arm

EL0 app support in TF-RMM

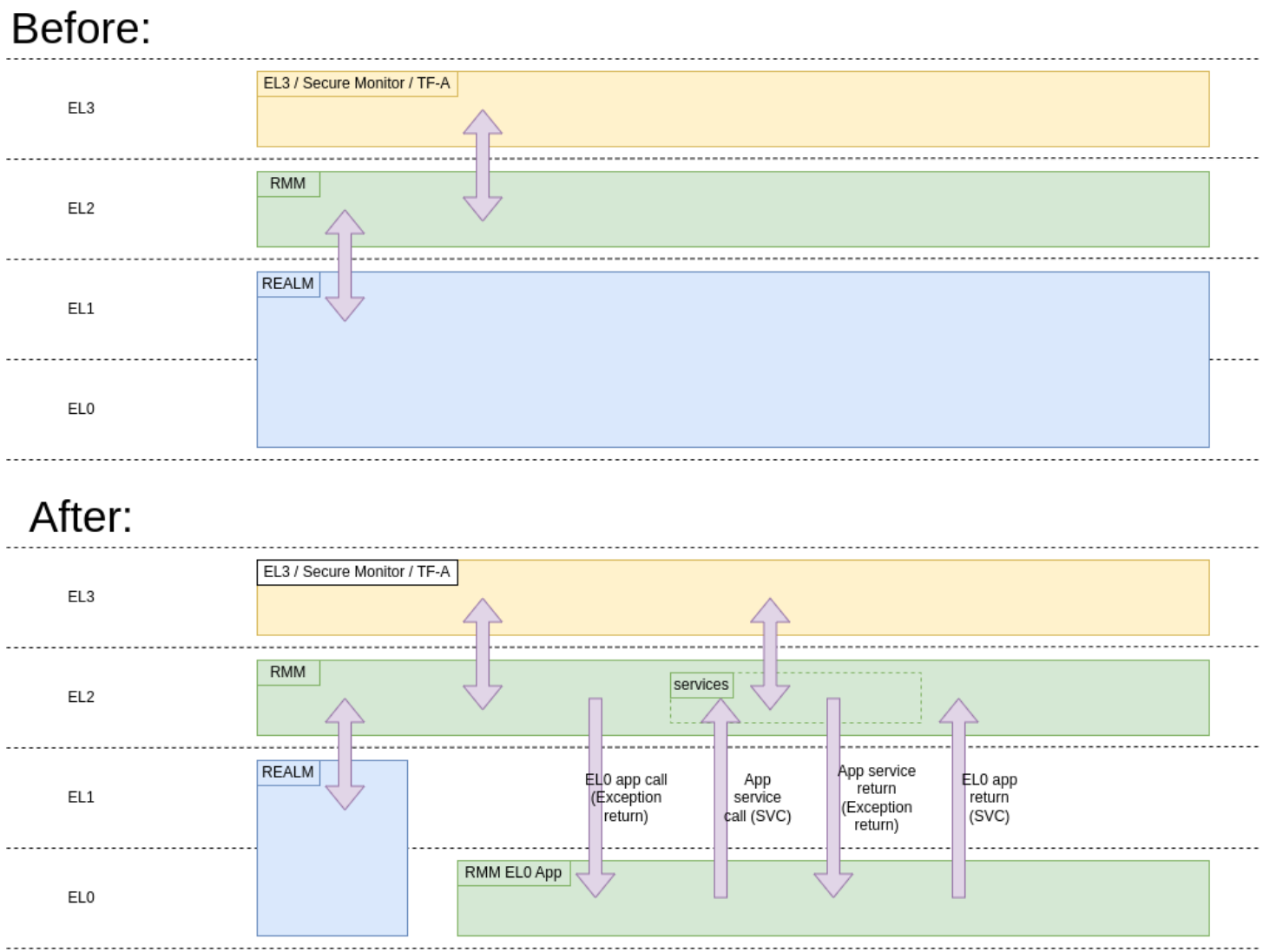
Rationale, Design and Implementation details

Mate Toth-Pal , Soby Mathew

23-01-2025

Introduction

- EL0 app support is a mechanism to Deprivilege parts of RMM by running them at EL0 (using Virtual Host Extension)
- Isolated address space for the EL0 App.
- Option to have per-CPU or per REC/PDEV thread for the App.
  - Referred to as an app instance in later slides.



