Fine-Grained Formal Specification and Analysis of Buddy Memory Allocation in Zephyr RTOS

Authors: Feng Zhang¹, Yongwang Zhao*,¹, Dianfu Ma¹, Wensheng Niu²

*. Corresponding Author
1. School of Computer Science and Engineering, Beihang Univerisity, China
2. Aeronautical Computing Technique Research Institute, Xi’an, China
Contents

1. Introduction
2. Buddy Memory Allocation Algorithm in Zephyr
3. Fine-Grained Formal Specification in Isabelle/HOL
4. Formal Proof
5. Results and Discussions
6. Conclusions
1. Introduction - Abstract

- Memory management (MM) is a critical component of OS
- Bugs in MM may crash OS or the whole critical system
- This paper presents a case study of formal verification on the buddy memory allocation component of the Zephyr RTOS:
  - Provide Fine-Grained formal specification in Isabelle/HOL
  - Conduct Formal proof using the interactive prover in Isabelle
  - Find two flaws in the C code when executing sequentially
1. Introduction – Research Status

➢ Verification of the TLSF algorithm in Event-B:
  • Only verifies an abstract specification at the requirement level
  • Not check consistency between elements in the data structure

➢ seL4 pushes the memory allocation outside of the kernel

➢ Yu et al. introduce a low-level language CAP (certified assembly programming) in Coq
  • Build certified programs
  • Present a certified library for dynamic storage allocation
  • Not a kernel’s component but a certified memory library
  • 75 lines C code
1. Introduction – summary

- We create a fine-grained formal specification:
  - All the elements of the data structure
  - All the operations (initialization, allocation and release)
  - System clocks and simple kernel scheduling
  - The execution of memory allocation is preemptive

- We concentrate in five types of critical properties:
  - Invariants
  - Correctness of doubly linked lists
  - Functional correctness of events
  - Conformity of event specifications to kernel requirements
  - Livelock-free of the system specification.
Contents

1. Introduction
2. Buddy Memory Allocation Algorithm in Zephyr
3. Fine-Grained Formal Specification in Isabell/HOL
4. Formal Proof
5. Results and Discussions
6. Conclusions
2.1 – Zephyr Project

- Zephyr Project is a Linux Foundation Project
- Be perfect for building simple connected sensors:
  - up to modems and small IoT wireless gateways
  - Built with safety and security in mind
  - Cross-architecture with growing developer tool support
  - Complete, fully integrated, highly configurable, modular for flexibility, better than roll-your-own
  - Product development ready
  - Permissively licensed
2.2 – Zephyr OS Kernel

- Derived from **Wind River**’s commercial **Microkernel Profile**

- **Microkernel Profile** has evolved over 20 years from DSP RTOS technology known as **Virtuoso**

- Used in several commercial applications:
  - satellites, military command and control communications, radar, telecommunications and image processing
  - successful **Philae** Landing on Comet Churyumov–Gerasimenko and the accompanying **Rosetta Orbiter**
2.2 – Buddy Memory Allocation Algorithm in Zephyr Kernel

- (1) Pool and block Initialization
  - only be defined and initialized at compile time

- (2) Block Allocation
  - Quad-Partitioning: iteratively partitioning larger blocks into smaller quad-ones

- (3) Block Release
  - Immediately, automatically, and recursively combining smaller blocks into bigger ones
2.2 – Buddy Memory Allocation Algorithm in Zephyr Kernel

Algorithm 1 pool_alloc (p, size, block)

Input: p: the request block: information
Output: ZERO: success

1: alloc_l = -1; free_l
2: for i=0; i < p.n_l
3: if block size at alloc_l == i
4: break
5: end if
6: alloc_l = i
7: if ! level_empty
8: free_l = i
9: end if
10: end for
11: if alloc_l < 0 ||
12: return NOMEM
13: end if
14: blk ← address of blk
15: if ! blk then
16: return EAGAIN
17: end if
18: for from_l = free_l,
19: partitioning bl
20: blk ← address
21: end for
22: block ← (blk, p, size)
23: return 0

Algorithm 2 k_mem_pool_alloc (p, size, block, timeout)

Input: p: the request block: information
Output: ZERO: success

1: if size > p.max
2: return NOMEM
3: else
4: if timeout >
5: end = curr
6: while (1) do
7: ret = pool.
8: if ret == 0
9: return
10: free_block (p, level-1, lsize, bn/4)
11: SCHEDULE
12: if timeout <= 0 then
13: break
14: end if
15: end if
26: block ← (blk, p, size)
27: return TIMEOUT
28: end if

