FreeRTOS Fault Injection Simulator

System and Device Programming Course - Politecnico di Torino - A.Y. 2021/2022

Professors: A. Savino, S. Quer, G. Cabodi, A. Vetrò

Candidates: Alessandro Franco, Simone Alberto Peirone, Marzio Vallero

Summary

Project's Objective

Stability and flexibility
Expandability and Customization
Cross Platform Compatibility
Ease of Use
Parallelism

Project Environment

The project is a C based application, which runs a modified version of **FreeRTOS's kernel**.

The CMake utility has been used to setup the compiler's environment for cross platform compatibility.
For this reason, most of the functions we introduced are masked and provide **two platform implementations**, one for the **Win32** and the other for the **POSIX** environment.

The project can be compiled by executing either the *`compile_win32.bat`* or the *`compile_posix.sh`*, on Windows or POSIX respectively.

The project may be run also inside of a **Docker container** for ease of use, through the command:
`docker build -t project:latest . && docker run project:latest`

FreeRTOS

FreeRTOS is an **open source real-time kernel** developed under the MIT license.

The version of the kernel we used has been compiled with the **heap_5 memory allocation policy**, which allows it to allocate and deallocate non contiguous areas of memory.

Its main strength has been its ability to be executed on threads, letting us build and environment which could be easily simulated on any host OS, thus removing the need of a physical ad-hoc board for testing.

The presence of **hook macros** provided by the RTOS favoured the custom development of the features we wanted to implement.

Project Structure

The Orchestrator Process

The orchestrator process is the main entrypoint of the project. It allows to list injection targets, run a golden execution and start injection campaigns. Each time a FreeRTOS simulation is required, it handles the execution of a child process, composed by:

- a **FreeRTOS instance** (multiple threads, one for each task)
- an **injector thread**

It **records the output of the children** and their execution time, determining which impact the injection had on the expected execution.

Since a crash would stop the regular execution of the program, **if** after **the injection** the RTOS instance **does not respond to interrupts** and the execution time is exceeded by five times the golden execution time, **the result of the injection is considered a crash** directly by the orchestrator and **the injection campaign is resumed**.

The Golden Execution

The golden execution represents a **full and correct execution** of the benchmark **without** the occurrence of **any externally introduced fault**. The benchmark consists of a qsort tasks and two tasks sharing a message queue.

It produces the *golden.txt* file, containing execution time in nanoseconds and the output of the benchmark.

The statistics provided by the *golden.txt* file are **used as a comparison metric** for injection campaigns in order to spot out **delays** and **faulty executions**.

The contents of the *golden.txt* file are subject to slight differences in the recorded value of the execution time, as the simulator runs on a host operating system and as such can be preempted by it at any time.
Generally a median of about *241 ms* with a standard deviation of *4 ms* is registered.
The outputs of the benchmark are deterministic, being a sorted list of words.

Target Extraction

Two alternative approaches:

- **dynamically read symbols from the simulator's executable file**: non-viable due to ASLR (Address Space Layout Randomization) which prevents a-priori knowledge of the memory addresses.
- **read injection targets at simulator startup** into a global accessible list.

```
struct target_s
{
   int id;
   char name[64];
   void *address;
   unsigned int size;  // size of an element
   unsigned int nmemb; // array length
   unsigned int type;
   struct target_s *parent;
   struct target_s *content;
   struct target_s *next;
};

target_t * read_timer_targets(struct target_s *target) {
   APPEND_TARGET(target, xActiveTimerList1, TYPE_LIST, NULL);
   APPEND_TARGET(target, xActiveTimerList2, TYPE_LIST, NULL);
   // ...
   return target;
}
```

Each source code file in the FreeRTOS codebase exposes a **read_*filename*_targets** function that returns the list of available target.

Targets are represented using the **target_s** structure which provides access to the **target type**, **name**, **starting memory address** and other details (i.e. the array length if applicable).

The available targets are the kernel structures defined in the `timer.c` and `tasks.c` source files.

The FreeRTOS Instance

The FreeRTOS instance and the injections orchestrator share the same codebase. The behaviour of the simulator is controlled by its command-line arguments: `–campaign input.csv` starts a new injections campaign.

