Rely-guarantee Reasoning about Concurrent Memory Management in Zephyr RTOS [⋆]

Yongwang Zhao (✉)[1,2] and David Sanán[3]

[1] School of Computer Science and Engineering, Beihang University, Beijing, China
[2] Beijing Advanced Innovation Center for Big Data and Brain Computing, Beihang University, Beijing, China
[3] School of Computer Science and Engineering, Nanyang Technological University, Singapore
Email: zhaoyw@buaa.edu.cn

Abstract. Formal verification of concurrent operating systems (OSs) is challenging, and in particular the verification of the dynamic memory management due to its complex data structures and allocation algorithm. Up to our knowledge, this paper presents the first formal specification and mechanized proof of a concurrent buddy memory allocation for a real-world OS. We develop a fine-grained formal specification of the buddy memory management in Zephyr RTOS. To ease validation of the specification and the source code, the provided specification closely follows the C code. Then, we use the rely-guarantee technique to conduct the compositional verification of functional correctness and invariant preservation. During the formal verification, we found three bugs in the C code of Zephyr.

1 Introduction

The operating system (OS) is a fundamental component of critical systems. Thus, correctness and reliability of systems highly depend on the system's underlying OS. As a key functionality of OSs, the memory management provides ways to dynamically allocate portions of memory to programs at their request, and to free them for reuse when no longer needed. Since program variables and data are stored in the allocated memory, an incorrect specification and implementation of the memory management may lead to system crashes or exploitable attacks on the whole system. RTOS are frequently deployed on critical systems, making formal verification of RTOS necessary to ensure their reliability. One of the state of the art RTOS is Zephyr RTOS [1], a Linux Foundation project. Zephyr is an open source RTOS for connected, resource-constrained devices, and built with security and safety design in mind. Zephyr uses a buddy memory allocation algorithm optimized for RTOS, and that allows multiple threads to concurrently manipulate shared memory pools with fine-grained locking.

Formal verification of the concurrent memory management in Zephyr is a challenging work. (1) To achieve high performance, data structures and algorithms in Zephyr

[⋆] This work has been supported in part by the National Natural Science Foundation of China (NSFC) under the Grant No.61872016, and the National Satellite of Excellence in Trustworthy Software Systems and the Award No. NRF2014NCR-NCR001-30, funded by NRF Singapore under National Cyber-security R&D (NCR) programme.

are laid out in a complex manner. The buddy memory allocation can split large blocks into smaller ones, allowing blocks of different sizes to be allocated and released efficiently while limiting memory fragmentation concerns. Seeking performance, Zephyr uses a multi-level structure where each level has a bitmap and a linked list of free memory blocks. The levels of bitmaps actually form a forest of quad trees of bits. Memory addresses are used as a reference to memory blocks, so the algorithm has to deal with address alignment and computation concerning the block size at each level, increasing the complexity of its verification. (2) A complex algorithm and data structures imply as well complex invariants that the formal model must preserve. These invariants have to guarantee the well-shaped bitmaps and their consistency to free lists. To prevent memory leaks and block overlapping, a precise reasoning shall keep track of both numerical and shape properties. (3) Thread preemption and fine-grained locking make the kernel execution of memory services to be concurrent.

In this paper, we apply the rely-guarantee reasoning technique to the concurrent buddy memory management in Zephyr. This work uses π-Core, a rely-guarantee framework for the specification and verification of concurrent reactive systems. π-Core introduces a concurrent imperative system specification language driven by "events" that supports reactive semantics of interrupt handlers (e.g. kernel services, scheduler) in OSs, and thus makes the formal specification of Zephyr simpler. The language embeds Isabelle/HOL data types and functions, therefore it is as rich as the own Isabelle/HOL. π-Core concurrent constructs allow the specification of Zephyr multi-thread interleaving, fine-grained locking, and thread preemption. Compositionality of rely-guarantee makes feasible to prove the functional correctness of Zephyr and invariants over its data structures. The formal specification and proofs are developed in Isabelle/HOL. They are available at https://lvpgroup.github.io/picore/.

We first analyze the structural properties of memory pools in Zephyr (Section 3). The properties clarify the constraints and consistency of quad trees, free block lists, memory pool configuration, and waiting threads. All of them are defined as invariants for which its preservation under the execution of services is formally verified. From the well-shaped properties of quad trees, we can derive a critical property to prevent memory leaks, i.e., memory blocks cover the whole memory address of the pool, but not overlap each other.

Together with the formal verification of Zephyr, we aim at the highest evaluation assurance level (EAL 7) of Common Criteria (CC) [2], which was declared this year as the candidate standard for security certification by the Zephyr project. Therefore, we develop a fine-grained low level formal specification of a buddy memory management (Section 4). The specification has a line-to-line correspondence with the Zephyr C code, and thus is able to do the *code-to-spec* review required by the EAL 7 evaluation, covering all the data structures and imperative statements present in the implementation.

We enforce the formal verification of functional correctness and invariant preservation by using a rely-guarantee proof system (Section 5), which supports total correctness for loops where fairness does not need to be considered. The formal verification revealed three bugs in the C code: an incorrect block split, an incorrect return from the kernel services, and non-termination of a loop (Section 6). Two of them are critical and have been repaired in the latest release of Zephyr. The third bug causes nontermination of the allocation service when trying to allocate a block of a larger size than the maximum allowed.

Related work. (1) Memory models [17] provide the necessary abstraction to separate the behaviour of a program from the behaviour of the memory it reads and writes. There are many formalizations of memory models in the literature, e.g., [14,19,10,21,15], where some of them only create an abstract specification of the services for memory allocation and release [10,21,15]. (2) Formal verification of OS memory management has been studied in CertiKOS[20,11], seL4 [13,12], Verisoft [3], and in the hypervisors from [4,5], where only the works in [11,4] consider concurrency. Comparing to buddy memory allocation, the data structures and algorithms verified in [11] are relatively simpler, without block split/coalescence and multiple levels of free lists and bitmaps. [4] only considers virtual mapping but not allocation or deallocation of memory areas. (3) Algorithms and implementations of dynamic memory allocation have been formally specified and verified in an extensive number of works [23,7,16,18,8,9]. However, the buddy memory allocation is only studied in [9], which does not consider concrete data structures (e.g. bitmaps) and concurrency. To the best of our knowledge, this paper presents the first formal specification and mechanized proof for a concurrent buddy memory allocation of a realistic operating system.

2 Concurrent Memory Management in Zephyr RTOS

In Zephyr, a memory pool is a kernel object that allows memory blocks to be dynamically allocated, from a designated memory region, and released back into the pool. Its definition in the C code is shown as follows. A memory pool's buffer ($*buf$) is an n_max-size array of blocks of max_sz bytes at level 0, with no wasted space between them. The size of the buffer is thus $n\_max \times max\_sz$ bytes long. Zephyr tries to accomplish a memory request by splitting available blocks into smaller ones fitting as best as possible the requested size. Each "level 0" block is a quad-block that can be split into four smaller "level 1" blocks of equal size. Likewise, each level 1 block is itself a quad-block that can be split again. At each level, the four smaller blocks become *buddies* or *partners* to each other. The block size at level l is thus $max\_sz/4^l$.

```
struct k_mem_block_id {
  u32_t pool : 8;
  u32_t level : 4;
  u32_t block : 20;
};
struct k_mem_pool_lvl {
  union {
    u32_t *bits_p;
    u32_t bits;
  };
  sys_dlist_t free_list;
};
```

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

The pool is initially configured with the parameters n_max and max_sz, together with a third parameter min_sz. min_sz defines the minimum size for an allocated block and must be at least $4 \times X$ ($X > 0$) bytes long. Memory pool blocks are recursively split into quarters until blocks of the minimum size are obtained, at which point no further split can occur. The depth at which min_sz blocks are allocated is n_levels and satisfies that $n\_max = min\_sz \times 4^{n\_levels}$.

