Formal Verification for A Buddy Allocation Model Specification *

Ke Jiang[1], Yongwang Zhao[2,3], David Sanán[1], and Yang Liu[1]

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

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. We apply formal methods to ensure the absence of any misbehavior. Rigorous mathematical proofs provide strong assurance to the development process. In this paper, we firstly present a specification for the buddy allocation algorithm. And thence we validate enough properties to guarantee functional correctness of this algorithm. Finally, we construct execution traces to verify the integrity for security. Through these efforts, we propose a buddy allocation model that provides both functional correctness and security. Also, we use interactive theorem prover Isabelle/HOL to carry out the verification work.

Keywords: Memory Specification · Formal Verification · Functional Correctness · Security.

1 Introduction

In the past several decades, buddy allocation algorithms have been applied in the memory management systems. Errors in any stage of the development process of the memory management component may lead to critical issues in other components who invoke it. To improve confidence on the reliability of the development process, verification of functional correctness and security properties is applied into each stage, from the specification level to the implementation level, even the machine code level. Formal methods have been successfully applied in the verification of many critical systems due to these methods provide strong assurance by rigorous mathematical proofs. Therefore, we apply formal methods

* Supported by organization x.

to ensure the absence of any misbehavior during the development process of the buddy allocation model.

here is related work.

In this paper, we propose a buddy allocation model, which supports both functional correctness and security. To achieve this goal, we develop a specification for buddy allocation algorithms by functional programming in Isabelle/HOL proof assistant. Our specification must consists of algorithm details as specific as possible for the sake of capturing any feature in the algorithms. Then we design a series of properties for functional correctness of the algorithm. After that, we apply theorem-proving method to prove these properties. To verify integrity for security, we construct execution traces based on event, and then prove that the operation of one domain on memory does not affect any other domains. So far, we have completed the construction and verification of the buddy allocation algorithm.

The following section briefly introduces the Isabelle/HOL verification environment. 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 Isabelle/HOL Verification Environment

Our specification and verification work is based on the interactive theorem prover Isabella/HOL. HOL represents the Higher-Order Logical and Isabelle is its generic interactive theorem prover. In Isabelle/HOL, it is usual to employ functional programming method to define a function and to adopt theorem proving technique to reason a lemma or a theorem. For a gentle introduction to Isabelle/HOL see [1].

Apart from commonly used types like *bool* and *nat*, Isabelle offers notion *datatype* to create a distinguished element to some existing types. Projection functions *fst* and *snd* comes with the tuple $(t_1 \times t_2)$. Notions *list* and *set* are used as constructors to create a collection of same type. Operation 'cons' denoted by '#' on a list means that adding an element to the head of list. The ith component of a list *xs* is written as *xs*!*i*. *SOME x* and *THE x* in the *set A* represent choosing an element arbitrarily, existing and unique element respectively. Furthermore, λ-abstractions are also contained in terms like $\lambda(x, y)$. *f x y*.

Non-recursive definition can be made with the **definition** command and the **primrec** function definition is used for primitive recursions. The notation [$A_1;\ldots;A_n] \Longrightarrow$ A represents an implication with assumptions $A_1;\ldots;A_n$ and conclusion A. Isabelle mainly employs backward deduction, which means to prove the main goal, we must firstly prove subgoals which are decomposed from the main goal. It uses the rules of the reasoning like introduction, elimination, destruction rules, etc., as well as automatic provers such as *SMT*.

3 Formalization of Buddy Allocation Model

The formalization of buddy allocation models consists of a specification for algorithms and proofs to properties for functional correctness. The challenge is that we apply quartering from Zephyr OS for buddy allocation operations, which brings complexity to proofs.

