LightRT from hezhiying

位文康 崔卓 郭彦禛 罗嘉宏

报告：罗嘉宏 luojh@mail.ustc.edu.cn @ 化学物理系

Table of Contents

1 Developing Targets

- **LightRT** is a simple OS for STM32F1 MCUs
- Mainly for embedded/IoT (Internet of Things) systems
  - Robots
  - Smart home
  - Industry control
- Keep **simple** and **extensible**
  - Modularized design
  - Clean code
  - Easy development

1 Developing Targets

- Highlights
  - Fully opensource toolchain and libraries
  - Written in C programming language and ARM assembly
  - HAL (Hardware abstraction layer) isolates hardware differences
  - Software simulation instead of on-board debug
  - Simple structure and easily rewritting functions
  - User space and Kernel space isolation
  - Automatic tools for generating repeating code

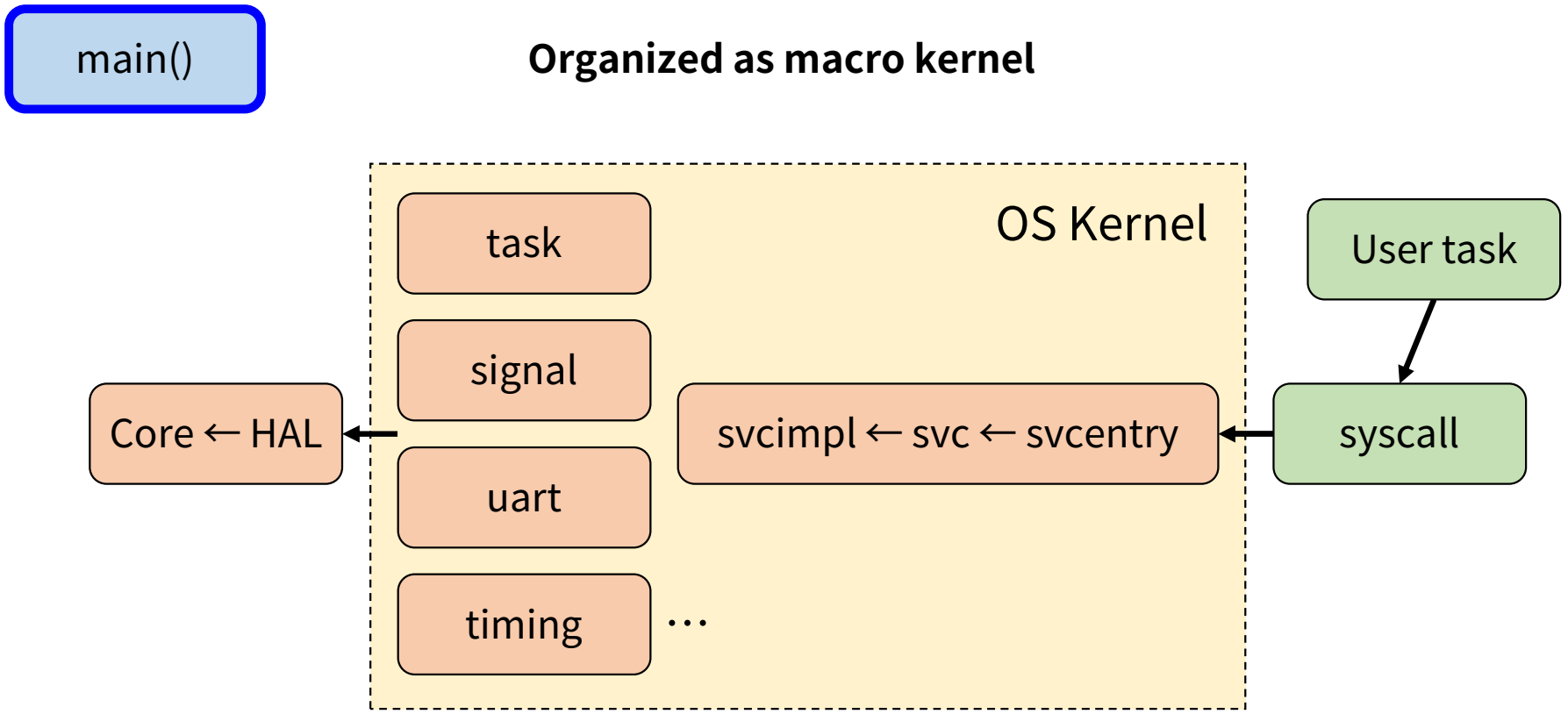
2 Environment and Toolchain

- STM32F1 is an **ARM Cortex-M** based MCU
  - RISC type **Armv7-M** instruction set
  - Low power
  - High performance