Every memory block is composed of a $level$; a $block$ index within the level, ranging from 0 to $(n\_max \times 4^{level}) - 1$; and the $data$ representing the block start address, which is equal to $buf + (max\_sz/4^{level}) \times block$. We use a tuple $(level, block)$ to uniquely represent a block within a pool p.

A memory pool keeps track of how its buffer space has been split using a linked list *free_list* with the start address of the free blocks in each level. To improve the performance of coalescing partner blocks, memory pools maintain a bitmap at each level to indicate the allocation status of each block in the level. This structure is represented by a C union of an integer *bits* and an array *bits_p*. The implementation can allocate the bitmaps at levels smaller than $max\_inlinle\_levels$ using only an integer *bits*. However, the number of blocks in levels higher than $max\_inlinle\_levels$ make necessary to allocate the bitmap information using the array *bits_map*. In such a design, the levels of bitmaps actually form a forest of complete quad trees. The bit i in the bitmap of level j is set to 1 for the block (i, j) iff it is a free block, i.e. it is in the free list at level i. Otherwise the bitmap for such block is set to 0.

Zephyr provides two kernel services *k_mem_pool_alloc* and *k_mem_pool_free*, for memory allocation and release respectively. The main part of the C code of *k_mem_pool_alloc* is shown in Fig. 1. When an application requests for a memory block, Zephyr first computes $alloc\_l$ and $free\_l$. $alloc\_l$ is the level with the size of the smallest block that will satisfy the request, and $free\_l$, with $free\_l \leqslant alloc\_l$, is the lowest level where there are free memory blocks. Since the services are concurrent, when the service tries to allocate a free block *blk* from level $free\_l$ (Line 8), blocks at that level may be allocated or merged into a bigger block by other concurrent threads. In such case the service will back out (Line 9) and tell the main function *k_mem_pool_alloc* to retry. If blk is successfully locked for allocation, then it is broken down to level $alloc\_l$ (Lines 11 - 14). The allocation service *k_mem_pool_alloc* supports a *timeout* parameter to allow threads waiting for that pool for a period of time when the call does not succeed. If the allocation fails (Line 24) and the timeout is not *K_NO_WAIT*, the thread is suspended (Line 30) in a linked list *wait_q* and the context is switched to another thread (Line 31).

Interruptions are always enabled in both services with the exception of the code for the functions *alloc_block* and *break_block*, which invoke *irq_lock* and *irq_unlock* to respectively enable and disable interruptions. Similar to *k_mem_pool_alloc*, the execution of *k_mem_pool_free* is interruptable too.

3 Defining Structures and Properties of Buddy Memory Pools

As a specification at design level, we use abstract data types to represent the complete structure of memory pools. We use an abstract reference *ref* in Isabelle to define pointers to memory pools. Starting addresses of memory blocks, memory pools, and unsigned integers in the implementation are defined as *natural* numbers (*nat*). Linked lists used in the implementation for the elements *levels* and *free_list*, together with the bitmaps used in *bits* and *bits_p*, are defined as a *list* type. C *structs* are modelled in Isabelle as *records* of the same name as the implementation and comprising the same data. There

```
 1  static int pool_alloc(struct k_mem_pool *p,struct k_mem_block *block,size_t size)
 2  {
 3    ..... //calcuate lsizes[], alloc_l and free_l
 4    if (alloc_l < 0 || free_l < 0) {
 5      block->data = NULL;
 6      return -ENOMEM;
 7    }
 8    blk = alloc_block(p, free_l, lsizes[free_l]);
 9    if (!blk) { return -EAGAIN; }
10    /* Iteratively break the smallest enclosing block... */
11    for (from_l = free_l; level_empty(p, alloc_l) && from_l < alloc_l;
12             from_l++) {
13      blk = break_block(p, blk, from_l, lsizes);
14    }
15    block->id.level = alloc_l; //assign block level to the variable *block
16    ......  //assign other block info to the variable *block
17    return 0;
18  }
19
20  int k_mem_pool_alloc(struct k_mem_pool *p, struct k_mem_block *block, size_t size,
         s32_t timeout)
21  {
22    ...... // initialize local vars, calculate the end time for timeout.
23    while (1) {
24      ret = pool_alloc(p, block, size);
25      if (ret == 0 || timeout == K_NO_WAIT ||
26          ret == -EAGAIN || (ret && ret != -ENOMEM)) {
27        return ret;
28      }
29      key = irq_lock();
30      _pend_current_thread(&p->wait_q, timeout);
31      _Swap(key);
32      ...... //if timeout > 0, break the loop if time out
33    }
34    return -EAGAIN;
35  }
```

Fig. 1: The C Source Code of Memory Allocation in Zephyr v1.8.0

are two exceptions to this: (1) $k\_mem\_block\_id$ and k_mem_block are merged in one single record, (2) the union in the struct $k\_mem\_pool\_lvl$ is replaced by a single list representing the bitmap, and thus *max_inline_level* is removed.

The Zephyr implementation makes use of a bitmap to represent the state of a memory block. The bit j of the bitmap for level a i is set to 1 iff the memory address of the memory block (i, j) is in the free list at level i. A bit j at a level i is set to 0 under the following conditions: (1) its corresponding memory block is allocated (*ALLOCATED*), (2) the memory block has been split (*DIVIDED*), (3) the memory block is being split in the allocation service (*ALLOCATING*) (Line 13 in Fig. 1), (4) the memory block is being coalesced in the release service (*FREEING*), and (5) the memory block does not exist (*NOEXIST*). Instead of only using a binary representation, our formal specification models the bitmap using a datatype *BlockState* that is composed of these cases together with *FREE*. The reason of this decision is to simplify proving that the bitmap shape is well-formed. In particular, this representation makes less complex to verify the case in which the descendant of a free block is a non-free block. This is the case where the last free block has not been split and therefore lower levels do not exist. We illustrate a

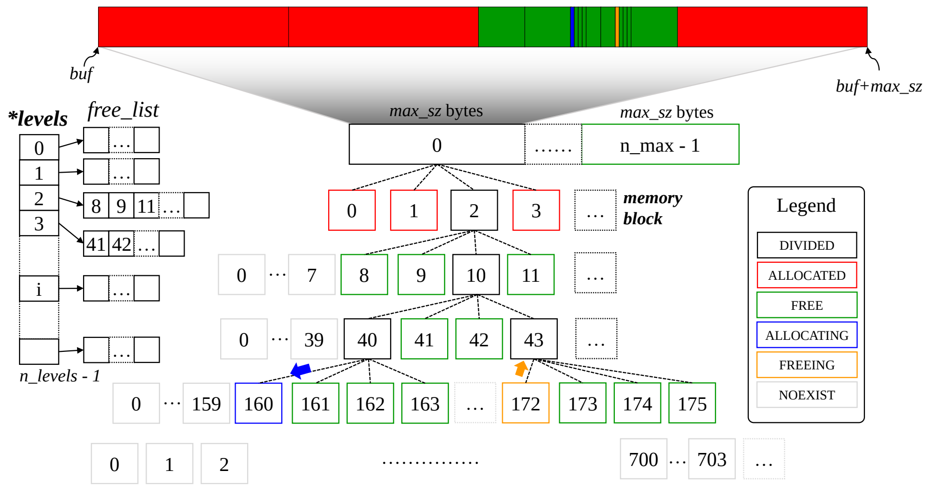

Fig. 2: Structure of Memory Pools

structure of a memory pool in Fig. 2. The top of the figure shows the real memory of the first block at level 0.