3.1 Memory Model Specification

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

$$\begin{aligned}(set: 'a)\ tree = & Leaf\ (L: 'a)\ |\\ & Node\ (LL: 'a\ tree)\ (LR: 'a\ tree)\ (RL: 'a\ tree)\ (RR: 'a\ tree)\\ & for\ map: tree\_map\end{aligned}$$

The quad-tree constructed by induction contains two pieces of information: the memory block state (indicated as *ALLOC* and *FREE* respectively); a address tag occupied by a memory block (indicated as *ID*). The mapping function **set** assists to collect the leaves from a tree. With the help of the block state and the function **get_level**, allocated leaves or free leaves from different levels can be gathered by **allocsets**, **freesets** and **freesets_level**. To create a new leaf, we have to pick up a new *ID* to this new leaf by the strategy of *SOME p.* $p \notin$ *idset*. Later, we will prove that with this strategy, all leaves have different *IDs*.

Based on the above quad-tree structure, next we specify two operations with the buddy algorithms: **alloc** and **free**. For *rsize* (the size of requested memory block) in allocation operation, there is a definition that maps it to the level of the quad-tree and gives the most suitable level *rlv*. The concept of *most suitable level* will be proved in the next subsection. For the specification, we only use level for both the memory blocks of the quad-tree and requested memory block. And the smaller the level, the larger size the memory block.

Definition 1. *exists_freelevel*
exists_freelevel blo_set lv $\equiv \exists l.\ l \leq lv \wedge \exists b \in blo\_set.\ freesets\_level\ b\ l \neq \emptyset$

Definition 2. *freesets_maxlevel*
freesets_maxlevel blo_set lv $\equiv$
$\quad THE\ lmax.\ lmax \leq lv \wedge$
$\qquad \exists b \in blo\_set.\ freesets\_level\ b\ lmax \neq \emptyset \wedge$
$\qquad \forall l \leq lv.\ \exists b \in blo\_set.\ freesets\_level\ b\ l \neq \emptyset \longrightarrow l \leq lmax$

Definition 3. *Allocation Operation*
alloc blo_set rlv $\equiv$
$\quad$ *if exists_freelevel blo_set rlv then*
$\quad\quad$ *lmax = freesets_maxlevel blo_set rlv*
$\quad\quad$ *if lmax = rlv then*
$\quad\quad\quad$ *btree = SOME b.* $b \in blo\_set \wedge freesets\_level\ b\ rlv \neq \emptyset$

l = SOME l. l ∈ freesets_level btree rlv
else
btree = SOME b. b ∈ blo_set ∧ freesets_level b lmax ≠ ∅
l = split (SOME l. l ∈ freesets_level btree lmax) (rlv - lmax)
else False

The first step of allocation is to check whether there is a quad-tree in *blo_set* (the memory pool tree collection), that has such free memory blocks whose level is less than or equal to *rlv*. This step is done by **exists_freelevel**, and if it returns *False* then the allocation progress stops. Otherwise, the next step conducted by **freesets_maxlevel** is to return the maximum level among all levels with free memory blocks. If the maximum level is equal to *rlv*, then any free memory block in *rlv* is to be allocated. If not, any free memory block in the maximum level is to be conducted by **split** showed in Fig. 1 until free memory block in *rlv* appears and then be allocated. The state of the assigned leaf is set to *ALLOC*.

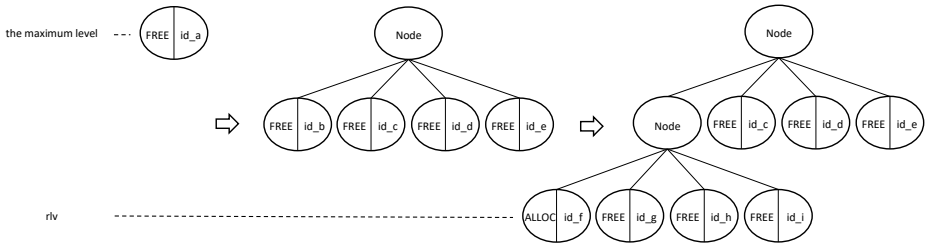

Fig. 1. The progress of dividing a free memory block

Definition 4. *Deallocation Operation*
free blo_set b ≡
if ∃btree ∈ blo_set. b ∈ tree.set btree then
if State b = FREE then False
else btree = THE t. t ∈ blo_set ∧ b ∈ tree.set t
merge (reset btree b FREE)
else False

