Formal Verification for A Buddy Allocation Model Specification

1st Ke Jiang
School of Computer Science and Engineering
Nanyang Technological University
Singapore, Singapore
johnjiang@ntu.edu.sg

2nd Yongwang Zhao
School of Computer Science and Engineering
Beihang University
Beijing Advanced Innovation Center for Big Data and Brain Computing
Beihang University
Beijing, China
zhaoyw@buaa.edu.cn

3rd David Sanán
School of Computer Science and Engineering
Nanyang Technological University
Singapore, Singapore
sanan@ntu.edu.sg

4th Yang Liu
School of Computer Science and Engineering
Nanyang Technological University
Singapore, Singapore
yangliu@ntu.edu.sg

***Abstract*—Buddy allocation algorithms are widely adopted by memory management systems to manage the address space accessed by applications. However, errors in any stage of the development process of the memory management component, from the specification to the implementation, may lead to critical issues in other components using it. Rigorous mathematical proofs provide strong assurance to the development process. We apply formal methods to ensure the correctness in the specification of a buddy allocation algorithm. In this paper, we use the interactive theorem prover Isabell/HOL to construct a specification for a buddy allocation algorithm consisting on operations to allocate and dispose memory areas. Thence we verify that the operations preserve key invariants over the memory to guarantee functional correctness of the algorithm. Finally, we verify that the operations also preserve integrity of the memory, therefore they do not affect other memory areas previously allocated.**

***Index Terms*—Memory Specification, Formal Verification, Functional Correctness, Security.**

1. Introduction

Correctness of applications and libraries of a system using dynamic memory allocation relies on properties the memory have to preserve. Errors in any stage of the development process of the memory management component, may break those properties, leading to critical issues in the rest of the system. Along the last decades, formal methods have been successfully applied in the verification of critical systems. To improve confidence on the reliability of a memory management, verification of functional correctness and security properties is applied from the top specification layers down to the implementation and the machine code.

Formal verification has been applied on memory managers going from very abstract models to more concrete ones. For instance, the work in [6] formalizes an abstraction of the memory management in term of write and read operations. They prove sequential consistency over the abstraction, so the memory is expressed as ordered sequence of reads and writes. Also focusing on a high level of abstraction, the work in [9] provides a memory model for an imperative language, defining the necessary memory operations for the language at the specification and implementation levels. It defines an axiomatic reasoning framework, proving that the memory semantics satisfy such axiomatic rules. In a similar way [11], also provides an axiomatic proof system for a sequential memory model, bringing together an unified representation of the memory rules. In this sense, the work in [11] proves that their memory rules also satisfy the ones in [9].

For concrete managers, the work from [8] specifies a heap manager from the implementation level directly and verifies its functional correctness. It develops a library for separation logic to handle the pointers in C source codes and to ensure the separation of memory blocks. Another work from [2] specifies and verifies a memory allocation module of Contiki, where blocks are preallocated. The work specifies C source codes by Frama-C in memory structure and allocation and deallocation operations. Its verification detects errors like out-of-bounds accesses and potentially harmful situations by automatic provers like Frama-C/Wp. Also, work from [1] formally verifies the correctness of a heap allocator at the source code level. The C source codes are translated into semantics language in Isabelle/HOL using CParser and AutoCorres. A specification consists of allocation and deallocation operations is built and formally proved. The equality between semantics language and specification is also proved automatically.

In this paper, we develop a specification for buddy allocation algorithms in the Isabelle/HOL proof assistant. The buddy allocation algorithms provide to applications with two services for the allocation and disposal of memory blocks. Our specification consists of algorithm details as specific as possible for the sake of capturing any feature in the algorithms. Then we specify and prove a number of properties for functional correctness of the algorithm. Finally, we show that the memory services preserve integrity of other memory areas previously allocated.

The following section briefly introduces the background of Isabelle/HOL verification environment and buddy allocation algorithms. The next section is about the formalization of buddy allocation model including representation of our specification and proofs to properties for functional correctness. The verification of integrity for security is arranged in the following section. The last section is about the conclusions and future work.

2. Background

In this section we give the necessary background for the buddy allocation algorithms we provide the specification for, and the Isabelle/HOL theorem prover that we use for its specification and verification.

2.1. Buddy Allocation Algorithm

The buddy memory system was originally described by [4]. Today it is widely used in many operation systems, in particular most versions of UNIX. For example, BSD [7] uses the buddy memory system for blocks smaller than a page, i.e., 4 kilobytes. The classic description of the buddy system is released by [5].

According to the algorithms, free blocks on each size are maintained in a multilevel list, as a result it is easy to find a block of the requested size if one is available. If there is no block of the requested size, allocation operation will search for the first nonempty free list for blocks, starting from the level where the requested size is allocated. Then a large block which is picked from the nonempty free list is split, it is divided into two smaller blocks and each smaller block becomes an unique buddy to the other. If the size of smaller block is still too large, one of the two smaller blocks is split again. The split process stops until the requested size appears. Then one of the available blocks is marked as occupied and returned to the requesting application. The others are added to the appropriate free lists.

When a block is deallocated, the algorithms checks whether the block can be merged. A split block can only be merged with its unique buddy block, which then reforms the larger block they were split from. Among this process, the algorithms apply a flag bit strategy to quickly check if blocks belonging to the same parent are free, in order to decide whether to merge these blocks into one block. With this way, buddy memory system has small external fragmentation.

To implement buddy allocation algorithms, it applies two important data structures multilevel free linked-list and multilevel free bitmap. The block to be allocated or deallocated is directly picked from the head of linked-list or added into the tail. Bitmap uses 0 and 1 to implement the flag bit strategy. Fig. 1 describes a moment in memory system using these two date structures.

