HW#4 RTOS Analysis

Chun-Jen Tsai
NYCU
12/02/2022

Homework Goal

- In this homework, you will use Aquila to run a real-time OS (RTOS) application, `rtos_test`, with two threads
- Your tasks:
  - Trace the code and analyze how thread management is done in the RTOS (put this in your report)
  - Measure the context-switching overhead of the application (need to add counters in Aquila)
  - Measure the synchronization overhead of the application (better to add counters in Aquila)
  - Implement mutex function of FreeRTOS using LR/SC instructions
- You should upload your report to E3 by 12/25, 23:55.

FreeRTOS

- FreeRTOS is a real-time operating system kernel for embedded devices
  - Developed by Richard Barry in 2003
  - Barry joined Amazon Web Services (AWS) and passed the stewardship of the project to AWS in 2017
  - The project adopts MIT License
- We used the version v202111.00 of FreeRTOS
  - All RISC-V unrelated sources are removed (way too big)
  - The original package available at https://www.freertos.org/
  - No source code of FreeRTOS is modified for Aquila

Source Tree of `rtos_test`

```
rtos_test/                                  277 directories, 1447 files
├── FreeRTOS/
│   ├── Demo/
│   ├── License/
│   ├── README.md
│   ├── Source/
│   │   ├── croutine.c
│   │   ├── event_groups.c
│   │   ├── include/
│   │   ├── list.c
│   │   ├── portable/
│   │   │   ├── Common/
│   │   │   ├── GCC/
│   │   │   │   └── RISC-V/
│   │   │   │       ├── Documentation.url
│   │   │   │       ├── chip_specific_extensions/
│   │   │   │       ├── port.c
│   │   │   │       ├── portASM.S
│   │   │   │       ├── portmacro.h
│   │   │   │       └── readme.txt
│   │   │   ├── MemMang/
│   │   │   └── readme.txt
│   │   ├── queue.c
│   │   ├── stream_buffer.c
│   │   ├── tasks.c
│   │   └── timers.c
│   └── Test/
├── FreeRTOSConfig.h
├── Makefile
├── rtos_test.c
└── rtos_test.ld
```

portASM.S → Timer interrupt handling routine.

write in

tasks.c → Multi-thread managers.

FreeRTOSConfig.h → FreeRTOS configurations for application.

After "make", a `build/` directory would contain the `rtos_test.map` and `rtos_test.objdump` files.