The deallocation progress firstly checks whether there is a quad-tree in *blo_set* that the occupied memory block to be released belongs to this tree. If there is no such tree, the procedure returns *False*. Next, if the state of the occupied memory block is *FREE*, the progress also returns *False*. When all conditions are met, the memory block is returned to the tree it belongs to, thereafter merging operation is executed. The merging operation is to combine all free memory blocks that belong to the same parent tree showed in Fig. 2.

At this point, we have done the specification for the buddy memory algorithm. Next, we are going to verify some properties to guarantee the functional correctness of this specification.

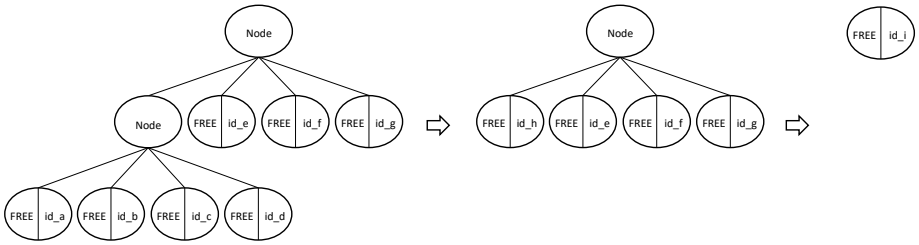

Fig. 2. The progress of merging all free memory blocks

3.2 Memory Model Properties

Once the specification is finished, the preconditions & postconditions for functions as well as the invariants are to be raised up to ensure the functional correctness of the specification. We try to answer these questions: Does the algorithm pick out the most suitable block from all the available blocks? Does the algorithm correctly adjust the state of the relevant memory block after allocation and deallocation? How does the algorithm make sure to execute the merging operation after deallocation? How does the specification ensure the correctness of the quad-tree structure? What invariants does the specification preserve? Does the algorithm satisfy some security properties? Answering these questions contributes to the construction of a reliability system. We give preconditions and postconditions in the first place.

Definition 5. *Allocation Failure*
$\neg$ *exists_freelevel blo_set rlv* $\longrightarrow$ *fst (alloc blo_set rlv) = blo_set*

Definition 6. *Freesets for Direct Allocation*
exists_freelevel blo_set rlv $\wedge$ *freesets_maxlevel blo_set rlv = rlv* $\longrightarrow$
$\exists l.\ l \in$ *freesets blo_set* $\wedge\ l \notin$ *freesets fst (alloc blo_set rlv)* $\wedge$
freesets blo_set = freesets fst (alloc blo_set rlv) $\cup\ \{l\}$

Definition 7. *Allocsets for Direct Allocation*
exists_freelevel blo_set rlv $\wedge$ *freesets_maxlevel blo_set rlv = rlv* $\longrightarrow$
$\exists l.\ l \notin$ *allocsets blo_set* $\wedge\ l \in$ *allocsets fst (alloc blo_set rlv)* $\wedge$
allocsets fst (alloc blo_set rlv) = allocsets blo_set $\cup\ \{l\}$

Definition 8. *Allocsets for Indirect Allocation*
exists_freelevel blo_set rlv $\wedge$ *freesets_maxlevel blo_set rlv* $\neq$ *rlv* $\longrightarrow$
$\exists l.\ l \notin$ *allocsets blo_set* $\wedge\ l \in$ *allocsets fst (alloc blo_set rlv)* $\wedge$
allocsets fst (alloc blo_set rlv) = allocsets blo_set $\cup\ \{l\}$

Definition 9. *Deallocation failure 1*
$\nexists$*btree* $\in$ *blo_set.* $b \in$ *tree.set btree* $\longrightarrow$ *fst (free blo_set b) = blo_set*