The structural properties clarify the constraints on and consistency of quad trees, free block lists, the memory pool configuration, and waiting threads. All of them are thought of as invariants on the kernel state and have been formally verified on the formal specification in Isabelle/HOL.

Well-shaped bitmaps. We say that the logical memory block j at a level i physically exists iff the bitmap j for the level i is *ALLOCATED*, *FREE*, *ALLOCATING*, or *FREEING*, represented by the predicate $is\_memblock$. We do not consider blocks marked as *DIVIDED* as physical blocks since it is only a logical block containing other blocks. Threads may split and coalesce memory blocks. A valid forest is defined by the following rules: (1) the parent bit of an existing memory block is *DIVIDED* and its child bits are *NOEXIST*, denoted by the predicate $noexist\_bits$ that checks for a given bitmap b and a position j that nodes $b!j$ to $b!(j+3)$ are set as *NOEXIST*; (2) the parent bit of a *DIVIDED* block is also *DIVIDED*; and (3) the child bits of a *NOEXIST* bit are also *NOEXIST* and its parent can not be a *DIVIDED* block. The property is defined as the predicate **inv-bitmap**(s), where s is the state.

There are two additional properties on bitmaps. First, the address space of any memory pool cannot be empty, i.e., the bits at level 0 have to be different to *NOEXIST*. Second, the allocation algorithm may split a memory block into smaller ones, but not the those blocks at the lowest level (i.e. level $n\_levels - 1$), therefore the bits at the lowest level cannot not be *DIVIDED*. The first property is defined as **inv-bitmap0**(s) and the second as **inv-bitmapn**(s).

Consistency of the memory configuration. The configuration of a memory pool is set when it is initialized. Since the minimum block size is aligned to 4 bytes, there must exists an $n > 0$ such that the maximum size of a pool is equal to $4 \times n \times 4^{n\_levels}$, relating the number of levels of a level 0 block with its maximum size. Moreover, the number of blocks at level 0 and the number of levels have to be greater than zero, since the memory pool cannot be empty. The number of levels is equal to the length of the pool $levels$ list. Finally, the length of the bitmap at level i should be $n\_max \times 4^i$. This property is defined as **inv-mempool-info**(s).

Memory partition property. Memory blocks partition the pool they belong to, and then not overlapping blocks and the absence of memory leaks are critical properties. For a memory block of index j at level i, its address space is the interval $[j \times (max\_sz/4^i), (j+1) \times (max\_sz/4^i))$. For any relative memory address $addr$ in the memory domain of a memory pool, and hence $addr < n\_max * max\_sz$, there is one and only one memory block whose address space contains $addr$. Here, we use relative address for $addr$. The property is defined as **mem-part**(s).

From the invariants of the bitmap, we derive the general property for the memory partition.

Theorem 1 (Memory Partition). *For any kernel state s, If the memory pools in s are consistent in their configuration, and their bitmaps are well-shaped, the memory pools satisfy the partition property in s:*

$$\textbf{\textit{inv\_mempool\_info}}(s) \wedge \textbf{\textit{inv\_bitmap}}(s) \wedge \textbf{\textit{inv\_bitmap0}}(s) \wedge \textbf{\textit{inv\_bitmapn}}(s) \Longrightarrow \textbf{\textit{mem\_part}}(s)$$

Together with the memory partition property, pools must also satisfy the following:

No partner fragmentation. The memory release algorithm in Zephyr coalesces free partner memory blocks into blocks as large as possible for all the descendants from the root level, without including it. Thus, a memory pool does not contain four *FREE* partner bits.

Validity of free block lists. The free list at one level keeps the starting address of free memory blocks. The memory management ensures that the addresses in the list are valid, i.e., they are different from each other and aligned to the *block size*, which at a level i is given by $(max\_sz/4^i)$. Moreover, a memory block is in the free list iff the corresponding bit of the bitmap is *FREE*.

Non-overlapping of memory pools. The memory spaces of the set of pools defined in a system must be disjoint, so the memory addresses of a pool does not belong to the memory space of any other pool.

Other properties. The state of a suspended thread in *wait_q* has to be consistent with the threads waiting for a memory pool. Threads can only be blocked once, and those threads waiting for available memory blocks have to be in a *BLOCKED* state. During allocation and free of a memory block, blocks of the tree may temporally be manipulated during the coalesce and division process. A block can be only manipulated by a thread at a time, and the state bit of a block being temporally manipulate has to be *FREEING* or *ALLOCATING*.

4 Formalizing Zephyr Memory Management

For the purpose of formal verification of event-driven systems such as OSs, we have developed π-Core, a framework for rely-guarantee reasoning of components running in parallel invoking events. π-Core has support for concurrent OSs features like modelling shared-variable concurrency of multiple threads, interruptable execution of handlers, self-suspending threads, and rescheduling. In this section, we first introduce the modelling language in π-Core and an execution model of Zephyr using this language. Then we discuss in detail the low-level design specification for the kernel services that the memory management provides. Since this work focuses on the memory management, we only provide very abstract models for other kernel functionalities such as the kernel scheduling and thread control.

4.1 Event-based Execution Model of Zephyr

The language in π-Core. Interrupt handlers in π-Core are considered as reaction services which are represented as *events*:

$$\textbf{EVENT}\ \mathcal{E}\ [p_1, ..., p_n]@\kappa\ \textbf{WHEN}\ g\ \textbf{THEN}\ P\ \textbf{END}$$

In this representation, an event is a parametrized imperative program P with a name $\mathcal{E}$, a list of service input parameters $p_1, ..., p_n$, and a guard condition g to determine the conditions triggering the event. In addition to the input parameters, an event has a special parameter κ which indicates the execution context, e.g. the scheduler and the thread invoking the event. The imperative commands of an event body P in π-Core are standard sequential constructs such as conditional execution, loop, and sequential composition of programs. It also includes a synchronization construct for concurrent processes represented by **AWAIT** b **THEN** P **END**. The body P is executed atomically if and only if the boolean condition b holds, not progressing otherwise. **ATOM** P **END** denotes an *Await* statement for which its guard is $True$.