The executable `rtos_test.elf` will be in the top directory

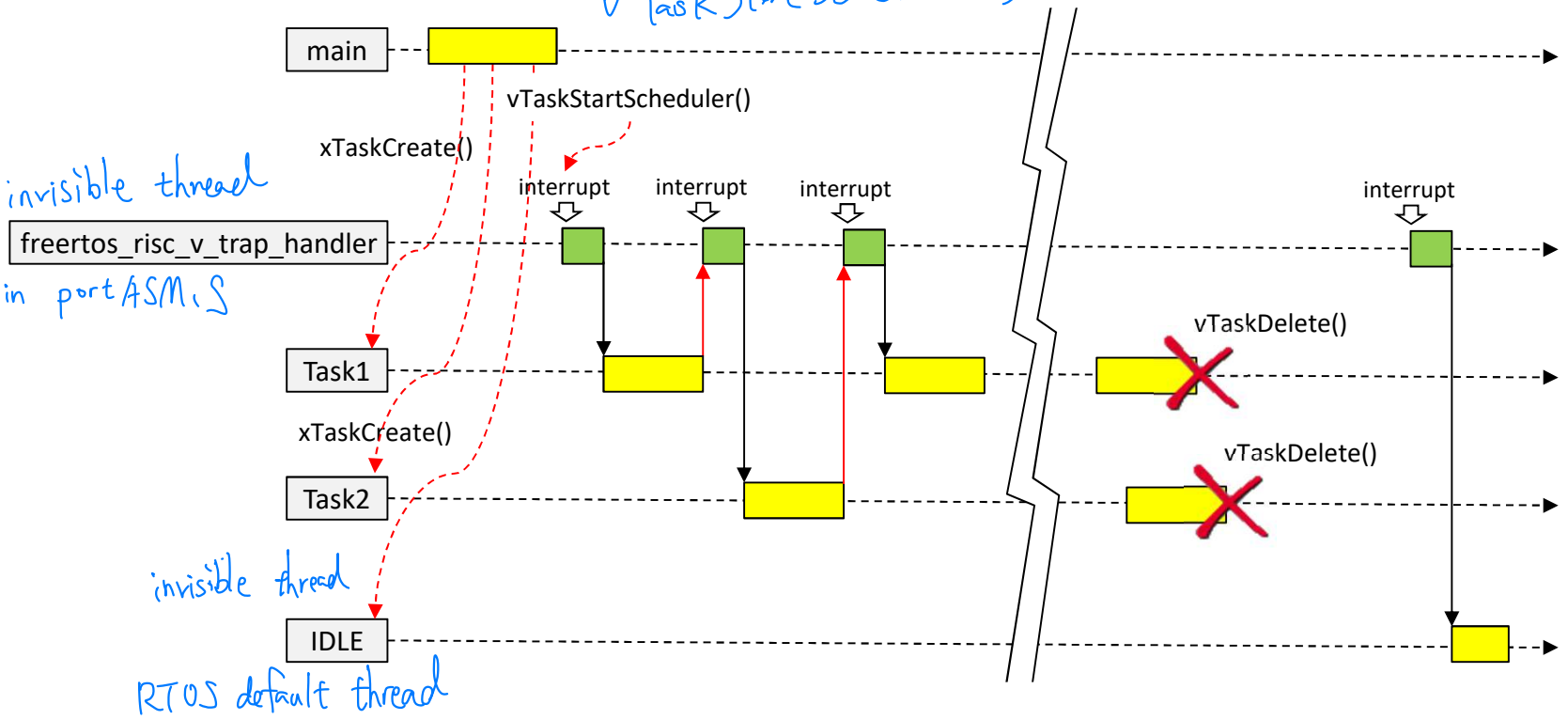
UML: Unified Modeling Language - Sequence diagram, Class diagram are 2 famous UML.

Application Behavior

- The are three visible threads and two invisible threads in the application:

xTaskCreate() : creates thread.

vTaskStartScheduler() : starts the thread execution.



Linux top instruction calculates CPU Usage by calculating 1 − CPU time in IDLE state

Protection of Shared Resources

- In a preemptive multi-tasking OS, shared resources cannot be modified without protection:

```
volatile int
    shared_counter = 0,
    done = 0;

Task1()
{
    while (!done)
    {
        shared_counter++;
    }
}

Task2()
{
    while (!done)
    {
        shared_counter++;
    }
}
```

During the execution of 0x8000_0040 ~_0048.
An Interrupt may hit, corrupting the data.

```
80000030 <Task1>:
80000030:  lui   a3, 0x80008
80000034:  lw    a5, 8(a3)    # <done>
80000038:  bnez  a5, 80000054 <Task1+0x24>
8000003c:  lui   a4, 0x80008
80000040:  lw    a5, 12(a4)   # <shared_counter>
80000044:  addi  a5, a5, 1
80000048:  sw    a5, 12(a4)
8000004c:  lw    a5, 8(a3)
80000050:  beqz  a5, 80000040 <Task1+0x10>
80000054:  ret
```

Mutex: value of 1 or 0, can only be released by the locker.