Definition 10. *Deallocation failure 2*
$\exists$*btree* $\in$ *blo_set.* $b \in$ *tree.set btree* $\wedge$ *State b = FREE* $\longrightarrow$
fst (free blo_set b) = blo_set

Definition 11. *Allocsets for Deallocation Success*
$\exists btree \in blo\_set.\ b \in tree.set\ btree \wedge State\ b \neq FREE \longrightarrow$
$allocsets\ blo\_set = allocsets\ fst\ (free\ blo\_set\ b) \cup \{b\}$

Theorem 1. *The buddy allocation specification satisfies all the preconditions and postconditions above.*

Proof. Apply preconditions into the definitions of allocation and deallocation, then the certain result of executions can be calculated according to the branches in the functions. The variables that change can be traced through state changes.

To guarantee that the algorithm pocks out the most suitable block, two properties have to be proved: 1. the correctness of the mapping from the request memory block size to the level of the quad-tree. 2. the correctness of the quad-tree structure. Next are these two properties.

Definition 12. *Mapping Request Sizes & Allocation Levels*
$output\_level\ blo\_list\ rsize \equiv THE\ l.\ l < L\ blo\_list \wedge rsize \leq blo\_list\ !\ l \wedge$
$(L\ blo\_list > 1 \wedge l < L\ blo\_list - 1) \longrightarrow rsize > L\ blo\_list\ !\ (l+1)$

Static linked list *blo_list* is used to store the size of block for each level in a quad-tree. And *blo_list* is a strictly decreasing list. Function L is used to get the length of a list. As mentioned above, the smaller the level, the larger size the memory block. The most suitable block means that its size has to be greater than or equal to the size of request block, and there is no smaller block that meets this condition. Through this definition, the most suitable block is picked up from *blo_list*, and then mapped to the correct level of the quad-tree. Some lemmas ensure the correctness of this definition.

Lemma 1. $L\ blo\_list > 0 \wedge rsize \leq blo\_list\ !\ (L\ blo\_list - 1) \longrightarrow output\_level\ blo\_list\ rsize = L\ blo\_list - 1$

Lemma 2. $L\ blo\_list > 1 \wedge l < L\ blo\_list - 1 \wedge rsize \leq blo\_list\ !\ l \wedge rsize > blo\_list\ !\ (l + 1) \longrightarrow output\_level\ blo\_list\ rsize = l$

Proof. To prove these two lemmas, the existence of such a level should be prove firstly by the predicates of rsize. Then with the assistance of the strictly decreasing blo_list, the exact level can be located.

Next, we introduce a lemma alone to prove the correctness of a quad-tree structure from the aspect of its level. The below **root** checks whether the tree is a root tree and **child** gives us all immediate child nodes.

Lemma 3. $(root\ btree \longrightarrow get\_level\ btree = 0) \wedge (get\_level\ btree = l \wedge l \geq 0 \wedge$
$\forall chtree \in child\ btree \longrightarrow get\_level\ chtree = l + 1)$

Proof. Induction can prove this because of the recursive structure of the quad-tree.

With the correctness of the mapping operation and the structure of a quad-tree, we have the following theorem.

Theorem 2. *The buddy allocation specification picks out the most suitable memory block and operates on the correct level in the quad-tree.*

To avoid the fragmentation, merging operation is asked in the progress of deallocation. The following property is put forward to ensure the merging operation being correctly executed. The below **leaf** is to check whether the tree is a leaf.

Definition 13. *Four Free Leaves Belong to The Same Node*
is_FFL btree $\equiv \forall$*chtree* $\in$ *child btree. leaf chtree* $\wedge$ *State chtree = FREE*

Fig. 2 can explain this definition well. Just like the subtree in the lower left corner of the first picture, four child nodes are all leaves and their states are *FREE*. Therefore, merging operation is necessary during the progress of deallocation to handle this situation. The following are the lemmas that ensure the non-existence of such *FFL* trees after allocation and deallocation operations if non-existence of such trees in preconditions.