Why is this being done ?

- Sandboxing of complex functionality
- Isolation of sensitive data – similar principles as Address space isolation (ASI) in Kernel
  - Better CPU speculation/side-channel protection for Sensitive data in app
    - The app is unmapped by default and only mapped in when needed.
    - Possibility to apply side-channel mitigations as required on an app basis.
      - Eg, cache flush of app Virtual address space on app entry and exit.
- Allow for easier pre-emption/yield during long sequences.
- More Robustness for RMM in case an instance of EL0 app misbehaves.
  - A crash in an instance, may not affect another instance if the global data is not modified.
  - Possibility to re-initialize an app for new instances if the whole app crashes.

Advantages and Criteria

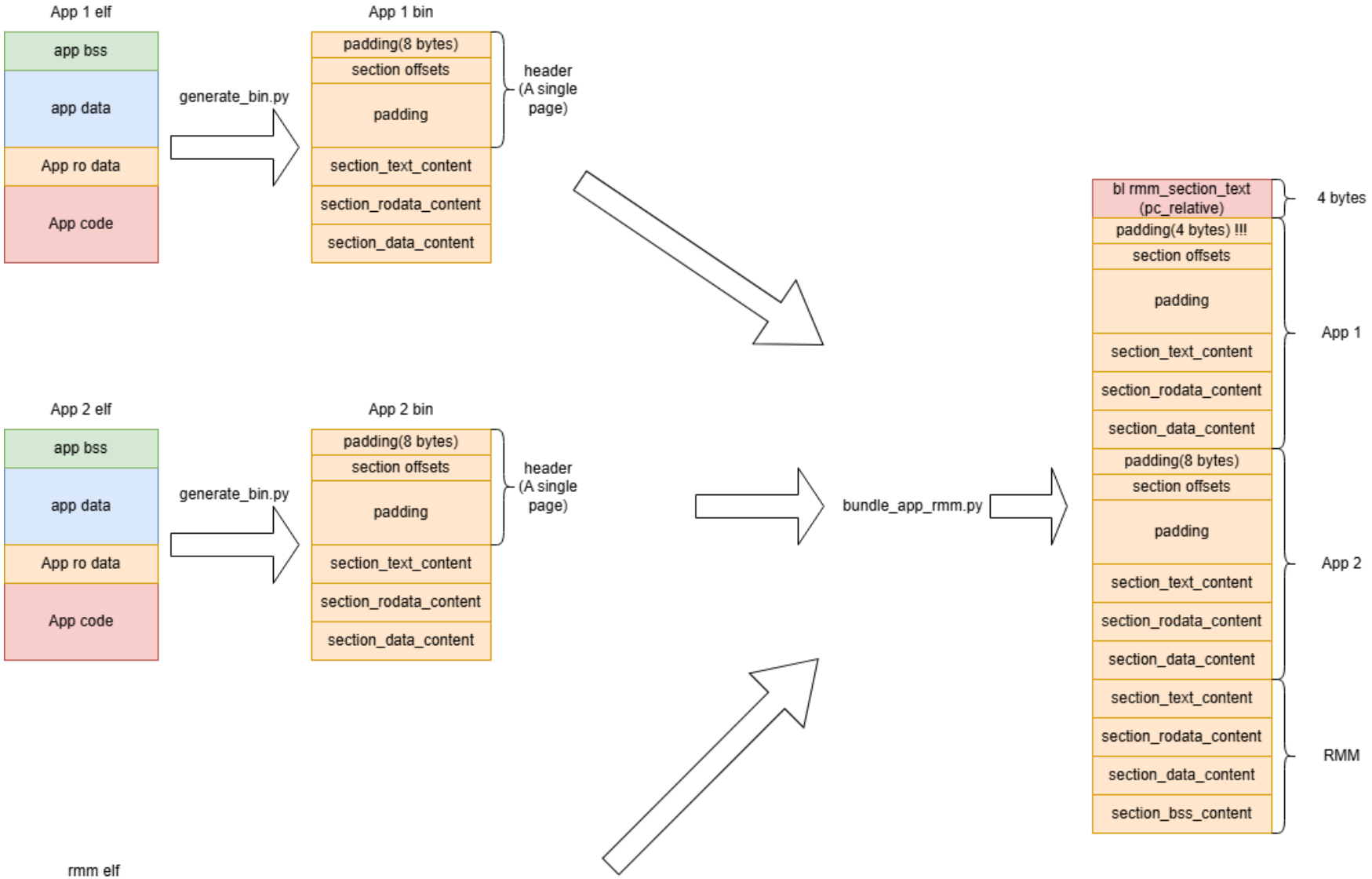
Other Advantages

- Can allow easier enablement of SIMD and other extensions within the App.
- Would allow platform specific implementations to be made an app thus keeping core RMM generic.
  - May also multiple variants of an app to be packaged to RMM at build time and one variant is used at runtime.

Criteria for a functionality to be moved to EL0 app

- Functionality has significant complexity, imports 3rd party code or sensitive data.
- Contained to certain use-cases and hence the overhead can be contained to those code-paths
  - Eg: Attestation, Device Assignment.
- Functionality which has platform specific implementation.

EL0 App build and packaging (RMM_ARCH=aarch64)


App 1 elf
app bss
app data
App ro data
App code
generate_bin.py
App 1 bin
padding(8 bytes)
section offsets
padding
section_text_content
section_rodata_content
section_data_content
header
(A single
page)
App 2 elf
app bss
app data
App ro data
App code
generate_bin.py