Algorithm 3 free_block (p, level, lsize, bn)

Input: p: the released pool; level: the released level number
Output: void

1: set_bit(p, level, bn)
2: if level > 0 && partner of bn are all free then
3: for i=0; i < 4; i++ do
4: clear_bit(p, level, bn+i)
5: if (bn&~3)+i != bn then
6: remove block ((bn&~3)+i) from its free list
7: end if
8: end for
9: free_block (p, level-1, lsize, bn/4)
10: return
11: SCHEDULE
12: if timeout <= 0 then
13: break
14: end if
15: end if
16: block ← (blk, p, size)
17: return
18: end while
19: return TIMEOUT
20: end if
Contents

1. Introduction
2. Buddy Memory Allocation Algorithm
3. Fine-Grained Formal Specification in Isabelle/HOL
4. Formal Proof
5. Results and Discussions
6. Conclusions
A. State Machine

- The state is defined as a record `StateD`
- the initial state $s_0$
- state-transition functions $\varphi$

**Definition 1:** State machine of the buddy memory allocation Component $\mathcal{M} = \langle S, \mathcal{E}, \varphi, s_0 \rangle$ is a tuple, where $S$ is the state space, $\mathcal{E}$ is a set of event labels, $s_0 \in S$ is the initial state, and $\varphi: \mathcal{E} \rightarrow \mathbb{P}(S \times S)$ is a set of state-transition functions.
B. Data Structure

```c
struct k_mem_block_id {
    u32_t pool : 8;
    u32_t level : 4;
    u32_t block : 20;
};

struct k_mem_block {
    void *data;
    struct k_mem_block_id id;
};

struct k_mem_pool_lvl {
    union {
        u32_t *bits_p;
        u32_t bits;
    }
    sys_dlist_t free_list;
};

struct k_mem_pool {
    void *buf;
    size_t max_sz;
    u16_t n_max;
    u8_t n_levels;
    u8_t max_inline_level;
    struct k_mem_pool_lvl *levels;
    wait_q_t wait_q;
};

type_synonym struct_block_id = "pool_num × level_num × block_num"

type_synonym struct_block = "struct_block_id × addr"

type_synonym BlockD = "(struct_block_id × block_state_type) tree"

type_synonym struct_pool_lvl = "bitMap × ref_freelist"

record PoolD = l0blist :: "BlockD list"
    name :: string
    max_sz :: nat
    nmax :: nat
    n_levels :: nat
    max_inline_level :: max_inline_lsz
    levels :: "struct_pool_lvl list"

record heap = ref5 :: "ref set"
    addr5 :: "addr set"
    addr2bid :: "addr → struct_block_id"
    ref2bid :: "sys_dnode_t → struct_block_id"
    head_next :: "ref → ref"
    tail_prev :: "ref → ref"

record StateD = globals :: heap
    poolsD :: "PoolD list"
    curD :: "Thread option"
    irq :: bool
    tickD :: nat
    t_stateD :: "Thread ⇒ thread_state_type"
    waitqD :: "Thread → pool_num"
    alloc_context ::
        "Thread → (pool_num × request_size × timeout × end_time)"
```

3 – Fine-Grained Formal Specification
3 – Fine-Grained Formal Specification

B. Data Structure

Free Lists of multiple levels

BitMaps of multiple levels

Abstract Trees

A: allocated
D: divided
F: free
C. Event Specification

- system behaviors based on Zephyr characteristics
  - system clocks $\text{time\_tick}$
  - the thread scheduling $\text{schedule}$

- actions operated on memory pools and blocks
  - pool and block initializations
  - block allocations
  - block release
### C. Event Specification

- **system behavior**
  - system clocks
  - the thread scheduler

- **actions operated**
  - pool and block initializations
  - block allocations
  - block release

---

**Algorithm 2** \( k_{\text{mem pool alloc}}(p, size, block, timeout) \)

**Input:**
- \( p \): the requested pool
- \( size \): the requested size
- \( block \): information of the allocated block
- \( \text{timeout} \): no_wait, forever or timed_wait

**Output:**
- \( \text{ZERO} \): successful allocation
- \( \text{NOMEM} \): no memory
- \( \text{TIMEOUT} \): timeout

```plaintext
1: if \( size > p.\text{max sz} \) then
2:  return \( \text{NOMEM} \)
3: else
4:  if \( \text{timeout} > 0 \) then
5:    end = current_time + \text{timeout}
6:  end if
7:  while (1) do
8:    \( \text{ret} = \text{pool alloc}(p, size, block) \)
9:    if \( \text{ret} == 0 \) || \( \text{timeout} == \text{no_wait} \) then
10:      return \( \text{ret} \)
11:    end if
12:    \text{SCHEDULE}();
13:    if \( \text{timeout} != \text{forever} \) then
14:      \( \text{timeout} = \text{end} - \text{current time} \)
15:    if \( \text{timeout} <= 0 \) then
16:      break
17:    end if
18:  end if
19: end while
20: return \( \text{TIMEOUT} \)
21: end if
```
3 – Fine-Grained Formal Specification