  - Reduced price
- Hardware target is **STM32F103ZGT6**
  - 72MHz clock
  - 96K SRAM (Internal)
  - 1M Flash (Internal)
  - External SRAM mapped via FSMC
  - No MMU (Memory Management Unit)

2 Environment and Toolchain

- Fully opensource toolchain
  - **Building**: GNU Make
  - **Compiler**: Cross-platform GCC (`arm-none-eabi-`)
  - **Debugger**: Multiarch GDB (`gdb-multiarch`)
  - **Hardware library**: `libopencm3`
  - **Simulation/VM**: Renode
  - **Version control**: `git`
- Environment packed by **Docker**
  - Isolated filesystem
  - Custom software installation
  - SSH / VSCode remote

3 Main code and code organization

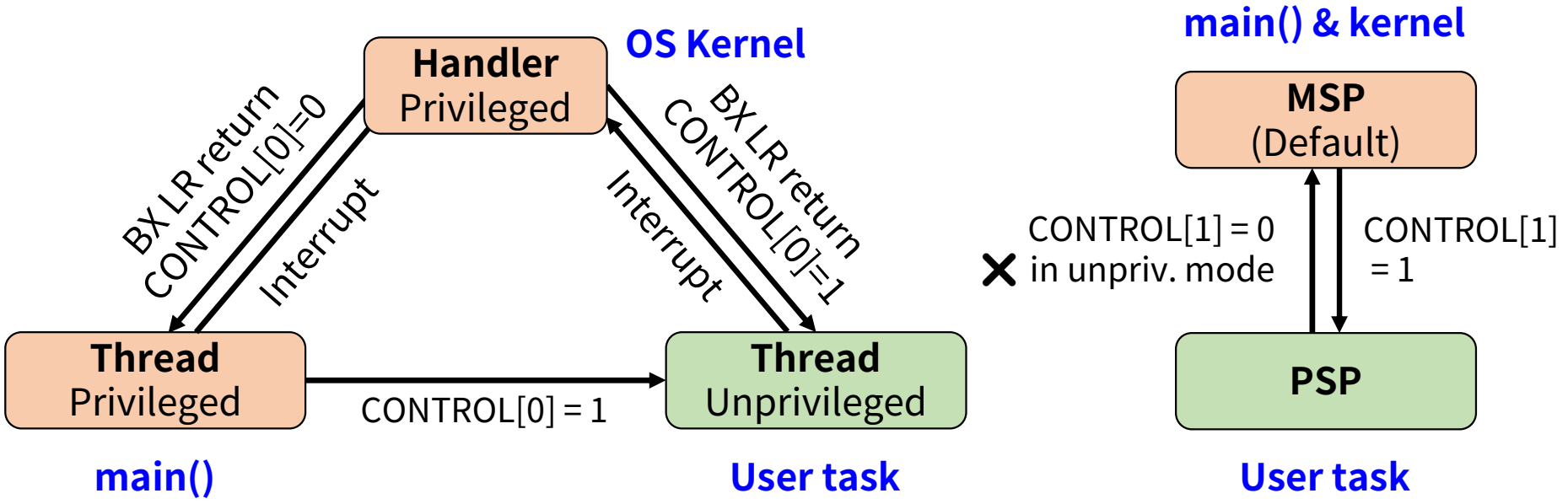


3 Main code and code organization

- `main()` – C system entry
  - Called by **reset handler** in **hardware library** (`libopencm3`)
- Initialize peripherals
  - System clock
  - UART controller
  - Timing (`systick`)
- Create first task
  - **Task 0** (`init`) will further run other tasks