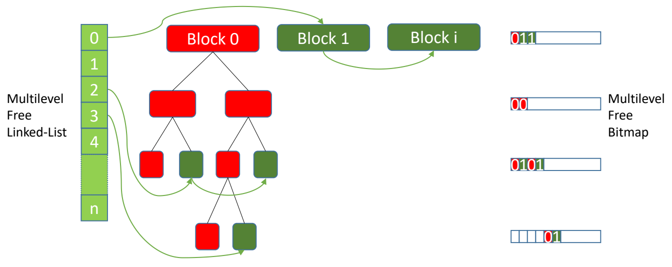

Figure 1. Structures in Buddy Allocation Algorithms

2.2. Isabelle/HOL

We use the interactive theorem prover Isabelle/HOL [10] to conduct the specification and verification of the memory management. Isabelle/HOL is a higher order logic theorem prover, using a typed lambda calculus-like functional language for specifications.

Isabelle/HOL includes a specification for simple common types such as naturals, integers, and booleans. It also specify some composed data types like tuples, records, lists, and sets that are parametrized on other types. Isabelle provides the interface *datatype* for the creation of user defined types based on type constructors.

Isabelle provides functions on predefined types to access their members or to provide additional operations over them. In the following we describe those functions that we use along this work. Tuples are denoted as $(t_1 \times t_2)$, projection function *fst* and *snd* respectively returns elements t_1 and t_2. Lists are defined as a datatype with an empty construct denoted with *NIL* or $[]$, and a concatenation construct denoted with $\#$, where $x\#xs$ adds x to the front of xs. The ith component of a list as is written as $as!i$. Isabelle/HOL provides functions for definite and indefinite descriptions. Definitive descriptions are represented by $THE\ x.\ P\ x$ and return the element uniquely described by the predicate P, else it returns and undefined value. Indefinite descriptions are represented by $SOME\ x.P\ x$ selecting a random element from the predicate P that must describe at least one element, else it returns an arbitrary value.

Isabelle/HOL allows user to create non-recursive specifications using the command *definition*, and to create recursive specifications using commands *primrec* and *recursive*.

3. Specification of Buddy Allocation Model

The specification of the buddy memory management consists of a model for the necessary data structures to represent the memory, and for the allocation and disposal operations that memory management provides. This specification follows the algorithms for the buddy memory management in Zephyr OS, which applies a quartering split over blocks.

3.1. State Representation

The specification begins with the structure of a quad-tree.

$$(set:\ 'a)\ tree\ =\ Leaf\ (L:\ 'a)\ |$$
$$Node\ (LL:\ 'a\ tree)\ (LR:\ 'a\ tree)\ (RL:\ 'a\ tree)\ (RR:\ 'a\ tree)$$

We use recursive method to construct a quad-tree. It has two forms: *Leaf* tree and *Node* tree. A *Node* tree is built by itself or a *Leaf* tree. The notation *Leaf* means the end of this construct process. Notations *LL*, *LR*, *RL* and *RR* return corresponding subtrees of the *Node* tree. Notation **set** means a function that gathers polymorphic notation *'a* of all the leaves as a collection.

$$\begin{aligned} block\_state\_type &= FREE\ |\ ALLOC \\ ID &= nat \\ Block &= (block\_state\_type\ \times\ ID)\ tree \end{aligned}$$

In this specification, we use tuple type (*block_state_type* $\times$ *ID*) to instantiate the polymorphic notation *'a* in the quad-tree structure. Type *block_state_type* stands for the usage state of a block. It consists of two subtypes indicated as *ALLOC* and *FREE* constructed by *datatype* function. Another type *ID* in basic type *nat* represents a range of address occupied by a memory block. Finally, type *Block* represents an instantiated quad-tree.

In addition, we create some auxiliary functions to manipulate the *block tree*. Function **get_level** takes two Blocks *btree* and *b* as inputs, then returns a nat *level* which represents the layer number that *b* locates in *btree* from the root whose layer number is 0. Function **allocsets** takes a Block *btree* and returns a Block set *aset* of all Leaf nodes whose *block_state_type* is *ALLOC* from *btree*. Function **freesets** has a similar definition but returns *FREE* ones. Function **freesets_level** takes a Block *btree* and a nat *level*, then returns a Block set *fset* of all Leaf nodes whose *block_state_type* is *FREE* and locate at *level* in *btree*. We use a notation *idset* to represent the collection of all used *IDs*. To create a new Leaf node, we have to pick up a new *ID* to mark it by the strategy of *SOME p.* $p \notin$ *idset*.

Before introducing allocation model, we create a function **output_level** that maps request sizes to the most suitable allocation levels in a quad-tree. The input parameters are a nat list *blo_list* and a nat *rsize*. Static linked list *blo_list* is used to store the size of blocks for each level in a quad-tree and its indexes represent the levels of a quad-tree. For example, the size of root is 1024*Mbytes* and the first level is 256*Mbytes*, then *blo_list*!0 is equal to 1024 and *blo_list*!1 is 256. The *blo_list* is a strictly decreasing list to simulate the fact that the smaller the level, the larger size the memory block. The function returns a nat index *l* in *blo_list* with these constrains: the size it represents has to be greater than or equal to the size of request block, and there is no smaller size that meets this condition. After that, the most suitable block size is picked up from *blo_list*, and then mapped to the correct level of the quad-tree by the index *l* in *blo_list*. We use *rlv* to represent the output. The definition of this mapping is as follows.