App 2 bin
padding(8 bytes)
section offsets
padding
section_text_content
section_rodata_content
section_data_content
header
(A single
page)
bundle_app_rmm.py
bl rmm_section_text
(pc_relative)
4 bytes
padding(4 bytes) !!!
section offsets
padding
section_text_content
section_rodata_content
section_data_content
App 1
padding(8 bytes)
section offsets
padding
section_text_content
section_rodata_content
section_data_content
App 2
section_text_content
section_rodata_content
section_data_content
section_bss_content
RMM
rmm elf
bss
data
ro data
code
objcopy
rmm bin
section_text_content
section_rodata_content
section_data_content
section_bss_content

Building and packaging apps

- Apps are built as separate elf files
- A python script is used to
  - Extract the binary content of the relevant sections
    - .text, .rodata, .data
  - Prepend a header to the extracted sections
- Elf sections must be page aligned, so that direct mapping .text, .rodata in the app memory is possible
- Header format is defined in RMM source as a C structure, the python script needs to be kept up-to-date on header format change
  - Header contains a header version, elf section offsets and lengths, stack/heap page count, app name and app id

Building and packaging apps (continued)

- The code running in EL2 built as a separate elf file
- Bin file is created from the elf using objcopy
- A bundled bin file is generated from the app bin files and the RMM EL2 bin file using a python script
  - The RMM EL2 bin is appended after the app binaries
  - A BL instruction is injected at the beginning of the bundled bin file that branches to the start of the RMM EL2 code
- During boot, The RMM EL2 code uses the address saved in the LR by the initial BL instruction to find the app headers for parsing.

