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

北京航空航天大学计算机新技术研究所
The Institute of Advanced Computing Technology

ACT

[redacted]

Authors: **Feng Zhang[1], Yongwang Zhao[*,1], Dianfu Ma[1], Wensheng Niu[2]**

*. Corresponding Author

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

北京航空航天大學
BEIHANG UNIVERSITY

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

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 requeste
block: informatio
Output: *ZERO*: succe

```
alloc_l = -1; free_
for i=0; i< p.n_l
    if block size at
        break
    end if
    alloc_l = i
    if ! level_empt
        free_l = i
    end if
end for
if alloc_1 < 0 || f
    return NOMEI
end if
blk ← address of
if ! blk then
    return EAGAI
end if
for from_l = free_
    partitioning bl
    blk ← addres
end for
block ← (blk, p
return 0
```

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

Input: *p*: the requeste
block: informatio
timeout: no_wai
Output: *ZERO*: succe

```
if size > p.max
    return NOMEM
else
    if timeout >
        end = curr
    end if
    while (1) do
        ret = pool_
        if ret == 0
            return r
        end if
        SCHEDUL
        if timeou
            timeo
            if timeout <= 0 then
                break
            end if
        end if
    end while
    return TIMEOUT
end if
```

Algorithm 3 free_block (*p*, *level*, *lsizes*, *bn*)

Input: *p*: the released pool; *level*: the released level number
lsizes: an array contains each level's block size; *bn*: the released block number
Output: *void*

```
set_bit(p, level, bn)
if level > 0 && partner of bn are all free then
    for i=0; i<4; i++ do
        clear_bit(p, level, bn+i)
        if (bn&~3)+i ≠ bn then
            remove block ((bn&~3)+i) from its free list
        end if
    end for
    free_block (p, level-1, lsizes, bn/4)
    return
end if
append the the released block (p, level, bn) into its free list
```

Contents

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

3 – Fine-Grained Formal Specification

A. State Machine

- The state is defined as a **record** *StateD*
- the initial state s_0
- state-transition functions $\boldsymbol{\varphi}$