Lemma 4. $\forall b \in$ *blo_set.* $\neg$ *is_FFL b* $\longrightarrow \forall b \in$ *fst (alloc blo_set rlv).* $\neg$ *is_FFL b*

Lemma 5. $\forall b \in$ *blo_set.* $\neg$ *is_FFL b* $\longrightarrow \forall b \in$ *fst (free blo_set b).* $\neg$ *is_FFL b*

Proof. Due to the operations being conducted on the recursive structure of the quad-tree, induction can solve the proofs with the assistants of the execution branches.

Theorem 3. *The buddy allocation specification guarantees non-existence of the FFL trees among all quad-trees.*

In the end, we prove two significant properties: memory isolation and non-leakage. The first one is to prove non-existence of the overlap in the address spaces. Isolation in address spaces makes sure domains' memory blocks are not maliciously overwritten. Memory leakage means that available memory blocks (including occupied and free blocks) are getting less and less. Then non-leakage protects the address spaces.

For the specification level, we provisionally use *ID* to present a contiguous addresses for a memory block. The mapping between a *ID* and a true range of address and the one-to-one uniqueness guarantee will be introduced in the future work. Therefore, in this specification isolation in address spaces means that all *ID*s that leaves bring are different. The strategy of creating a new leaf has been already introduced in the previous subsection.

Definition 14. *Different IDs*
is_different blo_set $\equiv \forall b \in$ *blo_set.* $\forall l \in$ *tree.set b.* $(\nexists l'.\ l' \in$ *tree.set (SOME b. b* $\in$ *blo_set)* $\wedge\ l' \neq l \wedge$ *ID l' = ID l)*

Lemma 6. *is_different blo_set ⟶ is_different fst (alloc blo_set rlv)*

Lemma 7. *is_different blo_set ⟶ is_different fst (free blo_set b)*

Proof. Induction proving can solve these lemmas with the assistants of the execution branches, because the operations are conducted on the recursive structure of the quad-tree.

Theorem 4. *The buddy allocation specification ensures all IDs of leaves are different.*

For the memory non-leakage, because we use the quad-tree structure and map all the memory blocks into the leaves of the tree, it becomes easy to prove the property. Firstly, we have this understanding to a quad-tree: all the leaves (including occupied and free leaves) are in use. Thus once the certain relationship between the number of nodes and leaves has been proved, we can infer that the quad-tree always maintains correct structure. Hence, we can prove that all the leaves are in use and none leaf is forgotten.

The first step is to prove the relationship between the number of nodes and leaves in a quad-tree.

Lemma 8. *q-tree b: Num (Leaf b) = Num (Node b) × 3 + 1*

Proof. We use induction to prove this lemma because of the recursive structure of the quad-tree.

Lemma 9. *∀b ∈ blo_set. q-tree b ⟶ ∀b ∈ fst (alloc blo_set rlv). q-tree b*

Lemma 10. *∀b ∈ blo_set. q-tree b ⟶ ∀b ∈ fst (free blo_set b). q-tree b*

Proof. Due to the operations being conducted on the recursive structure of the quad-tree, induction can solve the proofs with the assistants of the execution branches.

Theorem 5. *The buddy allocation specification guarantees any tree is a q-tree.*

To sum up, in this section we introduce the quad-tree structure to simulate memory because of the buddy allocation algorithms. Then we give a specification for the algorithms including **alloc** and **free** operations. After that, we give proofs for functional correctness including preconditions and postconditions, the most suitable memory block, non-existence of the *FFL* trees, memory isolation and non-leakage. Through these efforts, we give a functionally correct buddy memory model.

4 Model Memory Security

The concept of noninterference is introduced in [2] to provide a formalism for the specification fo security policies. The main idea is that domain u is non-interfering with domain v if no action performed by u can influence the subsequence outputs seen by v.