Definition 1 (Mapping Request Sizes to Allocation Levels).

$$\begin{aligned} &output\_level\ blo\_list\ rsize \triangleq THE\ l.\ l < |blo\_list| \\ &\wedge rsize \le blo\_list\ !\ l \\ &\wedge ((|blo\_list| > 1 \wedge l < |blo\_list| - 1) \longrightarrow rsize > blo\_list\ !\ (l+1)) \end{aligned}$$

3.2. Allocation Model

Now we specify allocation operation as function **alloc**. Firstly, we introduce two assistant functions. Function **exists_freelevel** takes a Block set *bset* and a nat *rlv* as inputs, then returns a bool result *re* represents whether there is a Block in *bset* (the collection of all quad-trees in memory system) that has such *FREE* Leaf nodes whose level is less than or equal to *rlv*. Function **freesets_maxlevel** has the same inputs, but returns a nat *lmax* represents the maximum level among all levels with *FREE* Leaf nodes but less than or equal to *rlv*. The definitions are as follows.

Definition 2 (existence of free blocks in a level).

$$\begin{aligned} &exists\_freelevel\ bset\ rlv \triangleq \exists l.\ l \le rlv \\ &\qquad\qquad \wedge \exists b \in bset.\ freesets\_level\ b\ l \ne \emptyset \end{aligned}$$

Definition 3 (maximum level of free blocks).

$$\begin{aligned} &freesets\_maxlevel\ bset\ rlv \triangleq THE\ lmax.\ lmax \le rlv \\ &\wedge \exists b \in bset.\ freesets\_level\ b\ lmax \ne \emptyset \\ &\wedge (\forall l \le rlv.\ \exists b \in bset.\ freesets\_level\ b\ l \ne \emptyset \longrightarrow l \le lmax) \end{aligned}$$

During the allocation process, if no such Leaf node right in the requested level exists but bigger Leaf nodes exist, then it is necessary to split a bigger Leaf node into small Leaf nodes until a Leaf node that satisfies the request appears. Function **split** divides a Leaf node *b* into a Node *btree* by recursion for a nat *lv* times. It invokes a function **divide** that takes a Leaf node *b* and returns a new non-terminal Node *n* with four terminal Leaf nodes. The leftmost Leaf of *n* is marked as *ALLOC*, while the rests are marked as *FREE*. The division operation always conducts on the leftmost subtree. A simple example describing this process is showed in Fig. 2 until *FREE* Leaf in the request level appears and then be allocated. We define the *split* operation as follows.

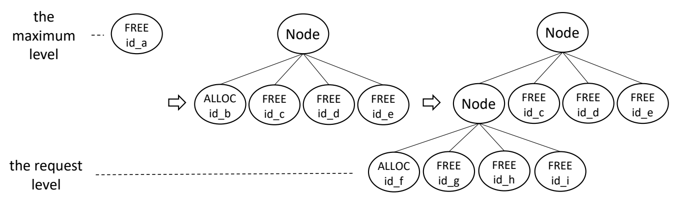

Figure 2. The progress of dividing a free leaf

Definition 4 (splitting a leaf).

```
$split\ b\ lv \triangleq if\ lv = 0\ then\ b$
  $else\ Node\ (split\ (LL\ divide\ b)\ (lv - 1))\ (LR\ divide\ b)$
              $(RL\ divide\ b)\ (RR\ divide\ b)$
```

In addition, we give functions **set_type**, **replace** and **replace_leaf** as follows: function *set_type* takes a Leaf node *b* and a target block_state_type *s*, then changes the type of *b* with *s*, finally returns it as a new Leaf node *b'*. function *replace* takes a Block *btree*, two Leaf nodes *b* and *b'* as inputs, then replace the Leaf node *b* with *b'* in *btree*, finally returns the updated Block *btree'*. function *replace_leaf* takes a Block *btree*, a Leaf node *b* and a Node *btr* as inputs, then replace the Leaf node *b* with *btr* in *btree*, lastly returns the updated Block *btree'*.

Now we give the process of allocation operation. It firstly checks whether there is Block in *bset* that has such *FREE* Leaf nodes whose level is less than or equal to *rlv* by function *exists_freelevel*. If it returns *False* then the allocation process stops and returns original *bset* and *False*. Otherwise, function *freesets_maxlevel* returns the maximum level *lmax* among all levels with *FREE* Leaf nodes but less than or equal to *rlv*. Two branches are as follows.

If the *lmax* is equal to requested level *rlv* then: randomly pick up such a Block as *btree*; pick up such a *FREE* Leaf node *l* in level *rlv* from *btree*; invoke *set_type* to set *l* type *ALLOC* as *l'*; invoke *replace* to update *btree* with *l'* as *btree'*; finally return updated Block set *bset* with *btree'* and *True*.

If the *lamx* is not equal to requested level *rlv* then: randomly pick up such a Block as *btree* who has *FREE* Leaf nodes in level *lmax*; pick up such a *FREE* Leaf node *l* in level *lmax* from *btree*; invoke *split* to split *l* into Node *btr*; invoke *replace_leaf* to update *btree* with *btr* as *btree'*; finally return updated Block set *bset* with *btree'* and *True*. The definition of allocation operation is as follows.