1. The simulator **matches the targets** specified in the `input.csv` file with the available targets.
2. For each target, the simulator **forks a FreeRTOS subprocess** with an additional thread that will perform the injection on the requested target at proper time.
3. The injector thread **performs the injection**.
4. The simulator waits for the child to terminate and **classifies the injection** outcome according to the returned exit code

```
# example of input.csv
# target,n_injections,time,time_interval_width,injection_distributions
uxTopReadyPriority,50,20000000,5000,u
xNumOfOverflows,50,20000000,5000,t
xDelayedTaskList2,50,20000000,5000,u
```

The Injector Thread

The injector thread handles **Single Event Upset** injections (SEUs) into the FreeRTOS **kernel data structures**.

The parameters needed to execute an injection are contained in the *thData_s* structure.

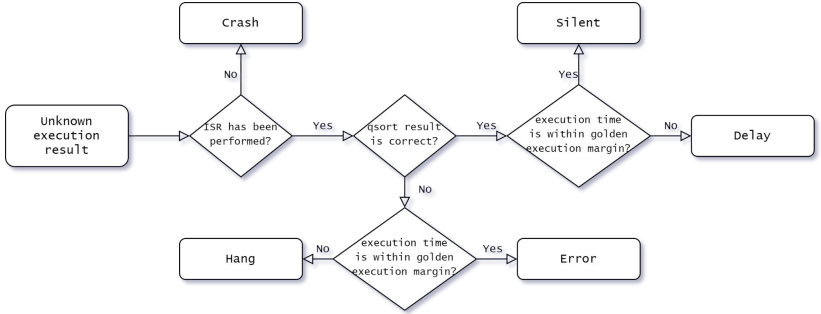
```
typedef struct thData_s
{
   void *address;
   unsigned long injTime, timeoutNs, offsetByte, offsetBit;

   // lists only
   int isList;
   int listPosition;

   // pointers only
   int isPointer;
   void *offset;

   target_t *target;
} thData_t;
```

Memory addresses are gathered during the **target extraction phase**: according to the size of the data structure, a bit is randomly selected for injection.

Since user level tasks cannot access the kernel space, the child process is used to handle the parallel execution of the instance's threads, providing to the injector free access to the target data structures.

Timings are marked in the input file used for the campaign. If a time deviation is reported, depending on the specified distribution the injection is shifted by a random number of nanoseconds.

Execution Classification



The simulator can classify injected executions as either **Silent, Delay, Error, Hang** or **Crash**.
It does so by performing comparisons between the *golden.txt* file and the current FreeRTOS's *loggerTrace*, with information formatted as `<time> <hookMacroType> <taskName>` on the last 10 calls performed by a *traceHookMacro*.

A correct benchmark output characterizes a **Silent** or **Delayed** execution, while an incorrect one characterizes an **Error** or **Hang** execution. The **Delayed** executions are thresholded with a **margin of 5%** with respect to the golden execution. Failure to process the ISR is classified as a **Crash**.

Once the FreeRTOS instance has been classified, the child communicates its exit status to the Orchestrator Process through the *exit()* call. Invoking **unhandled exceptions** or **segmentation faults** is directly **classified** as a **Crash**.

Parallelization

Since **statistically relevant data requires at least *10^5* injections** (see DATE conference 2009), parallel computation techniques have been exploited to speed up data gathering.

Each FreeRTOS instance is executed in an isolated process, thus can be parallelized by means of a waiting queue managed by the Orchestrator process.

A **double wait is used for each child process**, one on the process and the other onto an external **watchdog timer** associated to it, which waits for ***500%*** of the time **of a Golden Execution**.
Whether the process or the timer get signalled first, the result will be the same: the **child process gets terminated** and its **watchdog timer gets destroyed**.

The number of parallel processes can be set to any number, however it's important to note that overloading the host OS's scheduler will skew the results towards **Delay** and **Hang** exit statuses of the FreeRTOS instances.