Threads and kernel processes have their own execution context and local states. Each of them is modelled in π-Core as a set of events called *event systems* and denoted as **ESYS** $\mathcal{S} \equiv \{\mathcal{E}_0,\ ...,\ \mathcal{E}_n\}$. The operational semantics of an event system is the *sequential composition* of the execution of the events composing it. It consists in the continuous evaluation of the guards of the system events. From the set of events for which the associated guard g holds in the current state, one event $\mathcal{E}$ is non-deterministically selected to be triggered, and its body P executed. After P finishes, the evaluation of the guards starts again looking for the next event to be executed. Finally, π-Core has a construct for parallel composition of event systems $esys_0 \parallel ... \parallel esys_n$ which interleaves the execution of the events composing each event system $esys_i$ for $0 \leq i \leq n$.

Execution model of Zephyr. If we do not consider its initialization, an OS kernel can be consider as a reactive system that is in an *idle* loop until it receives an interruption which is handled by an interruption handler. Whilst interrupt handlers execution is atomic in sequential kernels, it can be interrupted in concurrent kernels [6,22] allowing services invoked by threads to be interrupted and resumed later. In the execution model of Zephyr, we consider a scheduler $\mathcal{S}$ and a set of threads $t_1, ..., t_n$. In this model, the

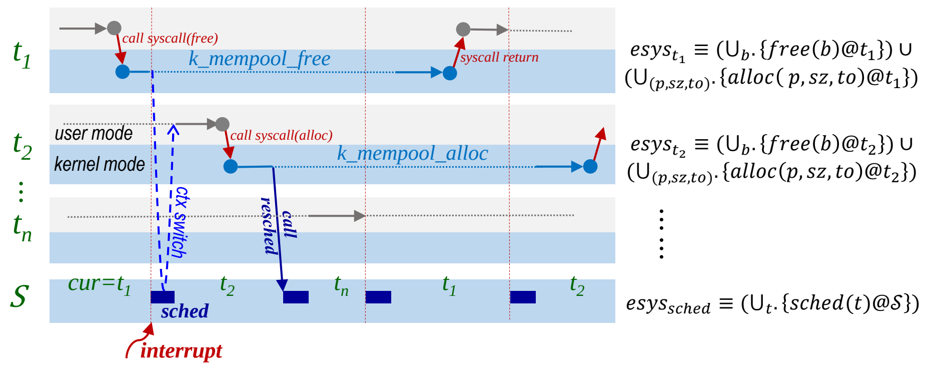
t_1
call syscall(free)
k_mempool_free
syscall return
$esys_{t_1} \equiv (\bigcup_b . \{free(b)@t_1\}) \cup (\bigcup_{(p,sz,to)} . \{alloc(p, sz, to)@t_1\})$
t_2
user mode
kernel mode
call syscall(alloc)
k_mempool_alloc
$esys_{t_2} \equiv (\bigcup_b . \{free(b)@t_2\}) \cup (\bigcup_{(p,sz,to)} . \{alloc(p, sz, to)@t_2\})$
ctx switch
call resched
t_n
S
cur=t_1
t_2
t_n
t_1
t_2
sched
interrupt
$esys_{sched} \equiv (\bigcup_t . \{sched(t)@\mathcal{S}\})$

Fig. 3: An Execution Model of Zephyr Memory Management

execution of the scheduler is atomic since kernel services can not interrupt it. But kernel services can be interrupted via the scheduler, i.e., the execution of a memory service invoked by a thread t_i may be interrupted by the kernel scheduler to execute a thread t_j. Fig.3 illustrates Zephyr execution model, where solid lines represent execution steps of the threads/kernel services and dotted lines mean the suspension of the thread/code. For instance, the execution of *k_mempool_free* in thread t_1 is interrupted by the scheduler, and the context is switched to thread t_2 which invokes *k_mempool_alloc*. During the execution of t_2, the kernel service may suspend the thread and switch to another thread t_n by calling *rescheduling*. Later, the execution is switched back to t_1 and continues the execution of *k_mempool_free* in a different state from when it was interrupted.

The event systems of Zephyr are illustrated in the right part of Fig. 3. A user thread t_i invoke allocation/release services, thus the event system for t_i is $esys_{t_i}$, a set composed of the events *alloc* and *free*. The input parameters for these events correspond with the arguments of the service implementation, that are constrained by the guard for each service. Together with system users we model the event service for the scheduler $esys_{sched}$ consisting on a unique event *sched* whose argument is a thread t to be scheduled when t is in the *READY* state. The formal specification of the memory management is the parallel composition of the event system for the threads and the scheduler $esys_{t_1} \parallel ... \parallel esys_{t_n} \parallel esys_{sched}$

Thread context and preemption. Events are parametrized by a thread identifier used to access to the execution context of the thread invoking it. As shown in Figure 3, the execution of an event executed by a thread can be stopped by the scheduler to be resumed later. This behaviour is modelled using a global variable cur that indicates the thread being currently has been scheduled and is being executed, and conditioning the execution of parametrized events in t only when t is scheduled. This is achieved by using the expression t ► p $\equiv$ **AWAIT** $cur = t$ **THEN** p **END**, so an event invoked by a thread t only progresses when t is scheduled. This scheme allows to use rely-guarantee for concurrent execution of threads on mono-core architectures, where only the scheduled thread is able to modify the memory.

4.2 Formal Specification of Memory Management Services

This section discusses the formal specification of the memory management services. These services deal with the initialization of pools, and memory allocation and release.

System state. The system state includes the memory model introduced in Section 4, together with the thread under execution in variable cur and local variables to the memory services used to keep temporal changes to the structure, guards in conditional and loop statements, and index accesses. The memory model is represented as a set *mem_pools* storing the references of all memory pools and a mapping *mem_pool_info* to query a pool by a pool reference. Local variables are modelled as total functions from threads to variable values, representing that the event is accessing the thread context. In the formal model of the events we represent access to a state component c using $\acute{}c$ and the value of a local component c for the thread t is represented as $\acute{}c\ t$. Local variables *allocating_node* and *freeing_node* are relevant for the memory services, storing the temporal blocks being split/coalesced in alloc/release services respectively.

Memory pool initialization. Zephyr defines and initializes memory pools at compile time by constructing a static variable of type ***struct*** *k_mem_pool*. The implementation initializes each pool with *n_max* level 0 blocks with size *max_sz* bytes. Bitmaps of level 0 are set to 1 and free list contains all level 0 blocks. Bitmaps and free lists of other level are initialized to 0 and to the empty list respectively. In the formal model, we specify a state corresponding to the implementation initial state and we show that it belongs to the set of states satisfying the invariant.

Memory allocation/release services. The C code of Zephyr uses the recursive function *free_block* to coalesce free partner blocks and the *break* statement to stop the execution of a loop statements, which are not supported by the imperative language in π-Core. The formal specification overcomes this by transforming the recursion into a loop controlled by the recursion conditoin, and using a control variable to exit loops with breaks when the condition to execute the loop break is satisfied. Additionally, the memory management services use the atomic body *irq_lock(); P; irq_unlock();* to keep interruption handlers *reentrant* by disabling interruptions. We simplify this behaviour in the specification using an **ATOM** statement, avoiding that the service is interrupted at that point. The rest of the formal specification closely follows the implementation, where variables are modified using higher order functions changing the state as the code does it. The reason of using Isabelle/HOL functions is that π-Core does not provide a semantic for expressions, using instead state transformer relying on high order functions to change the state.