Definition 5 (Allocation Operation).

```
$alloc\ bset\ rlv \triangleq$
    $if\ exists\_freelevel\ bset\ rlv\ then$
        $lmax = freesets\_maxlevel\ bset\ rlv$
        $if\ lmax = rlv\ then$
            $btree = SOME\ b.\ b \in bset \wedge freesets\_level\ b\ rlv \neq \emptyset$
            $l = SOME\ l.\ l \in freesets\_level\ btree\ rlv$
            $btree' = replace\ btree\ l\ (set\_type\ l\ ALLOC)$
            $return\ (bset - \{btree\} \cup \{btree'\}, True)$
        $else$
            $btree = SOME\ b.\ b \in bset \wedge freesets\_level\ b\ lmax \neq \emptyset$
            $l = SOME\ l.\ l \in freesets\_level\ btree\ lmax$
            $btr' = split\ l\ (rlv - lmax)$
            $btree' = replace\_leaf\ btree\ l\ btr'$
            $return\ (bset - \{btree\} \cup \{btree'\}, True)$
    $else\ return\ (bset, False)$
```

3.3. Deallocation Model

During the deallocation process, if such a situation that four Leaf nodes belonging to a same parent Node are all *FREE* appears, function **merge** is invoked to merge these Leaf nodes to one Leaf node. It invokes a function **combine** that takes a Node *n* and returns a new terminal Leaf node *l iff* Node *n* has four terminal *FREE* Leaf nodes. Function *merge* takes a Node *n* and recursively invokes *combine* on each sub-node of *n* and *n* itself. A simple example that describes the merging process is showed in Fig. 3.

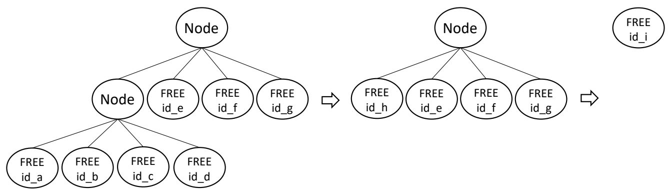

Figure 3. The progress of merging all free memory blocks

With above discussion, we give the process of deallocation operation. It firstly checks whether there is a Block in *bset* that the Leaf node *b* to be deposed belongs to it. If there is no such tree, the procedure returns the original *bset* and *False*. Otherwise, if the type of *b* is *FREE*, it also returns the original *bset* and *False*. When all conditions are met, it picks up the Block in *bset* that *b* belongs to as *btree*; invokes *set_type* to set *b* type *FREE* as *b'*; invokes *replace* to update *btree* with *b'* as *btree'*; invokes *merge* to perform a necessary merging operation on *btree'* as *btree"*; finally returns updated Block set *bset* with *btree"* and *True*. The definition of deallocation operation is as follows.

Definition 6 (Deallocation Operation).

```
$free\ bset\ b \triangleq$
    $if\ \exists btree \in bset.\ b \in set\ btree\ then$
        $if\ fst\ b = FREE\ then$
            $return\ (bset, False)$
        $else$
            $btree = THE\ t.\ t \in bset \wedge b \in set\ t$
            $btree' = replace\ btree\ b\ (set\_type\ b\ FREE)$
            $btree'' = merge\ btree'$
            $return\ (bset - \{btree\} \cup \{btree''\}, True)$
    $else\ return\ (bset, False)$
```

At this point, we have done the specification for the buddy memory algorithms. Next, we are going to verify its functional correctness and security property.

4. Verification

In this section we show in Section 4.1 that the formal specification introduced in the previous section preserves a set of properties guaranteeing its functional correctness. As a security property, we also prove in Section 4.2 and Section 4.3 that the specification preserves integrity of the memory structures.

4.1. Functional correctness

We use preconditions & postconditions and invariants to depict and trace the executions of the specification for functional correctness purposes. We give preconditions and postconditions in the first place.

The following Lemma 1 gives such an implication: if any quad-tree Block in *blo_set* cannot satisfy the precondition: the level of its *FREE* leaf nodes is less than or equal to the value of *rlv*, then nothing is changed result from allocation failure.

Lemma 1 (Allocation Failure).

$$\neg exists\_freelevel\ blo\_set\ rlv \longrightarrow fst\ (alloc\ blo\_set\ rlv) = blo\_set$$

Then Lemma 2 and 3 respectively describes a transition during a direct allocation process that the *FREE* leaf node to be allocated no longer belongs to free sets and is a part of allocated sets. A direct allocation precess means that the existence of such a quad-tree Block in *blo_set*, the level of whose *FREE* leaf nodes is less than or equal to the value of *rlv*. In particular, the maximum level among these *FREE* leaf nodes is equal to the value of *rlv*, therefore the function picks up a *FREE* leaf node in level *rlv* to allocate.

Lemma 2 (Freesets for Direct Allocation).

$$\begin{aligned}&(exists\_freelevel\ blo\_set\ rlv \land freesets\_maxlevel\ blo\_set\ rlv = rlv)\\ &\longrightarrow (\exists l.\ l \in freesets\ blo\_set \land l \notin freesets\ fst\ (alloc\ blo\_set\ rlv)\\ &\quad \land freesets\ blo\_set = freesets\ fst\ (alloc\ blo\_set\ rlv) \cup \{l\})\end{aligned}$$

Lemma 3 (Allocsets for Direct Allocation).

$$\begin{aligned}&(exists\_freelevel\ blo\_set\ rlv \land freesets\_maxlevel\ blo\_set\ rlv = rlv)\\ &\longrightarrow (\exists l.\ l \notin allocsets\ blo\_set\ \land l \in allocsets\ fst\ (alloc\ blo\_set\ rlv)\\ &\quad \land allocsets\ fst\ (alloc\ blo\_set\ rlv) = allocsets\ blo\_set \cup \{l\})\end{aligned}$$

Next Lemma 4 introduces a transition during an indirect allocation process. The precondition preserves the existence of such a quad-tree Block in *blo_set*, the level of whose *FREE* leaf nodes is less than or equal to the value of *rlv*. However, the maximum level among these *FREE* leaf nodes is less than the value of *rlv*. With this precondition, *split* operation is invoked to divide a bigger Leaf node into small Leaf nodes until a Leaf node that satisfies the request appears. Then postcondition guarantees that a new Leaf node is added into allocated set while it dose not belong to this set previously.