EL0 app Memory and Runtime details for RMM_ARCH=aarch64

RMM memory setup

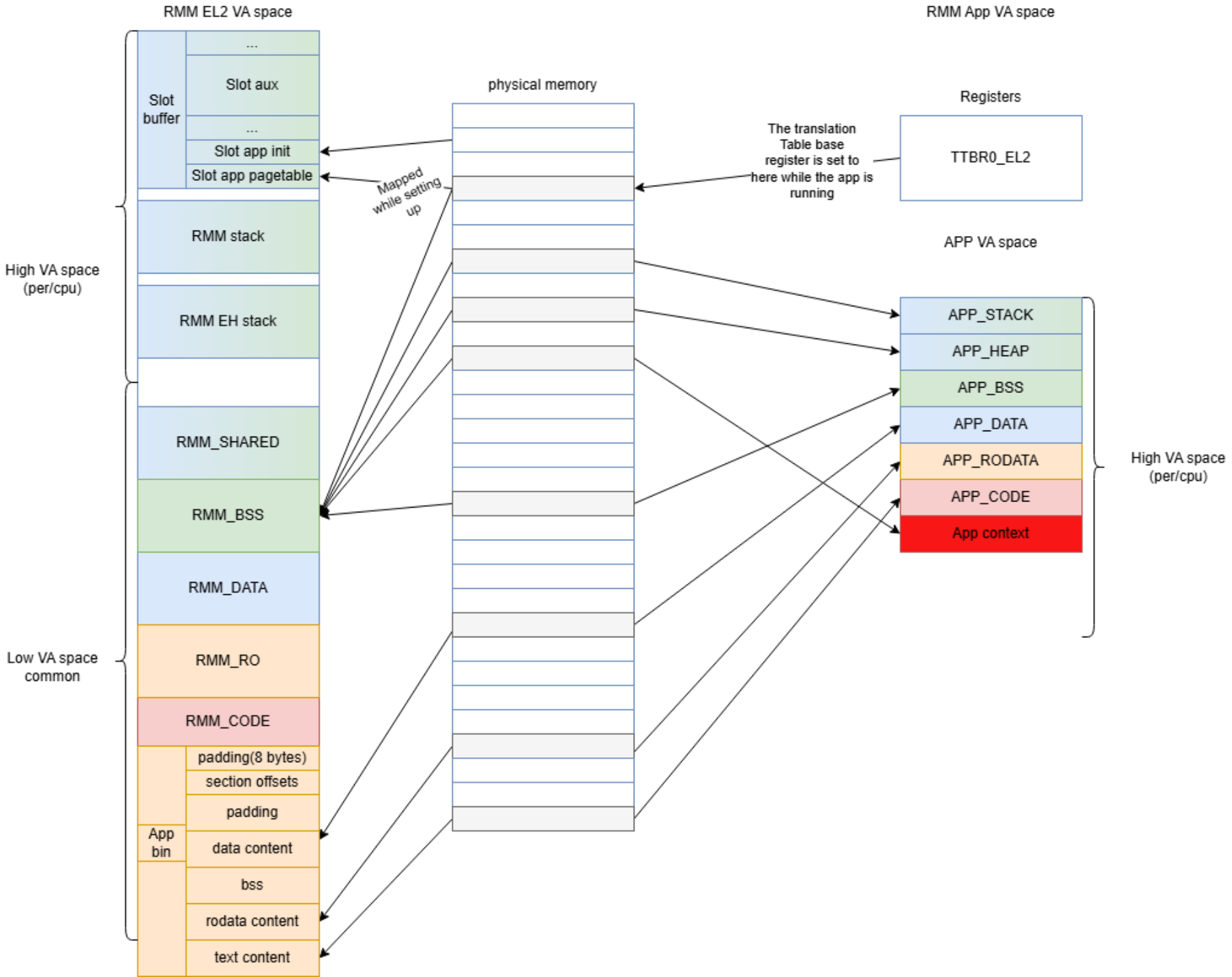
- RMM uses the FEAT_VHE extension to split the 64-bit VA space into two address spaces
  - Low VA range - Common mapping for all the CPUs
    - RMM Code
    - RMM RO
    - RMM RW
    - Shared pages with EL3
  - High VA range – Per CPU mapping
    - Slot buffer
    - CPUn stack
    - CPUn Exception stack
- For details see https://tf-rmm.readthedocs.io/en/latest/design/memory-management.html

RMM memory when app is running

- The low VA mapping is not changed, however translation is disabled
  - Using TCR_EL2.E0PD0
- On app entry, the High VA translation is reconfigured
  - As the High VA mapping is CPU specific, CPUs can independently enter/exit app as they execute RMM code
  - An app instance specific pagetable is set in TTBR1_EL2
    - Code
    - RO data
    - RW data
    - Stack
    - Heap
    - Shared page with RMM EL2
- The TTBR1_EL2 value configured for the app contains an app type specific ASID.
  - This helps reducing the need for TLB invalidation.