Fig. 4 illustrates the π-Core specification of the *free_block* function invoked by *k_mem_pool_free* when releasing a memory block. The code accesses the following variables: lsz, $lsize$, and lvl to keep information about the current level; blk, bn, and bb to represent the address and number of the block currently being accessed; $freeing\_node$ to represent the node being freeing; and i to iterate blocks. Additionally, the model includes the component $free\_block\_r$ to model the recursion condition. To simplify the representation the model uses predicates and functions to access and modify the state.

```
 1 WHILE ´free-block-r t DO
 2   t ▶ ´lsz := ´lsz (t := ´lsizes t ! (´lvl t));;
 3   t ▶ ´blk := ´blk (t := block-ptr (´mem-pool-info (pool b)) (´lsz t) (´bn t));;
 4   t ▶ ATOM
 5     ´mem-pool-info := set-bit-free ´mem-pool-info (pool b) (´lvl t) (´bn t);;
 6     ´freeing-node := ´freeing-node (t := None);;
 7     IF ´lvl t > 0 ∧ partner-bits (´mem-pool-info (pool b)) (´lvl t) (´bn t) THEN
 8       FOR ´i := ´i(t := 0); ´i t < 4; ´i := ´i(t := ´i t + 1) DO
 9         ´bb := ´bb (t := (´bn t div 4) ∗ 4 + ´i t);;
10         ´mem-pool-info := set-bit-noexist ´mem-pool-info (pool b) (´lvl t) (´bb t);;
11         ´block-pt := ´block-pt (t := block-ptr (´mem-pool-info (pool b)) (´lsz t) (´bb t));;
12         IF ´bn t ≠ ´bb t ∧ block-fits (´mem-pool-info (pool b)) (´block-pt t) (´lsz t) THEN
13           ´mem-pool-info := ´mem-pool-info ((pool b) :=
14                 remove-free-list (´mem-pool-info (pool b)) (´lvl t) (´block-pt t))
15         FI
16       ROF;;
17       ´lvl := ´lvl (t := ´lvl t − 1);;
18       ´bn := ´bn (t := ´bn t div 4);;
19       ´mem-pool-info := set-bit-freeing ´mem-pool-info (pool b) (´lvl t) (´bn t);;
20       ´freeing-node := ´freeing-node (t := Some ⦇pool = (pool b), level = (´lvl t),
21             block = (´bn t), data = block-ptr (´mem-pool-info (pool b))
22              (((ALIGN4 (max-sz (´mem-pool-info (pool b)))) div (4 ^ (´lvl t)))) (´bn t) ⦈)
23     ELSE
24       IF block-fits (´mem-pool-info (pool b)) (´blk t) (´lsz t) THEN
25         ´mem-pool-info := ´mem-pool-info ((pool b) :=
26           append-free-list (´mem-pool-info (pool b)) (´lvl t) (´blk t) )
27       FI;;
28       ´free-block-r := ´free-block-r (t := False)
29     FI
30   END
31 OD
```

Fig. 4: The π-Core Specification of *free_block*

Due to space constrains, we are unable to provide detailed explanation of these functions. However the name of the functions can help the reader to better understand their functionality. We refer readers to the Isabelle/HOL sources for the complete specification of the formal model.

In the C code, *free_block* is a recursive function with two conditions: (1) the block being released belongs to a level higher than zero, since blocks at level zero cannot be merged; and (2) the partners bits of the block being released are FREE so they can be merged into a bigger block. We represent (1) with the predicate $\acute{}lvl\ t > 0$ and (2) with the predicate $partner\_bit\_free$. The formal specification follows the same structure translating the recursive function into a loop that is controlled by a variable mimicking the recursion.

The formal specification for *free_block* first releases an allocated memory block bn setting it to *FREEING*. Then, the loop statement sets *free_block* to *FREE* (Line 5), and also checks that the iteration/recursive condition holds in Line 7. If the condition holds, the partner bits are set to *NOEXIST*, and remove their addresses from the free list for this level (Lines 12 - 14). Then, it sets the parent block bit to *FREEING* (Lines 17 - 22), and updates the variables controlling the current block and level numbers, before going back to the beginning of the loop again. If the iteration condition is not true it sets the bit to *FREE* and add the block to the free list (Lines 24 - 28) and sets the loop condition to false to end the procedure. This function is illustrated in Fig. 2. The block 172 is released by a thread and since its partner blocks (block $173 - 175$) are free, Zephyr coalesces the four blocks and sets their parent block 43 as *FREEING*. The coalescence continues iteratively if the partners of block 43 are all free.

5 Correctness and Rely-guarantee Proof

We have proven correctness of the buddy memory management in Zephyr using the rely-guarantee proof system of π-Core. We ensure functional correctness of each kernel service w.r.t. the defined pre/post conditions, invariant preservation, termination of loop statements in the kernel services, the preservation of the memory configuration during small steps of kernel services, and the separation of local variables of threads. In this section, we introduce the rely-guarantee proof system of π-Core and how these properties are specified and verified using it.

5.1 Rely-guarantee Proof Rules and Verification

A rely-guarantee specification for a system is a quadruple $RGCond = \langle pre, R, G, pst\rangle$, where pre is the pre-condition, R is the rely condition, G is the guarantee condition, and pst is the post-condition. The intuitive meaning of a valid rely-guarantee specification for a parallel component P, denoted by $\models P \textbf{ sat } \langle pre, R, G, pst\rangle$, is that if P is executed from an initial state $s \in pre$ and any environment transition belongs to the rely relation R, then the state transitions carried out by P belong to the guarantee relation G and the final states belong to pst.

We have defined a rely-guarantee axiomatic proof system for the π-Core specification language to prove validity of rely-guarantee specifications, and proven in Isabelle/HOL its soundness with regards to the the definition of validity. Some of the rules composing the axiomatic reasoning system are shown in Fig. 5.

A predicate P is stable w.r.t. a relation R, represented as $stable(P, R)$, when for any pair of states (s, t) such that $s \in P$ and $(s, t) \in R$ then $t \in P$. The intuitive meaning is that an environment represented by R does not affect the satisfiability of P. The parallel rule in Fig. 5 establishes compositionality of the proof system, where verification of the parallel specification can be reduced to the verification of individual event systems first and then to the verification of individual events. It is necessary that each event system $\mathcal{PS}(\kappa)$ satisfies its specification $\langle pres_\kappa, Rs_\kappa, Gs_\kappa, psts_\kappa\rangle$ (Premise 1); the pre-condition for the parallel composition implies all the event system's pre-conditions (Premise 2); the overall post-condition must be a logical consequence of all post-conditions of event systems (Premise 3); since an action transition of the concurrent system is performed by one of its event system, the guarantee condition Gs_κ of each event system must be a subset of the overall guarantee condition G (Premise 4); an

$$\frac{\begin{array}{l}[\textsc{Await}]\\ \vdash P\ \mathbf{sat}\ \langle pre \cap b \cap \{V\}, Id, UNIV, \{s \mid (V,s) \in G\} \cap pst\rangle \\ stable(pre,R) \quad stable(pst,R)\end{array}}{\vdash (\mathbf{Await}\ b\ P)\ \mathbf{sat}\ \langle pre, R, G, pst\rangle} \qquad \frac{\begin{array}{l}[\textsc{BasicEvt}]\\ \vdash body(\alpha)\ \mathbf{sat}\ \langle pre \cap guard(\alpha), R, G, pst\rangle \\ stable(pre,R) \quad \forall s.\ (s,s) \in G\end{array}}{\vdash \mathbf{Event}\ \alpha\ \mathbf{sat}\ \langle pre, R, G, pst\rangle}$$