Lemma 4 (Allocsets for Indirect Allocation).

$$\begin{aligned}&(exists\_freelevel\ blo\_set\ rlv \land freesets\_maxlevel\ blo\_set\ rlv \neq rlv)\\ &\longrightarrow (\exists l.\ l \notin allocsets\ blo\_set \land l \in allocsets\ fst\ (alloc\ blo\_set\ rlv)\\ &\land allocsets\ fst\ (alloc\ blo\_set\ rlv) = allocsets\ blo\_set \cup \{l\})\end{aligned}$$

The next two Lemma 5 and 6 guarantee nothing is changed during the deallocation failure processes. This consequence is result from the Leaf node to be released does not belong to any quad-tree Block, or the type of the Leaf node to be freed is *FREE* already.

Lemma 5 (Deallocation failure 1).

$$\nexists btree \in blo\_set.\ b \in set\ btree \longrightarrow fst\ (free\ blo\_set\ b) = blo\_set$$

Lemma 6 (Deallocation failure 2).

$$\begin{aligned}&\exists btree \in blo\_set.\ b \in set\ btree \land fst\ b = FREE\\ &\longrightarrow fst\ (free\ blo\_set\ b) = blo\_set\end{aligned}$$

The last Lemma 7 ensures a correct transition during a deallocation success process. It describes that the Leaf node which is to be released dose not belong to allocated set any more.

Lemma 7 (Allocsets for Deallocation Success).

$$\begin{aligned}&\exists btree \in blo\_set.\ b \in set\ btree \land fst\ b \neq FREE\\ &\longrightarrow allocsets\ blo\_set = allocsets\ fst\ (free\ blo\_set\ b) \cup \{b\}\end{aligned}$$

After giving these lemmas for preconditions & postconditions, we prove that our buddy allocation specification satisfies them so that it meets the functional expectations. The first theorem we prove as follows.

Theorem 1. The buddy allocation specification satisfies all the lemmas for preconditions & postconditions above.

Next, we introduce invariants to guarantee functional correctness of the specification from another point of view. Firstly, we ensure that the allocation operation picks out *The Most Suitable Leaf Node* in Definition 1. Two properties are proved: the correctness of the mapping function from the size of requested memory block to the level of the quad-tree; the correctness of the quad-tree hierarchical structure. Here are these two properties.

Two lemmas ensure the correctness of Definition 1.

Lemma 8 (Correctness of Function output_level 1).

$$\begin{aligned}&|blo\_list| > 0 \land rsize \leq blo\_list\ !\ (|blo\_list| - 1)\\ &\longrightarrow output\_level\ blo\_list\ rsize = |blo\_list| - 1\end{aligned}$$

Lemma 9 (Correctness of Function output_level 2).

$$\begin{aligned}&|blo\_list| > 1 \land l < |blo\_list| - 1\\ &\land rsize \leq blo\_list\ !\ l \land rsize > blo\_list\ !\ (l+1)\\ &\longrightarrow output\_level\ blo\_list\ rsize = l\end{aligned}$$

With these two lemmas, we make sure the correctness of the mapping function *output_level*. Then we introduce a lemma alone to prove the correctness of the hierarchical structure of a quad-tree. Function **root** checks whether the tree is a Root tree and function **child** gives us a set of all immediate child nodes of a Node tree.

Lemma 10 (Hierarchical Structure of a Quad-tree).

$$\begin{aligned}&(root\ btree \longrightarrow get\_level\ btree = 0)\\ &\land (get\_level\ btree = l \land l \geq 0 \land chtree \in child\ btree\\ &\quad \longrightarrow get\_level\ chtree = l + 1)\end{aligned}$$

Until now, we have already proved the correctness of the mapping function *output_level* and the hierarchical structure of a quad-tree. Then we can deduce the following theorem to guarantee the property of picking out *The Most Suitable Leaf Node*.

Theorem 2. The buddy allocation specification picks out the most suitable Leaf node and allocate it on the correct level in a quad-tree.

The buddy allocation algorithms may reduce the fragmentation by invoking merge function during the process of deallocation. Whether this function is executed correctly can not be proved straightway. In order to prove this, we still consider the correctness of the structure to a quad-tree during allocation and deallocation processes. Considering the fact that there is not such a Node tree whose four immediate child nodes are all Leaf nodes and their types are *FREE*, a definition is constructed as follows to check whether a quad-tree is such a Node tree. The function **leaf** is to check whether the tree is a Leaf node.

Definition 7 (Four Free Leaves Belong to The Same Node).

$$is\_FFL\ btree \triangleq \forall chtree \in child\ btree.\ leaf\ chtree \\ \wedge fst\ chtree = FREE$$

Fig. 3 can explain this definition well. As the first picture shows, the subtree in the lower left corner is such a Node tree because its four child nodes are all Leaf nodes and their types are *FREE*. Therefore, merge function is necessary to handle this situation. The followings are lemmas that ensure the non-existence of such *FFL* trees after allocation and deallocation operations if they guarantee non-existence of such *FFL* trees in the assuming of implication expressions.

Lemma 11 (Non-existence of FFL during Allocation).

$$\forall b \in blo\_set.\ \neg\ is\_FFL\ b \\ \longrightarrow \forall b \in fst\ (alloc\ blo\_set\ rlv).\ \neg\ is\_FFL\ b$$

Lemma 12 (Non-existence of FFL during Deallocation).

$$\forall b \in blo\_set.\ \neg\ is\_FFL\ b \\ \longrightarrow \forall b \in fst\ (free\ blo\_set\ b).\ \neg\ is\_FFL\ b$$

We consider such a situation: operation system initialize the whole memory into a series of free blocks in different sizes, and these memory blocks are in the form of root nodes which are not split at the very beginning. This initialization state satisfies not-existence of *FFL* trees. Then according to Lemma 11 and 12, any state after initialization satisfies non-existence of *FFL* trees no matter operation system performs allocation or deallocation operations. Therefore, we can make sure the whole memory system preserves this property. We have this theorem as follows. Furthermore, it proves that the buddy allocation specification reduces the fragmentation by merge operation.