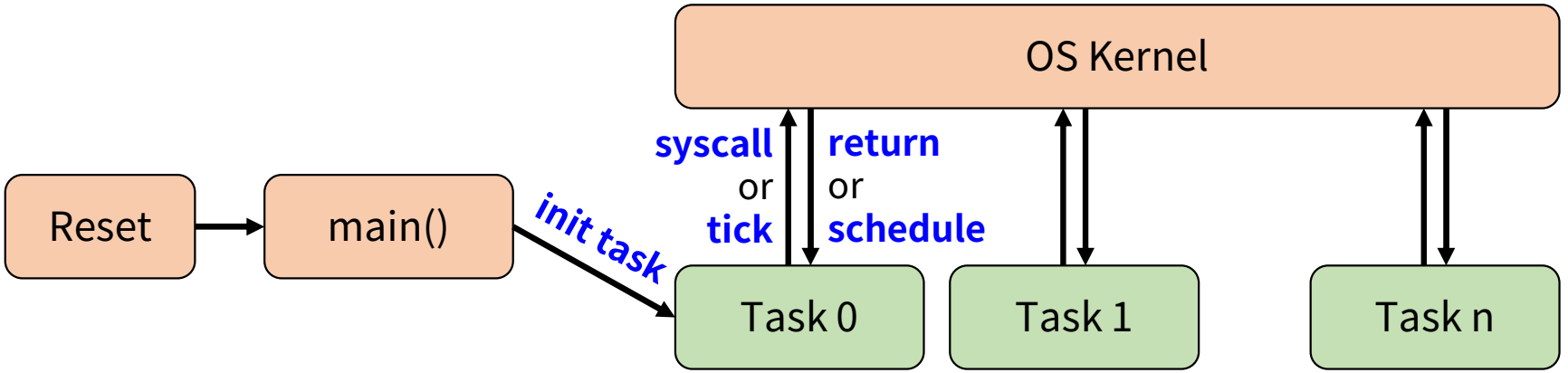
4 Running modes

- ARM Cortex-M core has **3 running modes** and **2 stacks**



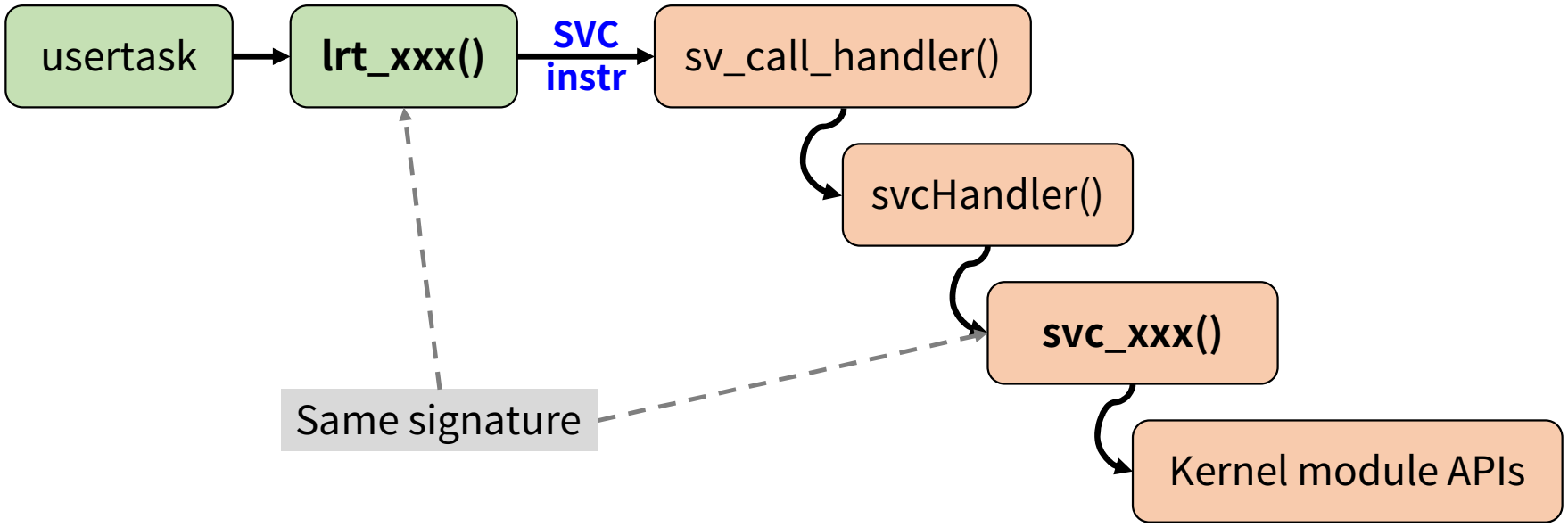
4 Running modes

Code	Running mode	Stack
Reset → main()	Thread, privileged	MSP
OS Kernel	Handler, privileged	MSP
User tasks	Thread, unprivileged	PSP



5 System call

- User task **call kernel routine** (task create, wait signal, ...)



5 System call

- Expose syscalls [`syscall.h`]
- Used to call kernel routines. Example:

```
__attribute__((naked)) uint32_t lrt_task_start (task_routine routine, int id);
```

Call this like

```
void mytask(void)
{
    // foo
}
lrt_task_start(mytask, 1);
```

5 System call

- System call switch to **kernel mode** [syscall.c]
- Used to call kernel routines. Example:

```
__attribute__((naked)) uint32_t lrt_task_start (task_routine routine, int id)
{
    __asm volatile (
        "mov r0, %0 \n" ← Pass arguments via register R0 to R3
        "mov r1, %1 \n"
        "svc #10 \n" ← Issue software interrupt via SVC instr.
        "bx lr \n" ← Return to user task
        :
        : "r" (routine), "r" (id)
        : "r0", "r1"
    );
}
```

5 System call

- SVC enters kernel mode (Handler mode) [`svcentry.c`]
- The SV call **interrupt handler**:

```