$$\frac{\begin{array}{l}[\textsc{While}]\\ \vdash P\ \mathbf{sat}\ \langle loopinv \cap b, R, G, loopinv\rangle \\ loopinv \cap -b \subseteq pst \quad \forall s.\ (s,s) \in G \\ stable(loopinv, R) \quad stable(pst, R)\end{array}}{\vdash (\mathbf{While}\ b\ P)\ \mathbf{sat}\ \langle loopinv, R, G, pst\rangle} \qquad \frac{\begin{array}{l}[\textsc{Par}]\\ (1)\forall\kappa.\ \vdash \mathcal{PS}(\kappa)\ \mathbf{sat}\ \langle pres_\kappa, Rs_\kappa, Gs_\kappa, psts_\kappa\rangle \\ (2)\forall\kappa.\ pre \subseteq pres_\kappa \quad (3)\forall\kappa.\ psts_\kappa \subseteq pst \quad (4)\forall\kappa.\ Gs_\kappa \subseteq G \\ (5)\forall\kappa.\ R \subseteq Rs_\kappa \quad (6)\forall\kappa,\kappa'.\ \kappa \neq \kappa' \longrightarrow Gs_\kappa \subseteq Rs_{\kappa'}\end{array}}{\vdash \mathcal{PS}\ \mathbf{sat}\ \langle pre, R, G, pst\rangle}$$

Fig. 5: Typical Rely-guarantee Proof Rules in π-Core

environment transition Rs_κ for the event system κ corresponds to a transition from the overall environment R (Premise 5); and an action transition of an event system κ should be defined in the rely condition of another event system κ', where $\kappa \neq \kappa'$ (Premise 6).

To prove loop termination, loop invariants are parametrized with a logical variable α. It suffices to show total correctness of a loop statement by the following proposition where $loopinv(\alpha)$ is the parametrize invariant, in which the logical variable is used to find a convergent relation to show that the number of iterations of the loop is finite.

$$\vdash P\ \mathbf{sat}\ \langle loopinv(\alpha) \cap \{\!|\ \alpha > 0\ |\!\}, R, G, \exists\beta < \alpha.\ loopinv(\beta)\rangle \wedge loopinv(\alpha) \cap \{\!|\ \alpha > 0\ |\!\} \subseteq \{\!|\ b\ |\!\}$$
$$\wedge\ loopinv(0) \subseteq \{\!|\ \neg b\ |\!\} \wedge \forall s \in loopinv(\alpha).\ (s,t) \in R \longrightarrow \exists\beta \leqslant \alpha.\ t \in loopinv(\beta)$$

5.2 Correctness Specification

Using the compositional reasoning of π-Core, correctness of Zephyr memory management can be specified and verified with the rely-guarantee specification of each event. The functional correctness of a kernel service is specified by its pre/post-conditions. Invariant preservation, memory configuration, and separation of local variables is specified in the guarantee condition of each service.

The guarantee condition for both memory services is defined as:

$$\textbf{Mem-pool-alloc-guar}\ t \equiv \overbrace{Id}^{(1)} \cup (\overbrace{gvars\_conf\_stable}^{(2)} \cap$$
$$\{(s,r).\ (\overbrace{cur\ s \neq Some\ t \longrightarrow gvars\text{-}nochange\ s\ r \wedge lvars\text{-}nochange\ t\ s\ r}^{(3.1)})$$
$$\wedge\ (\overbrace{cur\ s = Some\ t \longrightarrow inv\ s \longrightarrow inv\ r}^{(3.2)}) \wedge (\overbrace{\forall t'.\ t' \neq t \longrightarrow lvars\text{-}nochange\ t'\ s\ r}^{(4)})\})$$

This relation states that *alloc* and *free* services may not change the state (1), e.g., a blocked await or selecting branch on a conditional statement. If it changes the state then: (2) the static configuration of memory pools in the model do not change; (3.1) if the scheduled thread is not the thread invoking the event then variables for that thread do not change (since it is blocked in an *Await* as explained in Section 3); (3.2) if it is, then the relation preserves the memory invariant, and consequently each step of the event needs to preserve the invariant; (4) a thread does not change the local variables of other threads.

Using the π-Core proof rules we verify that the invariant introduced in Section 4 is preserved by all the events. Additionally, we prove that when starting in a valid memory configuration given by the invariant, then if the service does not returns an error code then it returns a valid memory block with size bigger or equal than the requested capacity. The property is specified by the following postcondition:

Mem-pool-alloc-pre $t \equiv \{s.\ \mathit{inv}\ s \wedge \mathit{allocating\text{-}node}\ s\ t = \mathit{None} \wedge \mathit{freeing\text{-}node}\ s\ t = \mathit{None}\}$

Mem-pool-alloc-post $t\ p\ sz\ \mathit{timeout} \equiv$

$$\begin{aligned}
&\{s.\ \mathit{inv}\ s \wedge \mathit{allocating\text{-}node}\ s\ t = \mathit{None} \wedge \mathit{freeing\text{-}node}\ s\ t = \mathit{None}\\
&\quad \wedge (\mathit{timeout} = \mathit{FOREVER} \longrightarrow\\
&\qquad (\mathit{ret}\ s\ t = \mathit{ESIZEERR} \wedge \mathit{mempoolalloc\text{-}ret}\ s\ t = \mathit{None} \vee\\
&\qquad \mathit{ret}\ s\ t = \mathit{OK} \wedge (\exists\, \mathit{mblk}.\ \mathit{mempoolalloc\text{-}ret}\ s\ t = \mathit{Some\ mblk} \wedge \mathit{mblk\text{-}valid}\ s\ p\ sz\ \mathit{mblk})))\\
&\quad \wedge (\mathit{timeout} = \mathit{NOWAIT} \longrightarrow\\
&\qquad ((\mathit{ret}\ s\ t = \mathit{ENOMEM} \vee \mathit{ret}\ s\ t = \mathit{ESIZEERR}) \wedge \mathit{mempoolalloc\text{-}ret}\ s\ t = \mathit{None}) \vee\\
&\qquad (\mathit{ret}\ s\ t = \mathit{OK} \wedge (\exists\, \mathit{mblk}.\ \mathit{mempoolalloc\text{-}ret}\ s\ t = \mathit{Some\ mblk} \wedge \mathit{mblk\text{-}valid}\ s\ p\ sz\ \mathit{mblk})))\\
&\quad \wedge (\mathit{timeout} > 0 \longrightarrow\\
&\qquad ((\mathit{ret}\ s\ t = \mathit{ETIMEOUT} \vee \mathit{ret}\ s\ t = \mathit{ESIZEERR}) \wedge \mathit{mempoolalloc\text{-}ret}\ s\ t = \mathit{None}) \vee\\
&\qquad (\mathit{ret}\ s\ t = \mathit{OK} \wedge (\exists\, \mathit{mblk}.\ \mathit{mempoolalloc\text{-}ret}\ s\ t = \mathit{Some\ mblk}\\
&\qquad\qquad\qquad\qquad \wedge\ \mathit{mblk\text{-}valid}\ s\ p\ sz\ \mathit{mblk})))\}
\end{aligned}$$

If a thread requests a memory block in mode *FOREVER*, it may successfully allocate a valid memory block, or fail (*ESIZEERR*) if the request size is larger than the size of the memory pool. If the thread is requesting a memory pool in mode *NOWAIT*, it may also get the result of *ENOMEM* if there is no available blocks. But if the thread is requesting in mode *TIMEOUT*, it will get the result of *ETIMEOUT* if there is no available blocks in *timeout* milliseconds.