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

3 – Fine-Grained Formal Specification

- B. Data Structure

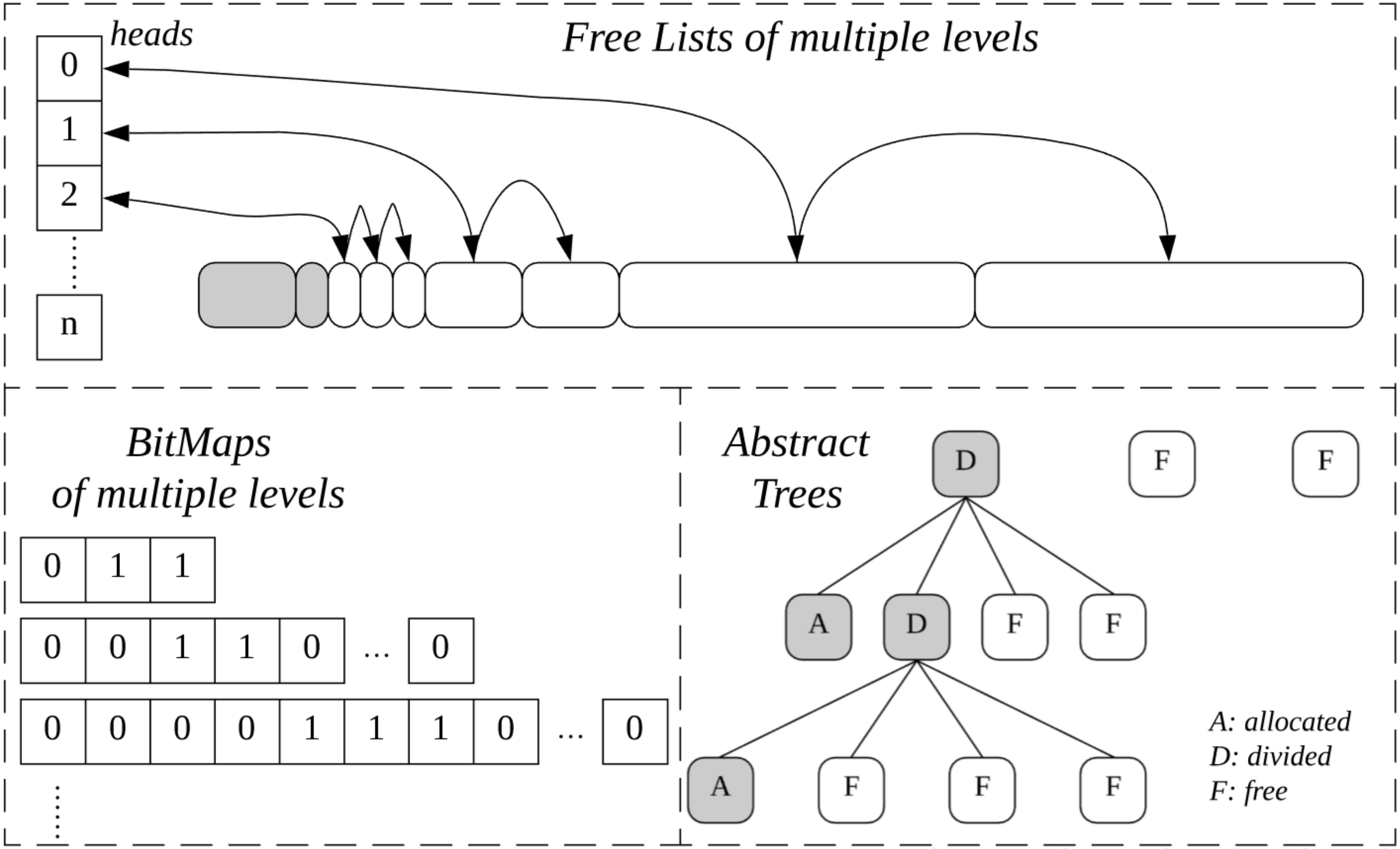
```c
struct k_mem_block_id {
        u32_t pool : 8;
        u32_t level : 4;
        u32_t block : 20; };
struct k_mem_block {
        void *data;
        struct k_mem_block_id id; };
struct k_mem_pool_lvl {
        union {
                u32_t *bits_p;
                u32_t bits;
        };
        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 = refS :: "ref set"
              addrS  :: "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

- B. Data Structure



3 – Fine-Grained Formal Specification

- C. Event Specification

  - system behaviors based on Zephyr characteristics
    - system clocks *time_tick*
    - the thread scheduling *schedule*
  - actions operated on memory pools and blocks
    - pool and block initializations
    - block allocations
    - block release

3 – Fine-Grained Formal Specification

C. Event Specification

- system beha
  - system clocks a
  - the thread sche
- actions opera
  - pool and block
  - block allocatior
  - block release

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

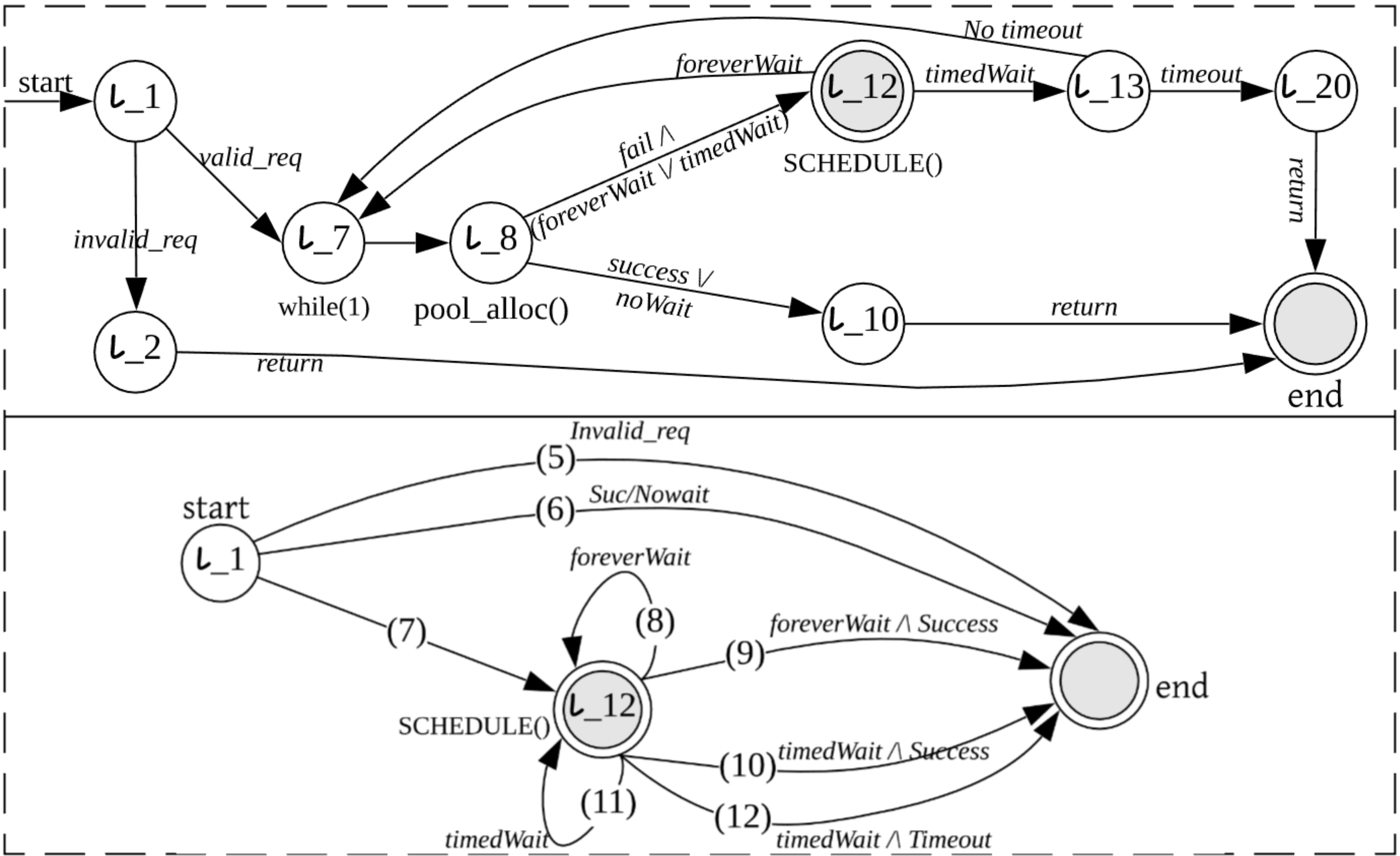
Input: p: the requested pool; $size$: the requested size
$block$: information of the allocated block;
$timeout$: no_wait, forever or timed_wait

Output: *ZERO*: successful allocation; *NOMEM*: no memory; *TIMEOUT*: timeout