__attribute__((naked)) void sv_call_handler(void)
{
    __asm volatile(
        "TST lr, #4             \n" // which stack pointer to use?
        "ITE EQ                 \n"
        "MRSEQ r0, MSP          \n" // r0 = MSP if bit 2 is 0
        "MRSNE r0, PSP          \n" // r0 = PSP if bit 2 is 1
        "B svcHandler           \n" ← The handler in C
    );
}
```

5 System call

- The SV call C handler [svc.c]
- Big switch structure to **select kernel routine**

```
void svcHandler(uint32_t *svc_args) {
    uint8_t *svc_instr = (uint8_t *)(svc_args[6] - 2);
    uint8_t svc_number = *svc_instr & 0xFF; ← Get SVC# from calling instr.
    switch (svc_number)
    {
        case 10: ← Check the SVC number 10
            svc_args[0] = svc_task_start(← Call corresponding kernel routine
                (task_routine)(svc_args[0]),
                (int)(svc_args[1])
            ); break;
...} }
```

5 System call

- The SV call C implement [`svcimpl.c`]
- Check parameter and system states, etc.
- Call the **kernel module API**

```
uint32_t svc_task_start(task_routine routine, int id)
{
    ... ← Parameter check, etc.
    task_create(routine, id); ← Call module API

    return E_SUCCESS;
}
```

6 Task management

- Data structure (task control block, **`tcb_t`**)