The property is indeed weak since even if the memory has a block able to allocate the requested size before invoking the allocation service, another thread running concurrently may have taken the block first during the execution of the service. For the same reason, the released block may be taken by another concurrent thread before the end of the release services.

5.3 Correctness Proof

In the π-Core system, verification of a rely-guarantee specification proving a property is carried out by inductively applying the proof rules for each system event and discharging the proof obligations the rules generate. Typically, these proof obligations require to prove stability of the pre- and post-condition to check that changes of the environment preserve them, and to show that a statement modifying a state from the precondition gets a state belonging to the postcondition.

To prove termination of the loop statement in *free_block* shown in Fig. 4, we define the loop invariant with the logical variable α as follows.

mp-free-loopinv $t\ b\ \alpha \equiv$

$$\begin{aligned}
&\{\!|\ \ldots \wedge \acute{}\mathit{inv} \wedge \mathit{level}\ b < \mathit{length}\ (\acute{}\mathit{lsizes}\ t)\\
&\quad \wedge (\forall\, \mathit{ii} {<} \mathit{length}\ (\acute{}\mathit{lsizes}\ t).\ \acute{}\mathit{lsizes}\ t\ !\ \mathit{ii} = (\mathit{max\text{-}sz}\ (\acute{}\mathit{mem\text{-}pool\text{-}info}\ (\mathit{pool}\ b)))\ \mathit{div}\ (4\ \hat{}\ \mathit{ii}))\\
&\quad \wedge \acute{}\mathit{bn}\ t < \mathit{length}\ (\mathit{bits}\ (\mathit{levels}\ (\acute{}\mathit{mem\text{-}pool\text{-}info}\ (\mathit{pool}\ b))!(\acute{}\mathit{lvl}\ t)))\\
&\quad \wedge \acute{}\mathit{bn}\ t = (\mathit{block}\ b)\ \mathit{div}\ (4\ \hat{}\ (\mathit{level}\ b - \acute{}\mathit{lvl}\ t)) \wedge \acute{}\mathit{lvl}\ t \le \mathit{level}\ b\\
&\quad \wedge (\acute{}\mathit{free\text{-}block\text{-}r}\ t \longrightarrow (\exists\, \mathit{blk}.\ \acute{}\mathit{freeing\text{-}node}\ t = \mathit{Some\ blk} \wedge \mathit{pool}\ \mathit{blk} = \mathit{pool}\ b\\
&\qquad\qquad\qquad\qquad \wedge\ \mathit{level}\ \mathit{blk} = \acute{}\mathit{lvl}\ t \wedge \mathit{block}\ \mathit{blk} = \acute{}\mathit{bn}\ t)\\
&\qquad\qquad\qquad \wedge\ \acute{}\mathit{alloc\text{-}memblk\text{-}data\text{-}valid}\ (\mathit{pool}\ b)\ (\mathit{the}\ (\acute{}\mathit{freeing\text{-}node}\ t)))
\end{aligned}$$

Table 1: Specification and Proof Statistics

π-**Core Language**		**Memory Management**	
Item	**LOS/LOP**	**Item**	**LOS/LOP**
Language and Proof Rules	700	*Specification*	400
Lemmas of Language/Semantics	3000	*Auxiliary Lemmas/Invariant*	1700
Soundness	7100	*Proof of Allocation*	10600
Invariant	100	*Proof of Free*	4950
Total	10,900	**Total**	17,650

$\wedge$ ($\neg$ ´*free-block-r t* $\longrightarrow$ ´*freeing-node t* $=$ *None*) $\}\!\!\} \cap$
$\{\!\!\{$ $\alpha =$ (*if* ´*freeing-node t* $\neq$ *None then* ´*lvl t* $+$ *1 else 0*) $\}\!\!\}$

$freeing\_node$ and lvt are local variables respectively storing the node being free and the level that the node belongs to. In the body of the loop, if $lvl\ t > 0$ and $partner\_bit$ is *true*, then $lvl = lvl - 1$ at the end of the body. Otherwise, $freeing\_node\ t = None$. So at the end of the loop body, α decreases or $\alpha = 0$. If $\alpha = 0$, we have $freeing\_node\ t = None$, and thus the negation of the loop condition $\neg free\_block\_r\ t$, concluding termination of *free_block*.

Due to concurrency, it is necessary to consider fairness to prove termination of the loop statement in *k_mempool_alloc* from Line 23 to 33 in Fig. 1. On the one hand, when a thread requests a memory block in the *FOREVER* mode, it is possible that there will never be available blocks since other threads do not release allocated blocks. On the other hand, even when other threads release blocks, it is possible that the available blocks are always raced by threads.

6 Evaluation and Results

Evaluation The verification conducted in this work is on Zhephyr v1.8.0, released in 2017. The C code of the buddy memory management is $\approx$ 400 lines, not counting blank lines and comments. Table 1 shows the statistics for the effort and size of the proofs in the Isabelle/HOL theorem prover. In total, the models and mechanized verification consists of $\approx$ 28,000 lines of specification and proofs, and the total effort is $\approx$ 12 person-months. The specification and proof of π-Core are reusable for the verification of other systems.

Bugs in Zephyr During the formal verification, we found 3 bugs in the C code of Zephyr. The first two bugs are critical and have been repaired in the latest release of Zephyr. To avoid the third one, callers to *k_mem_pool_alloc* have to constrain the argument *t_size size*.

(1) Incorrect block split: this bug is located in the loop in Line 11 of the *k_mem_pool_alloc* service, shown in Fig. 1. The *level_empty* function checks if a pool p has blocks in the free list at level *alloc_l*. Concurrent threads may release a memory block at that level making the call to *level_empty(p, alloc_l)* to return *false* and stopping the loop. In such case, it allocates a memory block of a bigger capacity at a level i but it still sets the level number of the block as *alloc_l* at Line 15. The service allocates a larger block to the requesting thread causing an internal fragmentation of $max\_sz/4^i - max\_sz/4^{alloc\_l}$ bytes. When this block is released, it will be inserted into the free list at level *alloc_l*, but not at level i, causing an external fragmentation of $max\_sz/4^i - max\_sz/4^{alloc\_l}$. The bug is fixed by removing the condition *level_empty(p, alloc_l)* in our specification.