App memory setup

- Code and RO data pages are mapped from the pages where the App is loaded (alongside RMM) during boot
- RW data is common for all instances of an App
  - .data: pages are mapped from the pages where the App is loaded (alongside RMM) during boot
  - .bss: pages are allocated from RMM EL2 .bss
- heap, stack are unique for each app instance
  - Allocation depends on instance type, eg: for per REC/PDEV intances, pages are allocated from Aux granules and for per CPU app instances, pages are allocated from RMM .bss.
- Shared page is allocated per app instance in the current version


RMM EL2 VA space
RMM App VA space
physical memory
Registers
The translation Table base register is set to here while the app is running
TTBR0_EL2
Mapped while setting up
APP VA space
High VA space (per/cpu)
Low VA space common
Slot buffer
...
Slot aux
...
Slot app init
Slot app pagetable
RMM stack
RMM EH stack
RMM_SHARED
RMM_BSS
RMM_DATA
RMM_RO
RMM_CODE
padding(8 bytes)
section offsets
padding
App bin
data content
bss
rodata content
text content
APP_STACK
APP_HEAP
APP_BSS
APP_DATA
APP_RODATA
APP_CODE
App context
High VA space (per/cpu)

App Initialization and Execution

- The framework provides 2 function calls for RMM to call at R-EL2:
  - init_app_data
  - app_run
- init_app_data
  - Creates an app instance of a specified ID
    - Populates pagetable
    - Initialises stack, heap and shared page
    - Sets up memory page for storing application context when it is not scheduled
    - Initialises the app context to the app entrypoint
  - Initialises the output parameter app_data. This can be used later to call an app
  - The pages to be used as pagetable, stack, heap, shared page and storing context are passed as an array of physical addresses
- app_run
  - Activates the app context and enters the EL0 app for execution

Using apps (continued)

- App entry function at EL0 implements a loop that executes app functions
- The function app_run requires a function id, and 4 arguments for the function that is specified by the ID
  - Apps execute the function that is specified by the ID
  - When the function is complete, an SVC is executed with the ret value in the x0 register
  - The ret value is returned by app_run to the caller
- Apps yielding is not supported yet by the current implementation
- An app instance doesn’t need to be destroyed. If an app instance is no longer used, its per instance memory resources can safely be repurposed.