Theorem 3. The buddy allocation specification guarantees non-existence of FFL trees among all memory blocks.

In the end, we prove two significant properties: memory isolation and non-leakage. The first one is to ensure non-existence of the overlap in address spaces. The isolation makes sure that memory blocks of a domain can not be maliciously overwritten by others. Another property the non-leakage guarantees that available memory blocks (including occupied and free ones) are not getting less and less. In other words, it protects the integrity of address spaces.

Now we begin with the memory isolation. We provisionally use type *ID* in basic type *nat* to represent a contiguous address occupied by a memory block in Section 3.1. To link *ID* to a real address, two things have to be introduced and proved: 1. a mapping function between a *ID* and a real address as well as its correctness; 2. the one-to-one uniqueness between a *ID* and a real address. We leave this part to the further work for a more detailed specification proof, in other words the design level of the buddy allocation. In this paper, we are not going to introduce real addresses into this specification, and we consider the correctness of the mapping function and promise its the uniqueness. Therefore, with these assumptions, isolation of address spaces means that all *IDs* which leaf nodes brings with are unique.

The following definition introduces a judgment whether two Leaf nodes have the same *ID*. Function **ID** gives the *ID* a Leaf node brings with.

Definition 8 (Different IDs).

$$is\_different\ blo\_set \triangleq \\ \forall b \in blo\_set.\ \forall l \in set\ b.\ (\nexists l'.\ l' \in set\ (SOME\ b.\ b \in blo\_set) \\ \wedge l' \neq l \wedge ID\ l' = ID\ l)$$

Below are two lemmas that ensure this property holds during the procedures of allocation and deallocation if it holds in the assuming of implication expressions.

Lemma 13 (Different IDs during Allocation).

$$is\_different\ blo\_set \longrightarrow is\_different\ fst\ (alloc\ blo\_set\ rlv)$$

Lemma 14 (Different IDs during Deallocation).

$$is\_different\ blo\_set \longrightarrow is\_different\ fst\ (free\ blo\_set\ b)$$

Let us go back to the initialization state. The whole memory is divided into a finite number of free root nodes. We can easily guarantee that all *IDs* these root nodes bring with are unique and initialize the collection *idset* with these used *IDs*. Then according to the lemmas above, any state after initialization satisfies *Different IDs* no matter operation system performs allocation or deallocation operations. As a result, we can guarantee the whole memory system preserve *Different IDs*. We have this theorem as follows.

Theorem 4. The buddy allocation specification ensures all IDs of Leaf nodes are unique.

Finally, with the assumptions that the correctness of the mapping function between a *ID* and a real address and promising its uniqueness, we prove the specification the property of the isolation of memory address spaces.

Next is for the non-leakage of memory blocks. We use a quad-tree structure and map all the blocks into the Leaf nodes of these trees, thence the non-leakage means that all the Leaf nodes (including occupied and free ones) are recorded in use. If we can infer a correct and certain relation between the number of Leaf nodes and Non-leaf nodes of a quad-tree, we can prove that all the Leaf nodes are in use and none of them is forgotten. To achieve this, the first step is to search and prove a certain relation between the numbers of Leaf nodes and Non-leaf nodes in a quad-tree. Through exploration, we have found and proved such a relation as follows. Functions **get_leaf** and **get_node** take a Block and return Leaf nodes set and Non-leaf nodes set respectively.

Lemma 15 (Relation of a Quad-tree Nodes).

$$Qtree\ b:\ size\ (get\_leaf\ b) = size\ (get\_node\ b) \times 3 + 1$$

After establishing this relation, we use the following two lemmas to guarantee all the quad-trees during execution procedures maintain this relation.

Lemma 16 (Qtree during Allocation).

$$\forall b \in blo\_set.\ Qtree\ b \longrightarrow \forall b \in fst\ (alloc\ blo\_set\ rlv).\ Qtree\ b$$

Lemma 17 (Qtree during Deallocation).

$\forall b \in blo\_set.\ Qtree\ b \longrightarrow \forall b \in fst\ (free\ blo\_set\ b).\ Qtree\ b$

In the end, we can prove that all quad-trees hold this relation between the number of Leaf nodes and Non-leaf nodes. The correct structure of a quad-tree means that all Leaf nodes (including occupied and free ones) are in use. We have this theorem as follows. Then considering the fact that all blocks are mapped into the Leaf nodes of these trees, we can ensure that the specification preserves memory non-leakage.

Theorem 5. The buddy allocation specification guarantees any Block is a Qtree.

To sum up, in this subsection we introduce preconditions & postconditions as well as significant invariants to ensure the formal buddy allocation specification preserves a set of properties which guarantee its functional correctness. In next two subsections, we build a security model and prove the specification preserves integrity of the memory structures.

4.2. A Security Model

Integrity is the assurance that the information is trustworthy and accurate. To achieve this, data must not be changed in transit. In this section, we try to prove the integrity property for buddy memory model. For this purpose, we firstly design a security model which consists of a nondeterministic state machine and the integrity property conditions. Next, we introduce interfaces into the buddy memory model, and package it into an event specification. We think of this event specification as an instantiated security model. The last step is to prove the instantiated model satisfy the integrity property. This part of work is following the work form [12].

