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 fundamental 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 $\equiv$
THE lmax. $lmax \leq lv \wedge$
$\exists b \in blo\_set.\ freesets\_level\ b\ lmax \neq \emptyset \wedge$
$\forall l \leq lv.\ \exists b \in blo\_set.\ freesets\_level\ b\ l \neq \emptyset \longrightarrow l \leq lmax$

Definition 3. *Allocation Operation*
alloc blo_set rlv $\equiv$
if exists_freelevel blo_set rlv then
$lmax =$ *freesets_maxlevel blo_set rlv*
if $lmax = rlv$ *then*
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*.

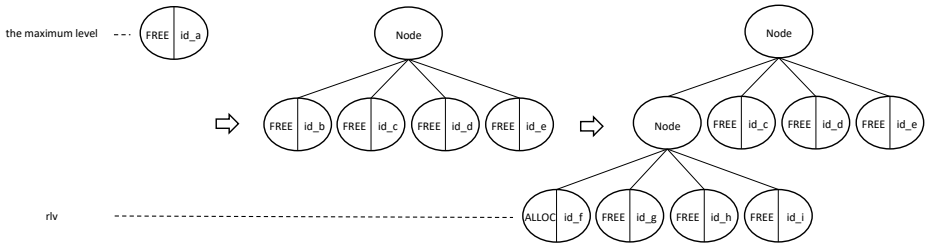

Fig. 1. The progress of dividing a free memory block

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.

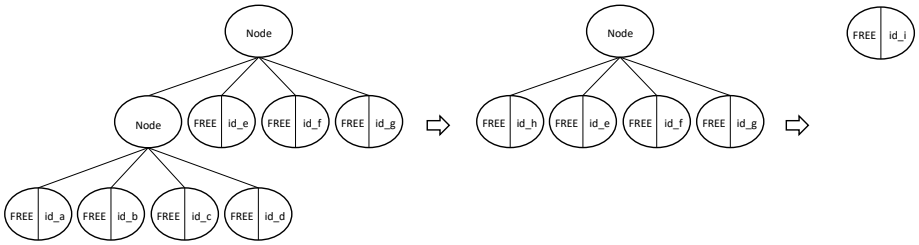

Fig. 2. The progress of merging all free memory blocks

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: Dose 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 the algorithm make sure to execute the merging operation after deallocation? How the specification ensure the correctness of the quad-tree structure? What invariants does the specification preserve? Dose 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. *Preconditions & Postconditions*
When call allocation, if ***exists_freelevel*** *returns false, then allocation fails;*
When call allocation, if ***exists_freelevel*** *returns true and* ***freesets_maxlevel*** *returns exactly the level to be allocated, then directly allocate;*
When call allocation, if ***exists_freelevel*** *returns true and* ***freesets_maxlevel*** *returns not the level to be allocated, then allocate through splitting;*
When call releasing, if the memory block to be freed dose not belong to this memory pool, then releasing fails;
When call releasing, if the memory block to be free belongs to this memory pool, then free the block and merge;
When allocation fails, if the thread will wait, then suspend this thread, insert it to the pool's waiting queue and schedule;
When releasing succeeds, if the pool's waiting queue is not empty, then wake all suspended threads.

Theorem 1. *The buddy memory model satisfies the preconditions and postconditions.*

These conditions are very simple because of the functional construction of the model. They are also very easy to understand and do not require additional explanation. These conditions only ensure that the buddy model follows the

design idea of the algorithm on the execution branches. However, they do not involve the design details inside the model. In other words, we need more invariants to ensure that the internal details of the model follow the design idea of the algorithm.

To make sure that we execute **alloc** and **free** on the right level of quad tree, we have to specify that the level of root is zero and the level of below leaf is one more level than the upper leaf. Thus, we introduce the following lemma.

Lemma 1. *The level of root is zero and the level of below leaf is one more level than the upper leaf.*

Then the question is that whether we choose a most suitable memory block for allocation. As we mentioned earlier, there is a mapping between the quad-tree hierarchy and the size of memory block. If we use static linked list *blo_list* to store block size at each level, we have the following mapping definition:

Definition 6. *Mapping Sizes & Levels*
output_level = THE l. l < length blo_list ∧ request_size ≤ blo_list ! l ∧ (length blo_list > 1 ∧ l < length blo_list - 1) ⟶ request_size > length blo_list ! (l+1)

We use *blo_list* of listing structure to record the mapping relation between request block size and quad tree's level. Due to the listing structure, quad tree's level can be expressed by the index of this list. Definition 2 gives us a function that how to get a suitable layer to allocate with the *blo_list*. A suitable layer means that the block size it represents has to be greater than or equal to the request block size, and there is no layer below it that meets the size condition. We indicate that any request block size which is greater than the first layer's size is not allowed because we handle this condition as an exception.