```
if size > p.max_sz then
    return NOMEM
else
    if timeout > 0 then
        end = current_time + timeout
    end if
    while (1) do
        ret = pool_alloc(p, size, block)
        if ret == 0 || timeout == no_wait then
            return ret
        end if
        SCHEDULE();
        if timeout != forever then
            timeout = end - current_time
            if timeout <= 0 then
                break
            end if
        end if
    end while
    return TIMEOUT
end if
```

3 – Fine-Grained Formal Specification

➢ C. Event Specification



3 – Fine-Grained Formal Specification

- D. State Space

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

definition “reachable s t =
$(\exists ts \in \text{TraceSpace}.\ ts!0 = s \wedge \text{last } ts = t)$”

definition “ReachStates $\equiv \{s.\ \text{reachable } s_0\ s\}$”

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

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 freel_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) freel_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"

```
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 –Formal Proof

➢ 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*

{| lsz=n_levels p ∧ n_levels p>0 ∧ max_sz p>0);
freel = snd((levels p)!(nat (snd alfl))) |}
alfl = **allocL_freeL** p h rsz (-1,-1) lsz
{| fst alfl>-1 ∧ snd alfl>-1 ⟶
(fst alfl ≥ snd alfl ∧ fst alfl≥0 ∧ snd alfl≥0 ∧
nat(fst alfl) < n_levels p ∧ nat(snd alfl) < n_levels p ∧
¬ level_empty (head_next h) freel) |}

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; lvls_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 s0D s ∧ fst (lookup_tree ((l0blist pn_s)!(bn div (4^ln_i))) (pn,ln_i,bn)) ⟶
  (if ¬partner_bitsLn lvls_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 ∃t 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'-ln_j));
                    bn_ln'_r =  (bn div (4^j) + 1)*(4^(ln'-ln_j)) in
                  if ln'=ln_j ∧ bn'=bn div (4^j) 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

C code	Isabelle Code				
	Specification		Proof		
.c	function/ definition	LOC	theorem/ lemma	LOP	total lines in Isabelle
600	109	800	338	9400	10200

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.

5 .

➢ B. Results of

- Return code
- Application

```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;
            }
        }
    } « end while 1 »

    return -EAGAIN;
} « end k_mem_pool_alloc »
```

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

6 – Conclusions

- We will perform formal analysis on the concurrent characteristics of the OS kernel
- For about 600lines C, our work consists of about 10200 lines of Isabelle
- Find two flaws in C code when executing sequentially

Thank you