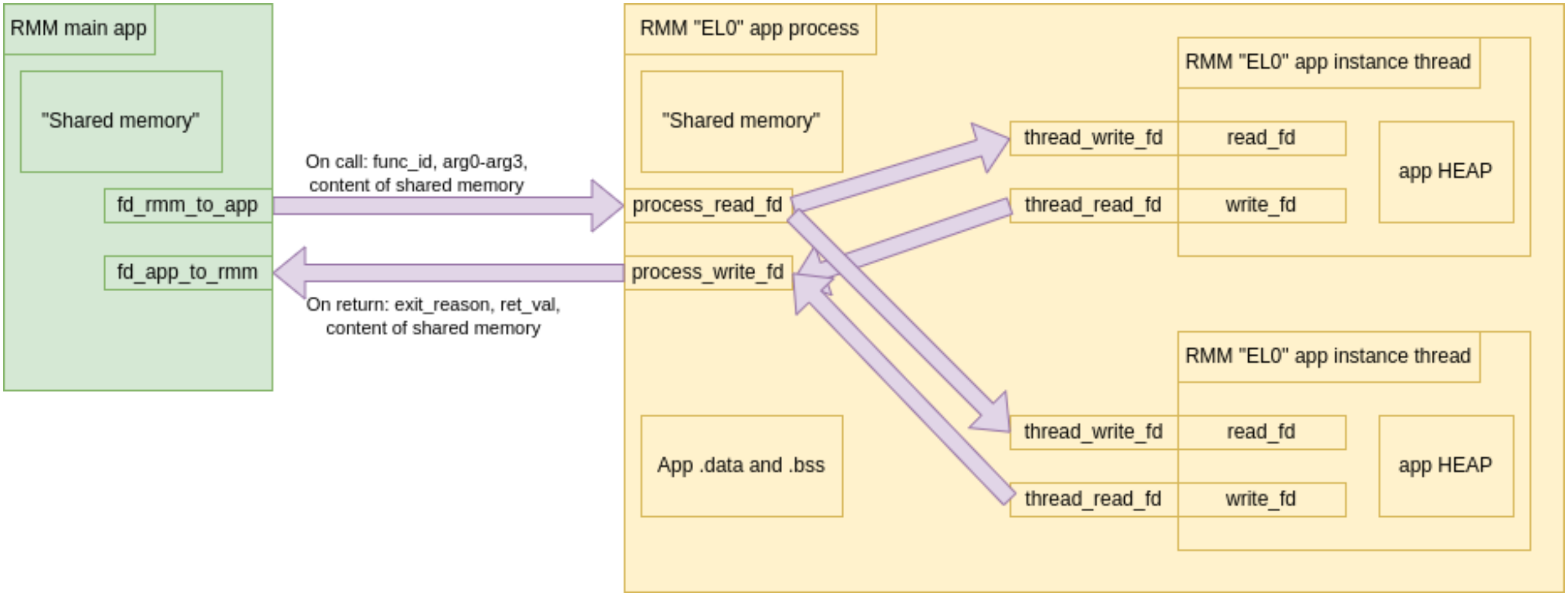
App service layer in RMM

- RMM provides services for apps (like logging).
- A service implementation is a function in EL2 RMM, which receives an app_data structure pointer, and 4 unsigned long parameters.
- The EL2 RMM <-> app shared page can be used to transfer data between the service and the app
- Services are expected to be reentrant and thread safe.
- Services are registered during RMM cold boot in a function pointer array. There is a single array in the system.
- The app that calls a service must do an SVC with a predefined immediate value. The index of the service to be executed is selected by the value of the X0 register.
- Parameters to the service are in x1-x4 registers

EL0 app in in case (RMM_ARCH=fake_host)

Fake host build

- It is possible to build RMM as a user space application, so the EL0 app framework should support this mode as well.
- The framework implementation is quite different compared to the FVP:
  - The applications are compiled as a standalone elf files.
  - The RMM core is compiled to the elf file rmm_core.elf.
  - The path to the app elf files are passed to rmm_core.elf as a command line parameter, along with the ID of the application.
  - The first time the main RMM process calls init_app_data the process is forked and the image of the requested app is loaded in the new process
  - For each init_app_data (including the first call) a new thread is created.
    - The current implementation leaks threads (i.e. threads are never destroyed)
  - The main thread in the app process is responsible for dispatching the app calls and returns between the main RMM process and the app thread.
  - Communication between the processes and the app main and app instance threads is done via pipes.
  - There is no shared memory between the main and the app processes, memory sharing is emulated by sending over the content of private “shared pages” between the core process and app threads.


RMM main app
"Shared memory"
fd_rmm_to_app
fd_app_to_rmm
On call: func_id, arg0-arg3,
content of shared memory
On return: exit_reason, ret_val,
content of shared memory
RMM "EL0" app process
"Shared memory"
process_read_fd
process_write_fd
App .data and .bss
RMM "EL0" app instance thread
thread_write_fd
read_fd
thread_read_fd
write_fd
app HEAP
RMM "EL0" app instance thread
thread_write_fd
read_fd
thread_read_fd
write_fd
app HEAP

arm

Merci
Danke
Gracias
Grazie
谢谢
ありがとう
Asante
Thank You
감사합니다
धन्यवाद
Kiitos
شكرًا
ধন্যবাদ
תודה
ధన్యవాదములు
Köszönöm

References

- TF-RMM: https://www.trustedfirmware.org/projects/tf-rmm
- Documentation: https://tf-rmm.readthedocs.io/en/latest/
- Memory management (No app): https://tf-rmm.readthedocs.io/en/latest/design/memory-management.html
- Fake_host build (No app): https://tf-rmm.readthedocs.io/en/latest/design/fake-host-architecture.html
- El0 app support patchstack: https://review.trustedfirmware.org/c/TF-RMM/tf-rmm/+/34007/7
  - Including:
    - Small refactors in RMM to support EL0 app framework
    - El0 app framework implementation + documentation
    - Refactored attestation library as an app