4.1 Memory Execution Model

Considering the fact that the execution of the buddy memory allocation is expressed as a series of operating actions among initialization, allocation, release, scheduler and time-ticking in a serial system, we introduce an event framework to simulate the whole processing of the entire system. Another factor of introducing this framework is that we can verify whether the states before and after an execution both satisfy a certain property of correctness or security.

As we define, *State* is the basic element of the execution trace. The execution of each operation must follow a certain order, therefore the trace is defined as a *list* of type *State*. A trace is defined looks like this:

type_synonym *Trace* = "*State list*"

Different orders of execution of operations compose different execution pathways. For modeling all the possible pathways, we introduces a collection for all execution pathways in a inductive way.

inductive_set *execution* :: "*Trace set*" *where*
zero_exe: "[*s*] ∈ *execution*" |
init_exe: "[*es* ∈ *execution*; *fst* (*k_mem_pool_define s nam nlv num*) = *t*; *t* = *es* ! *0*] ⟹ *s* # *es* ∈ *execution*" |
alloc_exe: "[*es* ∈ *execution*; *po* ∈ *set* (*pools s*); *fst* (*k_mem_pool_alloc s po lv ti*) = *t*; *t* = *es* ! *0*] ⟹ *s* # *es* ∈ *execution*" |
free_exe: "[*es* ∈ *execution*; *po* ∈ *set* (*pools s*); *fst* (*k_mem_pool_free s po num*) = *t*; *t* = *es* ! *0*] ⟹ *s* # *es* ∈ *execution*" |
tick_exe: "[*es* ∈ *execution*; *time_tick s* = *t*; *t* = *es* ! *0*] ⟹ *s* # *es* ∈ *execution*" |
schedule_exe: "[*es* ∈ *execution*; *schedule s* = *t*; *t* = *es* ! *0*] ⟹ *s* # *es* ∈ *execution*"

Firstly, the trace list which has only one state belongs to the execution pathways. Secondly, five operations on trace list have same pattern. Taking initialization function as an example. The *es* is one execution pathway of type *list* and initialization function translates the previous state *s* into state *t*. If the state *t* is the first element of list *es*, then we can add state *s* to the head of execution pathway *ex* to form a new list (*s* # *es*). Eventually, the new list (*s* # *es*) is also one execution pathway.

Thereafter, we have to define a reachable set using the execution trace. The purpose of introducing reachable set whose execution trace starts from *s0* is to prove all properties are established on reachable set. Following is the definition of reachable set.

definition "*reachable s t* ≡ (∃*s* ∈ *execution*. *ts* ! *0* = *s* ∧ *last ts* = *t*)"
definition "*reachable0 s* ≡ *reachable s0 s*"
definition "*ReachStates* ≡ {*s*. *reachable0 s*}"

But how to connect the allocation operation and releasing operation? The answer is the scheduling. According to the C-code level in Zepyhr OS, two places need the participation of scheduling: (1).Once the memory allocation process fails and one of the thread's parameters indicates that it will wait, we need to add the thread to the corresponding memory pool's waiting queue and then schedule another thread to run. (2).Each time after the releasing operation, the algorithm checks if there are any threads in the waiting queue of the memory pool. If there are, it wakes up all threads and scheduler arranges for one thread to execute. Although scheduling operation is not the primary role in memory algorithm verification, for algorithmic integrity, we introduce an abstract scheduling which takes any thread from the ready queue to our buddy memory model.

4.2 Security Proofs

5 Conclusions and Future Work

To sum up, we give a specification for buddy memory allocation algorithm.

References

1. T. Nipkow, L. C. Paulson, and M. Wenzel. Isabelle/HOL-A Proof Assistant for Higher-Order Logical, volume 2283 of LNCS. Springer-Verlag, 2002.
2. J. Goguen and J. Meseguer. Security Policies and Security Models, Proceedings of the IEEE Computer Society Symposium on Research in Security and Privacy, 1982, p. 11-20.