents, such as:
   - Cortex -M4 Devices Generic User Guide `https://developer.arm.com/documentation/dui0553/latest/`
   - Cortex-M4(F) Lazy Stacking and Context Switching - Application Note 298 `https://developer.arm.com/documentation/dai0298/a/`
3. There are also publicly available sources, such as article:
   - ARM Cortex-M RTOS Context Switching `https://interrupt.memfault.com/blog/cortex-m-rtos-context-switching`

Stacks

ARM has two registers for stacks:

main stack	Used by the core of RTOS and interrupts
process stack	Used by the application

Registers

R0-R12	generic purpose registers
MSP	main stack address
PSP	process stack address
LR	link register
PC	program counter
PSR	program status register
ASPR	application status register
IPSR	interrupt program status register
EPSR	execution program status register
PRIMASK	priority mask register
FAULTMASK	fault mask register
BASEPRI	base priority mask register
CONTROL	CONTROL register

Float registers

- Floating point unit (FPU) only on some processors
- Does not have to be used
- Provides additional registers:
  - 1/2 of the registers can be corrupted during a call
  - 1/2 of the registers have to be preserved during a call
- Context switching has to take care of the registers based on whenever FPU was used

MUNI
FACULTY
OF INFORMATICS