Semaphore: Not limited to binary; can be int. Anyone can change the value of a Semaphore

Default Execution

- When you compile and run the application you should see the following output
  - The result is bad because we did not enable mutex protection

```
================================================================================
Copyright (c) 2019-2022, EISL@NCTU, Hsinchu, Taiwan.
The Aquila SoC is ready.
Waiting for an ELF file to be sent from the UART ...

Program entry point at 0x800003A4, size = 0xAE58.
--------------------------------------------------------------------------------

Task 1 start running ...

Task 2 start running ...


At the end, the shared counter = 10000
Task1 local counter = 10000
Task2 local counter = 10000
Task1 counter + Task2 counter != Shared counter, the counter is corrupted.
```

Task 1 & Task 2 not blocked

Application with Mutex Protection

- With mutex protection of the shared variable, the output would be good
  - Made the change: "`#define USE_MUTEX 1`" in `rtos_test.c`

```
================================================================================
Copyright (c) 2019-2022, EISL@NCTU, Hsinchu, Taiwan.
The Aquila SoC is ready.
Waiting for an ELF file to be sent from the UART ...

Program entry point at 0x80000424, size = 0xAE58.
--------------------------------------------------------------------------------

Task 1 start running ...

Task 2 start running ...


At the end, the shared counter = 10000
Task1 local counter = 61
Task2 local counter = 9939
The shared counter is protected well.
```

Context-Switching Overhead

- A preemptive multi-tasking OS uses timer interrupts to assign CPU usage from one thread to the other
  - The context-switching overhead is inversely proportional to the time quantum (slice)
  - The default of FreeRTOS time quantum is 10 msec
- To measure the context-switching overhead, you must count the number of cycles between:
  - A timer interrupt arrives
  - A new thread begins execution
- You can change the time quantum size down to 5 or 1 msec to see its impact on the overhead

The Timer Interrupt Device

- In Aquila, the module `clint` is used to provide timer interrupts and software interrupts
- `Clint` has three registers
  - `mtime`: 64-bit counter of timer ticks
  - `mtimecmp`: Upper-threshold to trigger a timer interrupt
  - `msip`: a 32-bit register to trigger a software interrupt
- FreeRTOS will update `mtimecmp` to setup the next context-switch time (based on time quantum duration)

timer interrupt.

clint stands for Core Local Interrupt.

Changing Time Quantum

- In FreeRTOS, time quantum is configured by the header file: `FreeRTOSConfig.h`
  - Default time quantum is 10 msec:

```
...

#define CLINT_CTRL_ADDR                 ( 0xF0000000UL )
#define configMTIME_BASE_ADDRESS        ( CLINT_CTRL_ADDR + 0x0UL )
#define configMTIMECMP_BASE_ADDRESS     ( CLINT_CTRL_ADDR + 0x8UL )

#define configUSE_PREEMPTION            1
#define configUSE_IDLE_HOOK             0
#define configUSE_TICK_HOOK             1
#define configCPU_CLOCK_HZ              ( ( uint32_t ) ( 41666667 ) )
#define configTICK_RATE_HZ              ( ( TickType_t ) 100 )
#define configMAX_PRIORITIES            ( 7 )
#define configMINIMAL_STACK_SIZE        ( ( uint32_t ) 100 )
#define configTOTAL_HEAP_SIZE           ( ( size_t ) ( 12 * 1024 ) )

...
```

Synchronization Overhead

- In the sample application, we use two synchronization schemes of FreeRTOS: critical sections and mutex
  - Critical sections are used to protect the UART device so that different threads can print concurrent messages properly
  - A mutex is use to protect a shared variable so that both threads can modify the variable without corrupting it
  - In FreeRTOS, mutex is a special type of semaphore.
- You should measure the overhead of these synchronization schemes and put that in your report
  - The cycles to enter & leave a critical section
  - The cycles to take & give a mutex

Mutex Take/Give in FreeRTOS

- Note that, for FreeRTOS, most of the overhead in mutex take and give operations are spent in handling priority inversion
  - Queue structures are used to avoid a low priority task to block high priority tasks
  - We do not have different priorities for the two threads in the sample program
- Removing priority inversion processing reduces the synchronization overhead significantly, but it is not the right thing to do.