4.2.1. Memory State Machine. is designed to be event based. Thus, $\mathcal{S}$ represents the state space and $\mathcal{E}$ is the set of event labels. The state-transition function is characterized by φ, which has the form of $\varphi : \mathcal{E} \to \mathbb{P}(\mathcal{S} \times \mathcal{S})$. The state machine must execute from a initial state, therefore, $s_0 \in \mathcal{S}$ which is on behalf of the initial state must be included in this machine. The definition of this state machine is as follows.

Definition 9 (State Machine). $\mathcal{M} = \langle \mathcal{S}, \mathcal{E}, \varphi, s_0 \rangle$

Based on the state machine above, we introduce some auxiliary functions: The **execution(s, es)** function returns the set of final states by executing a sequence of of events *es* from a state *s*. The **reachable(s)** function (denoted as $\mathcal{R}(s)$) checks the reachability of a state *s* by the *execution* function.

Next, we add the concept of partitions to represent the entities that execute the state-transition function. Therefore, partitions are the basic domains. In addition, we introduce partition scheduling as a domain **scheduler**. And we give the strict restriction that *scheduler* cannot be interfered by any other domains. Its aim is to ensure that *scheduler* does not leak information by its scheduling decisions. Therefore, the domains ($\mathcal{D}$) in $\mathcal{M}$ are the configured partitions ($\mathcal{P}$) and the scheduler ($\mathbb{S}$), $\mathcal{D} = \mathcal{P} \cup \{\mathbb{S}\}$. The **dom(s, e)** function gives which partition is currently executing *e* in the state *s* by consulting the *scheduler*.

4.2.2. Integrity Definition. is referenced to [3] which provides a formalism for the specification of security policies. The main idea in this article is that domain *u* is non-interfering with domain *v* if no action performed by *u* can influence the subsequence outputs seen by *v*. According to this, we use the concepts of state equivalence and interfering to construct integrity property.

Firstly, state equivalence (denoted as $\sim$) means that states are identical for a domain seen by it. For example, some certain collections that accessed only by a domain are indistinguishable at two different states. We use $s \overset{d}{\sim} t$ to represent *s* and *t* are identical for domain *d*.

By the concept of state equivalence, interfering (denoted ad $\rightsquigarrow$) means that the state equivalence of some domain is broken due to the operations by another domain. And $\not\rightsquigarrow$ is the opposite relation of $\rightsquigarrow$. Since the *scheduler* can schedule other domains, it can interfere with them. However, the *scheduler* cannot be interfered by any other domains to ensure that the *scheduler* does not leak information by its scheduling decisions.

With these two concepts, we can easily define the integrity property conditions as follows.

Definition 10 (Integrity Property Conditions).
IPC(e) $\equiv \forall$d s s'. $\mathcal{R}(s) \land$ dom(s, e) $\not\rightsquigarrow$ d $\land$ (s, s') $\in \varphi(e) \longrightarrow (s \overset{d}{\sim} s')$

From the conditions, if the domain being scheduled to run promises not to interfere other domains, then the consequences in final state seen by other domains are identical, that is to say the domain being scheduled to run has only access to its own space. The integrity property conditions do guarantee that the information is trustworthy and accurate.

4.2.3. Security Model. is defined as follows based on the discussion above.

Definition 11 (Security Model). $\mathcal{S}\_\mathcal{M} = \langle \mathcal{M}, \mathcal{D}, dom, \rightsquigarrow, \sim \rangle$
with assumptions as follows.

1) $\forall$d $\in \mathcal{D}$. $\mathbb{S} \rightsquigarrow$ d
2) $\forall$d $\in \mathcal{D}$. d $\rightsquigarrow \mathbb{S} \longrightarrow$ d = $\mathbb{S}$
3) $\forall$s t e. $s \overset{\mathbb{S}}{\sim} t \longrightarrow$ dom(s,e) = dom(t,e)
4) $\forall$s e. $\mathcal{R}(s) \longrightarrow \exists$s'. (s, s') $\in \varphi(e)$
5) $\forall$e. IPC(e)

This security model constructs a sequential model for event-based specification to verify the integrity property. Next, we will instantiate this security model with our buddy allocation model. We are going to find that if our buddy allocation model satisfies the integrity property conditions.

4.3. Instantiation and Security Proofs

In this part, we instantiate a security model in the following ways: Setting a global state; Adding interfaces to allocation and deallocation operations and instantiating the events with scheduler; Adding state-transition function by interfaces; Instantiating the definitions of interfering and equivalence.

4.3.1. Instantiation. As a global state, it records all the information like domains and all kinds of resources. *State* mainly consists of the currently running domain (denoted by **Cur**) and the memory address spaces occupied by the partition (characterized by function **Par_Mem: partition $\rightharpoonup$ Mem_Add**, *Mem_Add* is a set of *ID*).

For the allocation and deallocation operations, it is only necessary to update *Par_Mem* information according to the success or failure of the operations, thus forming new operations **alloc_memory** and **free_memory**. These two interfaces are defined as follows. In addition, a scheduler that arbitrarily selects partitions to execute is defined as follows.

Definition 12 (Allocate Memory).
alloc_memory s $\equiv$ if (alloc successes) then (update Par_Mem s (Cur s)) else s

Definition 13 (Deallocate Memory).
free_memory s $\equiv$ if (free successes) then (update Par_Mem s (Cur s)) else s

Definition 14 (Scheduler).
scheduler $\equiv$ (Cur s = SOME p. p $\in$ par_set)

With these interfaces functions, then we give the state-transition function **exec_event(e)** as a instantiation of φ. Thus event *e* represents *alloc_memory*, *free_memory* and *scheduler*.