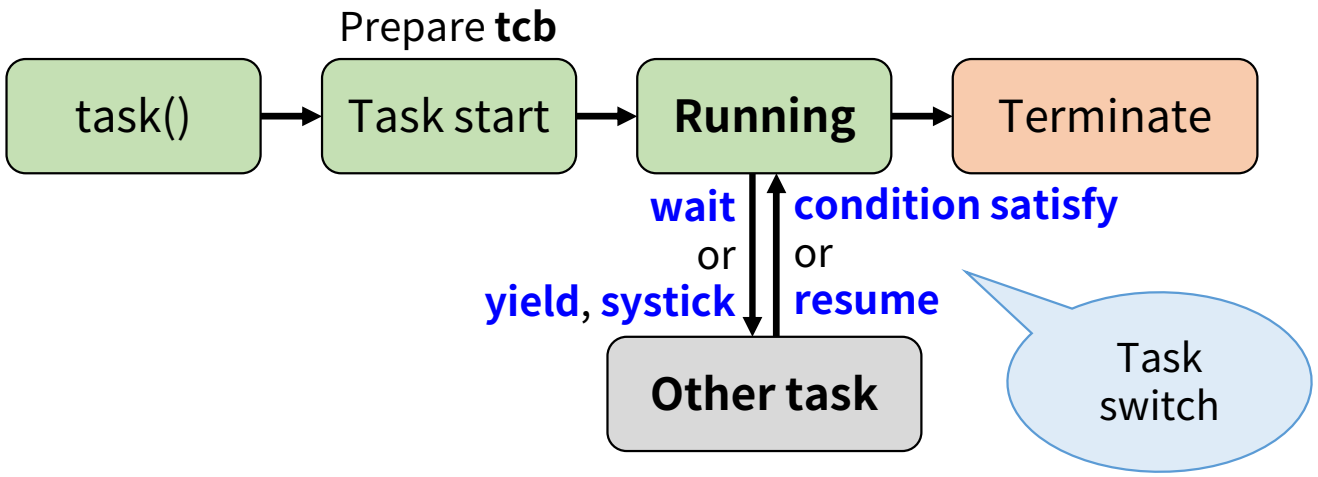
```
typedef struct
{
    uint32_t *sp;
    uint32_t stack[STACK_SIZE]; ← Thread stack

    int state;
    int prio; ← Properties

#include "signal.waitable.h" ← Additional waitable resources
#include "timing.waitable.h"
} tcb_t;
```

6 Task management

- Life of a **task** at a glance



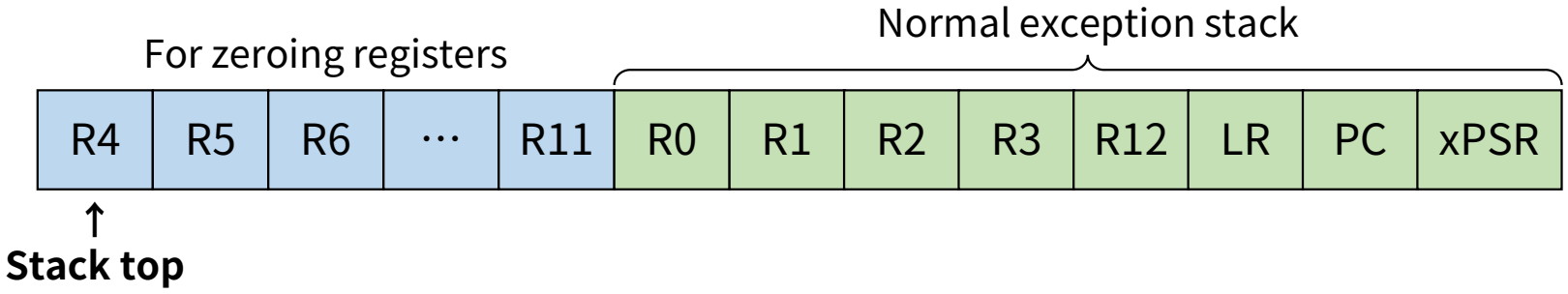
6 Task management

- The ARM Cortex-M **calling convention (AAPCS)**
  - **R0** to **R3**: Arguments, other save on stack
  - **R0**: Return value
  - **R0** – **R3**, **R12**: Caller saved
  - **LR**: Caller saved (via `BL`)
  - **R4** – **R11**: Callee saved
  - C calling prologue and epilogue save the general registers
  - Exception entry hardware save these registers (no software need)

`BL <label>` → Prologue → **Procedure code** → Epilogue → `BX LR`

6 Task management

- Start a task: **fake exception** method
  - **Construct a fake exception stack frame**
  - Use that stack as user task stack
  - Pop registers from stack, start user task