Priority Inversion:

Low priority thread was sent into the waiting queue without releasing its mutex. While a higher priority task waits for the held mutex, making it looks like an even lower priority task. Solution is to temporarily promote the low priority task, finishing it and release the mutex.

Mutex for Synchronization

- A mutex is a variable to indicate two states: “locked” and “unlocked” of a shared resource:

```
int mutex;

mutex_take(mutex);
    code that uses the shared resource.
mutex_give(mutex);
```

Execution blocked here if the mutex has been taken by other threads.

- There are several approaches to implement a mutex: ISA-independent SW, ISA-dependent SW, or hardware

Mutex Implementation

- Software mutex implementation techniques
  - Software algorithms (e.g. the Peterson's algorithm)
    - Drawback: time-consuming
  - Atomic test-and-set
    - Drawback: less efficient for multi-core systems
  - Conditional load-store
    - Drawback: only supported by new ISAs & CPUs
- Hardware mutex approach
  - A HW mutex is a device that contains a list of mutex registers
    - An unlocked register has zero in it
    - Each thread write their ID to the register to lock the mutex
    - Each register conditionally accepts the write requests
  - Suitable for synchronization even between HW and SW

Peterson's Mutex Algorithm

- Peterson's algorithm† guarantees exclusive accesses to a shared resource among n threads (running on n cores) without special assembly instructions
- A two-thread version is as follows:

CPU 0

```
/* trying protocol for T1 */
Q1 = true; /* request to enter */
TURN = 1;  /* who's turn to wait */
wait until not Q2 or TURN == 2;
Critical Section;
/* exit protocol for T1 */
Q1 = false;
```

CPU 1

```
/* trying protocol for T2 */
Q2 = true; /* request to enter   */
TURN = 2;  /* who's turn to wait */
wait until not Q1 or TURN == 1;
Critical Section;
/* exit protocol for T2 */
Q2 = false;
```

† G. L. Peterson, "Myth about the Mutual Exclusion Problem," *Information Processing Letters*, **12**, no 3, June 30, 1981.

Test-and-Set Atomic Instructions

- For synchronization, a thread must execute the following code before entering a critical section:

```
int mutex; /* '0' means unlocked, '1' means locked */

while ( test_and_set(mutex) == 1) /* busy waiting */
  Code that uses the shared resource.
mutex = 0;
```

- A 'SWAP' instruction (`amoswap.w` in RSIC-V) can be used to implement the test-and-set function:

```
int test_and_set(int mutex)
{
        temp = mutex;
        mutex = 1;
        return temp;
}
```

The first two lines cannot be interrupted during execution!

Conditional Load/Store Instructions

- Conditional load/store allows atomic operation without locking the buses
  - In RISC-V, we have LR/SC instructions
  - In ARM, we have LDREX/STREX instructions
- In this HW, you should try to use the atomic instructions (either lock-based or lock-free) to implement mutex and measure the overhead
  - The atomic operations are implemented in `atomic_unit.v`

Example Code:

- Mutex take:

local label

```
asm volatile ("lui t0, %hi(lock_addr)");
asm volatile ("lw t3, %lo(lock_addr)(t0)");
asm volatile ("li t0, 1");
asm volatile ("0:");
asm volatile ("lw t1, (t3)");
asm volatile ("bnez t1, 0b");
asm volatile ("amoswap.w.aq t1, t0, (t3)");
asm volatile ("bnez t1, 0b");
```

- Mutex give:

```
asm volatile ("lui t0, %hi(lock_addr)");
asm volatile ("lw t3, %lo(lock_addr)(t0)");
asm volatile ("amoswap.w.rl x0, x0, (t3)");
```

Comments on the Homework Upload

- The key point of this homework is RTOS analysis and implementation of a mutex using atomic instructions
- Performance optimization is not required
  - Hardware optimization of the atomic unit is required to gain good performance
- Your analysis of the RTOS behavior will account for 50% of the report credit