Usage and Results

Execution Commands and Results

Print a list of all the **possible injection targets** in the kernel's structures:
`./sim.exe --list`

Run a **golden execution**:
`./sim.exe --golden`

Run a **single injection** at a specific time and on a specific bit, **used internally** to generate all instances of an injection campaign:
`./sim.exe --run <targetStructureName> <timeInjection> <offsetByte> <offsetBit>`

Run **one or multiple injection campaigns**:
`./sim.exe --campaign <inputFileName>.csv [-j=<parallelism>] [--no-pg-bar] [-y]`

Target	Time (ns)	nExecs	Silent %	Delay %	Error %	Hang %	Crash %
uxTopReadyPriority	20000000	50	0.00%	4.00%	0.00%	4.00%	92.00%
xNumOfOverflows	20000000	50	88.00%	12.00%	0.00%	0.00%	0.00%
xDelayedTaskList2	20000000	50	76.00%	24.00%	0.00%	0.00%	0.00%
xIdleTaskHandle	20000000	50	0.00%	0.00%	0.00%	0.00%	100.00%
xSchedulerRunning	20000000	50	66.00%	32.00%	0.00%	0.00%	2.00%
xTimerTaskHandle	20000000	50	92.00%	8.00%	0.00%	0.00%	0.00%
xTimerQueue	20000000	50	0.00%	0.00%	0.00%	0.00%	100.00%
xNextTaskUnblockTime	20000000	50	62.00%	4.00%	0.00%	32.00%	2.00%
xTasksWaitingTermination	20000000	50	58.00%	14.00%	0.00%	0.00%	28.00%
uxDeletedTasksWaitingCleanUp	20000000	50	0.00%	0.00%	0.00%	0.00%	100.00%
xActiveTimerList1	20000000	50	62.00%	24.00%	0.00%	0.00%	14.00%
*pxCurrentTimerList	20000000	50	68.00%	20.00%	0.00%	0.00%	12.00%
pxCurrentTCB.pxTopOfStack	20000000	5	0.00%	0.00%	0.00%	0.00%	100.00%
pxCurrentTCB.uxPriority	20000000	5	0.00%	0.00%	0.00%	0.00%	100.00%
pxCurrentTCB.ucNotifyState[2]	20000000	5	20.00%	80.00%	0.00%	0.00%	0.00%
pxReadyTasksLists[0][0]	20000000	5	20.00%	0.00%	0.00%	0.00%	80.00%

An example of the output produced by **small injection campaigns on different targets**.

Most of the results fall either into the Silent of Crash categories, due to the fact that most of the targets here presented belong to the `tasks.c` file.

There are no executions classified as Error due to the fact that injections on the **kernel's structure** tend to break higher level components. To cause an Error execution, we would need to inject onto **user level structures**.

Target and Benchmark Customization

To add new targets, it's necessary to implement a function internal to the file you want to inject onto and call that function during the target acquisition phase in the main.

```
// src/main.c

int main(int argc, char **argv)
{
   // ...
   // Read the available injection targets
   targets = read_tasks_targets(NULL);
   targets = read_timer_targets(targets);
   // add more target here
}
```

```
// src/main_blinky.c

/* Custom benchmarking */
xTaskCreate(prvQSortTask,               // Task function
            "qSort",                    // Task name
            configMINIMAL_STACK_SIZE,   // Task stack size
            NULL,                       // Task parameters
            tskIDLE_PRIORITY + 2,       // Task priority
            NULL);                      // Task handle
```

To customize the benchmark, it's enough to modify the tasks declared in main_blinky and/or add new callbacks in the `src/benchmark/benchmark.c` file.

Conclusions

Conclusions

The project aimed at producing an **operating system independent simulator** of FreeRTOS able to perform multiple injection campaigns on kernel structures and to analyze the results. Moreover, the execution on a host operating system had to **not break the real-time behaviour** of FreeRTOS. All of these properties have been successfully achieved, despite the **numerous pitfalls** we incurred into during development.

We delved into **advanced programming techniques** for code structuring and management, deepening our understanding and improving our **problem solving skills**. More than anything else, this project has been a test of knowledge and patience, forcing us to think outside of our programming *comfort zone*.

We are proud of our work and hope sincerely that it will aid developers looking to test and improve the robustness of FreeRTOS distributions in the future.

The Team

Alessandro Franco

Simone Alberto Peirone

Marzio Vallero

Thank you.