6 Task management

```
void task_create(task_routine task, int id)
{
    sp -= 8;
    sp[0] = 0x00000000;     // R0
    sp[1] = 0x01010101;     // R1
    sp[2] = 0x02020202;     // R2
    sp[3] = 0x03030303;     // R3
    sp[4] = 0x12121212;     // R12
    sp[5] = 0xFFFFFFFD;     // LR (return to thread mode, use PSP)
    sp[6] = (uint32_t)task; // PC (start address)
    sp[7] = 0x01000000;     // xPSR (Thumb bit)

    sp -= 8;
    for (int i = 0; i < 8; i++)
        sp[i] = 0x00000000;
    // Initialize other tcb fields
}
```

6 Task management

- Start a task: **fake exception** method


```
__attribute__((naked)) void task_start_first(uint32_t *sp)
{← argument sp is in R0
    __asm volatile(
        "MOVS R3, #3\n"
        "MSR CONTROL, R3\n" // Use PSP & Unpriv thread mode
        "MSR PSP, R0\n"     // Set PSP from argument
        "ISB\n"
        "POP {R4-R11}\n" ← Set the registers from stack
        "LDR LR, =0xFFFFFFFD\n"
        "BX LR\n"); ← Jump to start address as if in exception
}
```

6 Task management

- Start a task: **fake exception** method
  - Start the init task (task 0) in `main()`

```
task_create(init, 0);
current_task = &tasks[0];
register uint32_t *sp asm("r0") = current_task->sp;
task_start_first(sp);
```

6 Task management

- Switch from **task** to **task**
  - Save context (registers)
  - Select next task
  - Restore context (registers)
- Processed in **PendSV** interrupt
  - Handler mode, privileged

6 Task management

- **Context switch, Part I**
  - When PendSV is triggered, R0 – R3, R12, LR, PC, xPSR is pushed into stack by hardware.
  - Save R4 – R11 by software (also onto stack)
  - Save stack pointer to TCB

```
"MRS R0, PSP\n" // Load current process stack pointer into R0

"STMDB R0!, {R4-R11}\n" // Save R4-R11 onto current task's stack

// Save the updated stack pointer into current_task->sp
"LDR R1, =current_task\n"
"LDR R2, [R1]\n"
"STR R0, [R2]\n"
```

6 Task management

- **Context switch, Part II**
  - Select next task (in C)

```
tcb_t *select_next(void)
{
    int next_task = current_task_id;

    if (yield_source == YIELD_SYSTICK)
        next_task = get_next(); ← Auto select next task (priority scheduler)
    else
        next_task = yield_manual_next; ← Manually specified next task

    current_task_id = next_task;
    return &tasks[current_task_id];
}
```

6 Task management

- **Context switch, Part III**
  - Restore context in reversed order (stack is LIFO)

Exception entry save R0 – R3 ...
Select PSP
`"MRS R0, PSP\n"` **(R0 is *sp)**
Save R4 – R11
`"STMDB R0!, {R4-R11}\n"`
Save SP
```
"LDR R1, =current_task\n"
"LDR R2, [R1]\n"
"STR R0, [R2]\n"
```

Restore SP (R0 is *sp)
`"LDR R2, [R0]\n"`
Restore R4 – R11
`"LDMIA R2!, {R4-R11}\n"`
`"MSR PSP, R2\n"`
Select PSP and unpriv. mode
```
"MOV R0, #3\n"
"MSR CONTROL, R0\n"
"ISB\n"
```
Interrupt return restore R0 – R3 ...
```
"LDR R0, =0xFFFFFFFD\n"
"MOV LR, R0\n"
"BX LR\n"
```

7 Waitable objects

- **Waitable objects** are kernel objects that can be waited by a task. Currently: **timer** (delay), **signal**
- For example, when waiting for signal:

```
tasks[current_task_id].wait_signal_id = id;
tasks[current_task_id].state = TASK_WAIT; ← The task will no longer be switched to
When signal triggers:
for (int i = 0; i < MAX_TASKS; ++i)
    if (tasks[i].state == TASK_WAIT && tasks[i].wait_signal_id == id)
    {
        tasks[i].state = TASK_NORMAL; ← Resume scheduling
        tasks[i].wait_signal_id = -1;
    }
```

8 Communication

- Currently supports **high speed USART**
  - RS422 like differential signal
  - 26 Mbps max on APB2
- Future development
  - Berkeley socket style API
  - Ethernet / Wi-Fi
  - Asynchronous IO
  - Multiple nodes on switch (FPGA based?)