C. Event Specification
3 – Fine-Grained Formal Specification

D. State Space

```plaintext
type_synonym Trace = “StateD list”
inductive_set TraceSpace :: “Trace set”

definition “reachable s t =
    (∃ts ∈ TraceSpace. ts!0=s ∧ last ts = t)”

definition “ReachStates ≡ {s. reachable s₀ s}”
```
Contents

1. Introduction
2. Buddy Memory Allocation Algorithm in Zephyr
3. Fine-Grained Formal Specification in Isabelle/HOL
4. Formal Proof
5. Results and Discussions
6. Conclusions
4 – Formal Proof

4.1 Invariants - Consistency of Data Structure

- **bitMap_freelistS s** specifies the consistency between bit_maps and free lists.

- **bitMap_treeS s** specifies the consistency between bit_maps and abstract trees.

```
definition Inv_Bitmap_freelist_tree s ≡
  bitMap_freelistS s ∧ bitMap_treeS s
```

**Table 3** The Specification *bitMap_freelistS*

```
definition “bitMap_freelist pn p h ≡
  ∀ln<n_levels p. ∀bn< (nmax p)*(4^ln).
  let frel_ln = snd((levels p)!ln);
  ndref = (case (get_refBYbid (ref2bid h) (pn,ln,bn)) of Some ndr ⇒ ndr);
  bmGet = get_bit_ptr (fst((levels p)!ln)) bn in
  fst bmGet ≠ snd bmGet ↔ nodeINfreelist (head_next h) frel_ln ndref ≠ None"
definition “bitMap_freelistS s ≡ ∀pn<length(poolsD s).
  let p = (poolsD s)!pn; h = globals s in bitMap_freelist pn p h”
```
4 – Formal Proof

4.2 Correctness of Doubly Linked Lists

- The pointer in C is specified as a `ref` in Isabelle
- `ref = (UNIV::nat set)`
- `head_next :: "ref => ref“`
- `tail_prev :: "ref => ref”`

```c
struct _dnode {
    union {
        struct _dnode *head; /* ptr to head of list (sys_dlist_t) */
        struct _dnode *next; /* ptr to next node (sys_dnode_t) */
    }
    union {
        struct _dnode *tail; /* ptr to tail of list (sys_dlist_t) */
        struct _dnode *prev; /* ptr to previous node (sys_dnode_t) */
    }
};
typedef struct _dnode sys_dlist_t;
typedef struct _dnode sys_dnode_t;
```
4.2 Correctness of Doubly Linked Lists

- Length of a dilist
- Validity of a node
- Validity of a dlist
- Validity of appending actions
4 – Formal Proof

4.3 Functional Correctness of Events

- \{P\} C \{Q\}
- Our specifications are all total correctness specifications
- terminations are ensured by using the `primrec, fun, function` and `definition`

**Lemma 11. correctness of function allocL_freeL**

\[
\begin{align*}
\{ & \text{lfsz}=\text{n\_levels} \ p \land \text{n\_levels} \ p>0 \land \text{max\_sz} \ p>0 \} \\
\text{freel} = \text{snd}((\text{levels} \ p)!(\text{nat} (\text{snd} \ \text{alfl})))) \ || \\
\text{alfl} = \text{allocL\_freeL} \ p \ h \ \text{rsz \ (-1,-1)} \ \text{lfsz} \\
\{ & \text{fst} \ \text{alfl}>-1 \land \text{snd} \ \text{alfl}>-1 \rightarrow \\
\text{(fst} \ \text{alfl} \geq \text{snd} \ \text{alfl} \land \text{fst} \ \text{alfl} \geq 0 \land \text{snd} \ \text{alfl} \geq 0 \land \\
\text{nat(fst} \ \text{alfl}) < \text{n\_levels} \ p \land \text{nat(snd} \ \text{alfl}) < \text{n\_levels} \ p \land \\
\neg \ \text{level\_empty \ (head\_next} \ h \ \text{freel)}) \ |}
\end{align*}
\]
4 – Formal Proof

➢ 4.4 Conformity of Event Specifications to Kernel Requirements

Lemma ExFreeNdPartners:

∀s pn bn. let pn_s = (poolsD s)!pn; ln_i = n_levels pn_s - i; lvl_s s = levels pn_s in
poolsD s ≠ [] ∧ pn<length(poolsD s) ∧ 0<i ∧ i≤n_levels pn_s ∧ bn<(nmax pn_s)*(4^ln_i) ∧ reachableD sOD s ∧ fst (lookup_tree ((l0blist pn_s)!(bn div (4^ln_i))) (pn,ln_i,bn)) →
(if ¬partner_bitsLn lvl_s s ln_i bn then
  ∃t. let pn_t = (poolsD t)!pn in reachableD s t ∧
  (∀ln'<n_levels pn_t.∀bn'<(nmax pn_t)*(4^ln').
    let blk_n_t' = lookup_tree ((l0blist pn_t)!(bn' div (4^ln'))) (pn,ln',bn');
    blk_n_s' = lookup_tree ((l0blist pn_s)!(bn' div (4^ln'))) (pn,ln',bn') in
    if ln'=ln_i ∧ bn'=bn then fst blk_n_t' ∧ isLeaf(snd blk_n_t') ∧ bstate (snd blk_n_t')=FREE
  else blk_n_t'👀blk_n_s')
else ∃j. let ln_j = ln_i - j;
    pn_t = (poolsD t)!pn in reachableD s t ∧ 0<j ∧ j≤ln_i ∧
    (∀ln'<n_levels pn_t.∀bn'<(nmax pn_t)*(4^ln').
      let blk_n_t' = lookup_tree ((l0blist pn_t)!(bn' div (4^ln'))) (pn,ln',bn');
      blk_n_s' = lookup_tree ((l0blist pn_s)!(bn' div (4^ln'))) (pn,ln',bn');
      bn_ln'_l = (bn div (4^j))*(4^(ln_i'-ln_j));
      bn_ln'_r = (bn div (4^j) + 1)*(4^(ln_i'-ln_j)) in
      if ln'=ln_j ∧ bn'=bn then
        fst blk_n_t' ∧ isLeaf(snd blk_n_t') ∧ bstate (snd blk_n_t')=FREE
      else if ln'>ln_j ∧ (bn'>bn ln'_l ∧ bn'<bn ln'_r) then ¬ fst blk_n_t'
else blk_n_t'=blk_n_s'))
Contents

1. Introduction
2. Buddy Memory Allocation Algorithm in Zephyr
3. Fine-Grained Formal Specification in Isabell/HOL
4. Formal Proof
5. Results and Discussions
6. Conclusions
5 – Results and Discussions

A. Evaluation

- 600 lines C
- 800 lines specification:
  109 functions/definitions
  12 primary events
- 9400 lines proof: 338 lemmas
5 – Results and Discussions

➢ B. Results of formal analysis: fine two flaws

☐ Return code not conform to the kernel requirement

☐ Application thread will fall into live lock.
B. Results of formal analysis: fine two flaws

- Return code not conform to the kernel requirement
- Application thread will fall into live lock.

Fine-Grained Formal Specification and Analysis of Buddy Memory Allocation in Zephyr RTOS

```c
int k_mem_pool_alloc(struct k_mem_pool *p, struct k_mem_block *block,
                   size_t size, s32_t timeout)
{
    int ret, key;
    s64_t end = 0;

    __ASSERT(!(__is_in_isr() && timeout != K_NO_WAIT), "!");

    if (timeout > 0) {
        end = _tick_get() + _ms_to_ticks(timeout);
    }

    while (1) {
        ret = pool_alloc(p, block, size);

        if (ret == 0 || timeout == K_NO_WAIT ||
            ret == -EAGAIN || (ret && ret != -ENOMEM)) {
            return ret;
        }

        key = irq_lock();
        _PEND_CURRENT_THREAD(&p->wait_q, timeout);
        _Swap(key);

        if (timeout != K_FOREVER) {
            timeout = end - _tick_get();

            if (timeout < 0) {
                break;
            }
        } else { /* end while */

        return -EAGAIN;
    }
} /* end k_mem_pool_alloc */
```
Contents

1. Introduction
2. Buddy Memory Allocation Algorithm in Zephyr
3. Fine-Grained Formal Specification in Isabelle/HOL
4. Formal Proof
5. Results and Discussions
6. Conclusions
6 – Conclusions

➢ We will perform formal analysis on the concurrent characteristics of the OS kernel

➢ For about 600 lines of C, our work consists of about 10200 lines of Isabelle

➢ Find two flaws in C code when executing sequentially
Thank you