Definition 15 (State-transition).
exec_event e $\equiv$ {(s, s'). s' $\in$ {(e s)} $\wedge$ e $\in$ {alloc_memory, free_memory, scheduler}}

Finally, we give the instantiation of integrity property we want to prove through the instantiations of $\rightsquigarrow$ and $\sim$. In this case, our goal is to prove that if the currently running domain is not interfering other domains, then the memory address spaces other domains owns maintain identical. To be more specific, operations conducted by any domains except for itself or *scheduler* will not change other domains' memory areas. Then we give the instantiations of $\rightsquigarrow$ and $\sim$ to achieve it.

Definition 16 (Instantiation of $\rightsquigarrow$ by Domain).
d1 $\rightsquigarrow$ d2 $\equiv$ (d1 = d2) $\vee$ is_scheduler d1

Definition 17 (Instantiation of $\sim$ by State and Domain).
$s \overset{d}{\sim} t$ $\equiv$ (is_scheduler d $\longrightarrow$ Cur s = Cur t) $\wedge$ (is_partition d $\longrightarrow$ Par_Mem s d = Par_Mem t d) $\wedge$ True

4.3.2. Security Proofs. To prove that the instantiated model is a security model, we have to prove item 1 to item 5 in $\mathcal{S}\_\mathcal{M}$ definition one by one. The first two assumptions are preserved by the interfering $\rightsquigarrow$ definition. The assumption 3 is preserved by $\sim$ for the scheduler. The assumption 4 of reachability is preserved with the function *exec_event(e)* by the following lemma.

Lemma 18. $\forall$s e. $\mathcal{R}(s)$ $\longrightarrow$ $\exists$s'. (s, s') $\in$ exec_event(e)

Next to prove the last assumption *IPC(e)*, we have to apply concrete conditions of *alloc_memory*, *free_memory* and *scheduler* to imply these events satisfy the integrity property conditions. Following are the lemmas of each concrete function. Functions **is_partition** and **is_scheduler** check the currently running domain is a partition or a scheduler.

Lemma 19 (IPC(e) of alloc_memory).
$\forall$d s s'. $\mathcal{R}(s)$ $\wedge$ is_partition (Cur s) $\wedge$ (Cur s) $\not\rightsquigarrow$ d $\wedge$ s' = alloc_memory s $\longrightarrow$ $s \overset{d}{\sim} s'$

Lemma 20 (IPC(e) of free_memory).
$\forall$d s s'. $\mathcal{R}(s)$ $\wedge$ is_partition (Cur s) $\wedge$ (Cur s) $\not\rightsquigarrow$ d $\wedge$ s' = free_memory s $\longrightarrow$ $s \overset{d}{\sim} s'$

Lemma 21 (IPC(e) of scheduler).
$\forall$d s s'. is_scheduler (Cur s) $\wedge$ (Cur s) $\not\rightsquigarrow$ d $\longrightarrow$ $s \overset{d}{\sim} s'$

Theorem 6. IPC(e) Satisfaction
$\forall$e $\in$ {alloc_memory, free_memory, scheduler}. IPC(e)

In the end, we proved that the event specification based on the buddy allocation model satisfies the integrity property and is a security model. According to the instantiation of interfering, by the buddy allocation model, as long as the domain is not the scheduler, the execution of one domain will not influence the memory address spaces that other domains have. The security proofs above ensure that that memory information is trustworthy and accurate.

References

[1] A. Sahebolamri, S. Constable, S. J. Chapin. A Formally Verified Heap Allocator, Electrical Engineering and Computer Science, 2018, p. 182.

[2] F. Mangano, S. Duquennoy, N. Kosmatov. Formal verification of a memory allocation module of Contiki with FRAMA-C: A case study, the 11th International Conference on Risks and Security of Internet and Systems, CRISIS, 2016, p. 114120.

[3] J. Goguen, J. Meseguer. Security Policies and Security Models, Proceedings of the IEEE Computer Society Symposium on Research in Security and Privacy, 1982, p. 11-20.

[4] K. C. Knowlton. A fast storage allocator, Commun. ACM, 1965, p. 623-624.

[5] Donald E. Knuth. Dynamic storage allocation. In The Art of Computer Programming, volume 1, section 2.5, pages 435455. Addison-Wesley, 1968.

[6] L. Higham, J. Kawash, N. Verwaal. Defining and comparing memory consistency models, Proceedings of the 10th International Conference on Parallel and Distributed Computing Systems, 1997, p. 349356.

[7] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. The Design and Implementation of the 4.4 BSD Operating System. Addison-Wesley, 1996.

[8] N. Marti, R. Affeldt, A. Yonezawa. Formal Verification of the Heap Manager of an Operating System Using Separation Logic, International Conference on Formal Engineering Methods, ICFEM: Formal Methods and Software Engineering, 2006, p. 400-419.

[9] S. Blazy, X. Leroy. Formal Verification of a Memory Model for C-Like Imperative Languages, ICFEM: Formal Methods and Software Engineering, 2005, p. 280-299.

[10] T. Nipkow, L. C. Paulson, and M. Wenzel. Isabelle/HOL-A Proof Assistant for Higher-Order Logical, volume 2283 of LNCS. Springer-Verlag, 2002.

[11] W. Mansky, G. Dmitri, S. Zdancewic. An Axiomatic Specification for Sequential Memory Models, Computer Aided Verification, July 2015, p. 413-428.

[12] Y. Zhao, D. Sanan, F. Zhang, Y. Liu. Refinement-based Specification and Security Analysis of Separation Kernels, IEEE Transactions on Depandable and Secure Computing, Volume 16, Issue 1, January 2019, p. 127-141.