Theorem 2. *Any request block whose size is smaller than or equal to the first layer's size can be mapped to such a suitable quad tree level.*

Now, with lemma 1 and theorem 2, we can make sure that any operation on block size can be executed correctly on the right level of quad tree.

Another question in the memory allocation algorithm is the fragmentation. Many reasons are responsible to this phenomenon. In this case, definition 2 which maps block size to relevant quad tree level and the *blo_list* eliminate internal fragmentation. We believe that external fragmentation in buddy memory allocation algorithm is caused by the fact that all free leaves from the same level and belong to the same root tree are not merged into the upper leaf after the release operation. According to the algorithm, this situation should not be allowed. Thus, we design the following theorem to ensure the external fragmentation in-existence.

Theorem 3. *There is not existence of such circumstance that all leaves from the same level and belong to the same root tree are FREE.*

This theorem is relative to a sub-operation in **free**, merge operation. Merge operation is important to handling the external fragmentation. The core service

in the merge operation is to check whether the memory block to be released with all other leaves forms the situation that all leaves from the same level and belong to the same root are *FREE*. If this happens, then merge operation has to combine these leaves into an upper leaf. The merge operation is a inductive progress until there is not existence of such nodes and leaves. Theorem 3 guarantees that merge operation has been done at every time and got the correct result.

According to the facts that: (1)the status of a running thread is *RUNNING*; (2)the status of any thread in waiting queue is *BLOCKED*. This theorem guarantees the correctness of a thread's status before and after each operation. The theorem is as follows.

Theorem 4. *(current_thread* $\neq$ *nil* $\longrightarrow$ *Status (current_thread) = RUNNING)* $\wedge$ *(waiting_queue* $\neq \{\} \wedge t \in$ *waiting_queue* $\longrightarrow$ *Status (t) = BLOCKED)*

Above the *alloc*, *free* and *schedule* operations layer, we add the concept of thread to simulate the real situation of the whole memory management system. No matter which operation a thread invokes and what the results it gets, the only thing we need to make sure is that the thread state is correct. Thus theorem 4 guarantees this very well.

In the end, we turn to two significant properties: isolation and non-leakage. Memory isolation means that any two blocks of memory will not overlap at the address. In the event of an overlap in the address, incorrect information will directly cause confusion and errors in the program, as well as serious consequences. Memory leakage means that available memory blocks (including allocated memory blocks and unallocated memory blocks) are getting less and less. In the case of program security, some malicious programs modify the size of available memory to achieve the purpose of attacking.

Before revealing this two properties, we introduce a concept of *ID* to represent the address range of a memory block and a lemma to maintain the relation between *ID* and address range.

Definition 7. *Mapping IDs & Addresses*
The root tree addresses from root_address_start to root_address_end, the leaf α *with ID is one of child leaves of root tree whose index is leaf_index.*
Then there is a mapping that ID represents the following range of address:
$\alpha\_address\_start = root\_address\_start + leaf\_index \times \frac{Size(root\_tree)}{n}$
$\alpha\_address\_end = \alpha\_address\_start + \frac{Size(root\_tree)}{n}$

With this definition, we have to guarantee the one-to-one mapping relation between the *ID* and the range of addresses. Then we have the following lemma to ensure this.

Lemma 2. *For any range of addresses in the memory pools, there is uniquely one ID of leaf in some Block tree corresponding to it.*

Then we adopted a series of strategies to produce such an *ID*. There is no overlap in memory addresses, which translates to all *ID*s being different.

Theorem 5. *If Block tree α is such a tree whose all IDs of leaves are different, then after any operations, the new tree maintains this property.*

With lemma 2 and theorem 5, we can say that all ranges of memory addresses are isolated and there is no overlap between them.

Due to we use the structure of tree and we have mapped all the memory blocks into the leaves in these tree, it becomes easy to prove the non-leakage of memory. In a tree, we have this understanding: all the leaves including allocated and unallocated leaves are all in use. Thus once we prove the number relationship between the nodes and the leaves of the N-tree, we can say that all the leaves are in use and none leaf is forgotten.

Lemma 3. *N-tree: Num (Leaf) = Num (Node) $\times$ n + 1*

Theorem 6. *If Block tree β is such a tree whose number of leaf and node has above relation, then after any operations, the new tree maintains this property.*

With lemma 3 and theorem 6, we can ensure that all memory blocks are in use and none of them is forgotten or stolen.

To sum up, in this section we introduce the N-tree structure to simulate memory because of the buddy operation methods. Then we give a specification of the algorithm including **alloc**, **free** and **schedule**. After that, we give axiomatic proof for functional correctness including preconditions and postconditions, the most suitable allocated size, invariants during execution, memory isolation and non-leakage. Through these specifications and proof, we guarantee to give a functionally correct buddy memory model, which is significant to the next security 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 noninterfering 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.