(2) Incorrect return from *k_mem_pool_alloc*: this bug is found at Line 26 in Fig. 1. When a suitable free block is allocated by another thread, the *pool_alloc* function returns *EAGAIN* at Line 9 to ask the thread to retry the allocation. When a thread invokes *k_mem_pool_alloc* in *FOREVER* mode and this case happens, the service returns *EAGAIN* immediately. However, a thread invoking *k_mem_pool_alloc* in *FOREVER* mode should keep retrying when it does not succeed. We repair the bug by removing the condition $ret == EAGAIN$ at Line 26. As explained in the comments of the C Code, *EAGAIN* should not be returned to threads invoking the service. Moreover, the *return EAGAIN* at Line 34 is actually the case of time out. Thus, we introduce a new return code *ETIMEOUT* in our specification.

(3) Non-termination of *k_mem_pool_alloc*: we have discussed that the loop statement at Lines 23 - 33 in Fig. 1 does not terminate. However, it should terminate in certain cases, which are actually violated in the C code. When a thread requests a memory block in *FOREVER* mode and the requested size is larger than *max_sz*, the maximum size of blocks, the loop at Lines 23 - 33 in Fig. 1 never finishes since *pool_alloc* always returns *ENOMEM*. The reason is that the "*return ENOMEM*" at Line 6 does not distinguish two cases, $alloc\_l < 0$ and $free\_l < 0$. In the first case, the requested size is larger than *max_sz* and the kernel service should return immediately. In the second case, there are no free blocks larger than the requested size and the service tries forever until some free block available. We repair the bug by splitting the *if* statement at Lines 4 - 7 into these two cases and introducing a new return code *ESIZEERR* in our specification. Then, we change the condition at Lines 25 - 26 to check that the returned value is *ESIZEERR* instead of *ENOMEM*.

7 Conclusion and Future Work

In this paper, we have developed a formal specification at low-level design of the concurrent buddy memory management of Zephyr RTOS. Using the rely-guarantee technique in the π-Core framework, we have formally verified a set of critical properties for OS kernels such as invariant preservation, and preservation of memory configuration. Finally, we identified some critical bugs in the C code of Zephyr.

Our work explores the challenges and cost of certifying concurrent OSs for the highest-level assurance. The definition of properties and rely-guarantee relations is complex and the verification task becomes expensive. We used 40 times of LOS/LOP than the C code at low-level design. Next, we are planning to verify other modules of Zephyr, which may be easier due to simpler data structures and algorithms. For the purpose of fully formal verification of OSs at source code level, we will replace the imperative language in π-Core by a more expressive one and add a verification condition generator (VCG) to reduce the cost of the verification.

References

1. The Zephyr Project. https://www.zephyrproject.org/, accessed: December 2018
2. Common Criteria for Information Technology Security Evaluation (v3.1, Release 5). https://www.commoncriteriaportal.org/ (April 2017)
3. Alkassar, E., Schirmer, N., Starostin, A.: Formal Pervasive Verification of a Paging Mechanism. In: Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 109–123. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
4. Blanchard, A., Kosmatov, N., Lemerre, M., Loulergue, F.: A Case Study on Formal Verification of the Anaxagoros Hypervisor Paging System with Frama-C. In: Proceedings of International Workshop on Formal Methods for Industrial Critical Systems. pp. 15–30. Springer International Publishing (2015)
5. Bolignano, P., Jensen, T., Siles, V.: Modeling and Abstraction of Memory Management in a Hypervisor. In: Proceedings of International Conference on Fundamental Approaches to Software Engineering (FASE). pp. 214–230. Springer Berlin Heidelberg (2016)
6. Chen, H., Wu, X., Shao, Z., Lockerman, J., Gu, R.: Toward Compositional Verification of Interruptible OS Kernels and Device Drivers. In: Proceedings of 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). pp. 431–447. ACM (2016)
7. Fang, B., Sighireanu, M.: Hierarchical Shape Abstraction for Analysis of Free List Memory Allocators. In: Proceedings of International Symposium on Logic-Based Program Synthesis and Transformation. pp. 151–167. Springer International Publishing (2017)
8. Fang, B., Sighireanu, M.: A refinement hierarchy for free list memory allocators. In: Proceedings of ACM SIGPLAN International Symposium on Memory Management. pp. 104–114. ACM (2017)
9. Fang, B., Sighireanu, M., Pu, G., Su, W., Abrial, J.R., Yang, M., Qiao, L.: Formal Modelling of List based Dynamic Memory Allocators. Science China Information Sciences 61(12), 103 – 122 (Nov 2018)
10. Gallardo, M.d.M., Merino, P., Sanán, D.: Model Checking Dynamic Memory Allocation in Operating Systems. Journal of Automated Reasoning 42(2), 229–264 (April 2009)
11. Gu, R., Shao, Z., Chen, H., Wu, X.N., Kim, J., Sjöberg, V., Costanzo, D.: CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In: Proceedings of 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI). pp. 653–669. USENIX Association, Savannah, GA (2016)
12. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., et al.: seL4: Formal Verification of an OS Kernel. In: Proceedings of 22nd ACM SIGOPS Symposium on Operating Systems Principles (SOSP). pp. 207–220. ACM Press (2009)
13. Klein, G., Tuch, H.: Towards Verified Virtual Memory in L4. In: Proceedings of TPHOLs Emerging Trends. p. 16 pages. Park City, Utah, USA (September 2004)
14. Leroy, X., Blazy, S.: Formal verification of a c-like memory model and its uses for verifying program transformations. Journal of Automated Reasoning 41(1), 1–31 (July 2008)
15. Mansky, W., Garbuzov, D., Zdancewic, S.: An Axiomatic Specification for Sequential Memory Models. In: Proceedings of International Conference on Computer Aided Verification (CAV). pp. 413–428. Springer International Publishing (2015)
16. Marti, N., Affeldt, R., Yonezawa, A.: Formal Verification of the Heap Manager of an Operating System Using Separation Logic. In: Proceedings of International Conference on Formal Engineering Methods (ICFEM). pp. 400–419. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)

17. Saraswat, V.A., Jagadeesan, R., Michael, M., von Praun, C.: A Theory of Memory Models. In: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP). pp. 161–172. ACM (2007)
18. Su, W., Abrial, J.R., Pu, G., Fang, B.: Formal Development of a Real-Time Operating System Memory Manager. In: Proceedings of International Conference on Engineering of Complex Computer Systems (ICECCS). pp. 130–139 (2016)
19. Tews, H., Völp, M., Weber, T.: Formal memory models for the verification of low-level operating-system code. Journal of Automated Reasoning 42(2), 189–227 (April 2009)
20. Vaynberg, A., Shao, Z.: Compositional Verification of a Baby Virtual Memory Manager. In: Proceedings of 2nd International Conference on Certified Programs and Proofs (CPP). pp. 143–159. Springer-Verlag, Berlin, Heidelberg (2012)
21. Ševčík, J., Vafeiadis, V., Zappa Nardelli, F., Jagannathan, S., Sewell, P.: CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. Journal of the ACM 60(3), 22:1–22:50 (June 2013)
22. Xu, F., Fu, M., Feng, X., Zhang, X., Zhang, H., Li, Z.: A Practical Verification Framework for Preemptive OS Kernels. In: Proceedings of 28th International Conference on Computer Aided Verification (CAV). pp. 59–79. Springer (July 2016)
23. Yu, D., Hamid, N.A., Shao, Z.: Building Certified Libraries for PCC: Dynamic Storage Allocation. In: Proceedings of European Symposium on Programming Languages and Systems (ESOP). pp. 363–379. Springer Berlin Heidelberg (2003)