9 Writing user code

- Very **simple**, **intuitive**, just like writing C code on PC

```
int signalid;
void mytask(void) {
    while (1) {
        lrt_signal_wait(signalid);
        uart_printf("Hello, world! Task 1 Current time: %d\n", lrt_clock_tick());
    }
}
void mytask2(void) {
    while (1) {
        lrt_clock_delay(20);
        uart_printf("Hello, world! Task 2 Current time: %d\n", lrt_clock_tick());
        lrt_signal_active(signalid);
    }
}
```

9 Writing user code

- Very **simple**, **intuitive**, just like writing C code on PC

```
void init(void) {
    uint32_t currentid = lrt_task_current();
    uart_printf("Init task started. Task ID is %d.\n", currentid);
    lrt_signal_new(&signalid);

    lrt_task_start(mytask, 1);
    lrt_task_prio(1, 2);

    lrt_task_start(mytask2, 2);
    lrt_task_prio(2, 1);

    while (1)
        ;
}
```

10 Toolchain review II

- **SVCC**: **SVC C**ompiler for generating syscall repeating code

svc.def file

```
14
task_start 10 2
    routine task_routine
    id int

task_terminate 11 1
    id int

...
```

```
/* This file is generated using SVC compiler. DO NOT DIRECTLY MODIFY. Edit svc.def fil
#include "syscall.h"
/* Entry for
{
    __asm vo
        "mov
        "mov
        "svc
        "bx
        :
        : "r
        : "r
    );
}
```

```
#ifndef _U_SYSCALL_H
#define _U_SYSCALL_H
/* This file is generated using SVC compiler. DO NOT DIRECTLY MODIFY. Ed
#include "../
/* SVC Call
/* SVC Call
/* SVC Call
/* SVC Call
/* SVC Call
/* SVC Call
/* SVC Call
/* SVC Call
/* SVC Call
```

```
/* This file is generated using SVC compiler. DO NOT DIRE
#include <stdint.h>
#include "svc.h"
void svcHand
{
    uint8_t
    uint8_t
    switch (
    {
        case
```

```
#ifndef _SVC_H
#define _SVC_H
/* This file is generated using SVC compile
#include <stdint.h>
#include "lightrt.h"
/* SVC Implement 10 */ uint32_t svc_task_st
/* SVC Implement 11 */ uint32_t svc_task_te
/* SVC Implement 12 */ uint32_t svc_task_yi
/* SVC Implement 13 */ uint32_t svc_task_st
/* SVC Implement 14 */ uint32_t svc_task_cu
/* SVC Implement 15 */ uint32_t svc_task_pr
/* SVC Implement 20 */ uint32_t svc_clock_t
```

10 Toolchain review II

- Using **Docker** for development
  - Preinstalled and configured toolchain
  - Greatly reduce the time cost of setting up environment
  - Ensure all members have exactly the same environment
  - **Docker** + **Git** + **VSCode** (remote via SSH)
- **Renode** instead of **Qemu**
  - Renode simulates IoT hardware better!
  - Plenty of peripherals (FSMC, UART, GPIO pins)
  - Transparent GDB server support
  - Written in C#, extensible and scalable

11 Final words

- **Why C?**
  - Absolutely compiles everywhere
  - Always allow you to do what you want
  - Still (relatively) safe if using static objects
- **What can be done further?**
  - Advanced scheduling algorithms
  - More kernel modules (communication, critical section, large memory management, filesystem, common peripheral like I^2C, etc.)
  - Dynamical memory allocation (stack, etc.)
  - Dynamical loader (Executable Flash → RAM)
  - External MMU (Paging, Cache, RDMA etc.)

Thanks

Light**RT** from hezhiying

位文康 崔卓 郭彦禛 罗嘉宏