Buddy Memory Model[*]

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

[1] SCSE, Nanyang Technological University, Singapore
`johnjiang,sanan,yangliu@ntu.edu.sg`
[2] SCSE, Beihang University, Beijing, China
`zhaoyw@buaa.edu.cn`

Abstract. It is reasonable that, from the perspective of memory blocks or units, the formalization generally includes two aspects: a set of memory operations, and the structure of memory blocks or units. However, formalization of a concrete memory model with detailed algorithm is much more different from the general method which requires as abstract as possible to be independent of the implementation details of any particular memory model. Concrete memory algorithm means that we have to explore every details of the design, which brings great complexity to the verification work, but of great significance for the practical application. For this aim, we represent an abstract specification for a concrete memory model from Zephyr OS with buddy memory algorithm, in forms of axioms that guarantee the correctness of memory structures and memory behaviors, as well as meeting the requirements of security. Thereafter, we analyze the complexity of this algorithm to provide improvement suggestions for industrial products.

Keywords: Zephyr OS · Buddy Memory Model · Axiom Specification.

1 Introduction

Many specifications treat memory as an assignment operation, thus they usually offer a set of locations and a set of values. Their memory behaviors may include some very abstract operations like allocation, free, writes and reads. However when reasoning about low-level codes in a specific operating system, our focus changes because of the concrete memory management algorithm. For example, when we invoke **k_mem_pool_alloc()** in Zephyr OS, it runs a complicated buddy memory algorithm. If we continue to abstract *k_mem_pool_alloc()* into a simple assignment operation, it will be out of date for verifying the memory management system in Zephyr OS.

Therefore the issue we care about is how, then, will it affect the correctness and security of memory operating system when we introduce complex memory algorithm above the abstract memory operations, which means we have to expand these abstract memory operations. Usually our verification to a memory model is nothing more than these properties: it should be capable to allocate

[*] Supported by organization x.

free memory and free allocated memory, write to allocated memory and read the last value written, etc., and it should not be possible to free memory that is already free, allocate memory that is already allocated, read values that have not been written, etc. Once we introduce memory algorithm, in fact, we can partially weaken the proof of these properties like writes and reads because these operations will be handled by the threads who requests memory services. Next, for the memory algorithm itself and memory blocks structure, we have to try to answer these questions: Can the memory algorithm pick out a appropriate block from the memory pools by searching or other auxiliary operations, such as cracking large blocks? Can the algorithm return the released block to the memory pools and adjust the free memory blocks, for example, to merge for expecting larger block requirements later? Dose the memory management algorithm correctly schedule the threads requesting the memory services, including blocking threads that do not get memory blocks and waking up threads in the wait queue. Can the memory pool be managed correctly at all times, such as free blocks and allocated blocks are distinguished, and memory blocks are unique? Dose the algorithm satisfy some security properties, such as memory services non-interference? These questions constitute a challenge to validating a memory management algorithm or system. Our work is to answer the questions above.

Our work is related to the actual requirements and applications of Zephyr OS in industry. Zephyr OS is a real-time operating system that supports the Internet of Things under the Linux Foundation and the demand for our work is exactly what this OS used in the communication system of high-speed trains. Among all the subsystems in Zephyr OS, the memory management system with concrete algorithm is the first module we must ensure correctness and security. Different from Linux, Zephyr OS adopts quadruple to allocate and reclaim memory blocks for efficiency reasons. This memory algorithm supports multi-threads shared accessing and interrupt locks. Due to the quadruple, the structure of memory block with multi-level bitmaps and multi-level doubly linked lists becomes quite complex. Because of this, our verification to this memory management system and its buddy memory algorithm will have substantial implications for the application of Zephyr OS in industry.

In this paper, we develop a specification for a sequential memory model that concludes allocation, free and scheduling operations with buddy memory algorithm. We offer a set of axioms that provide enough information to reason about the correctness of functional algorithm under a event-based framework. We also provide an interface for upper programs for verifying the noninterference security property. All definitions and proofs have been formalized in Isabelle/HOL proof assistant.

2 Isabelle/HOL Verification Environment

The verification environment which is based on the interactive theorem prover Isabella/HOL is also a key component to our work. 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 adn the **primrec** function definition is used for primitive recursions. The notation [A_1;. . . ;A_n] $\Longrightarrow$ A represents an implication with assumptions A_1;. . . ;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 Memory Model

The most striking feature of the buddy memory allocation algorithm is layering: A large memory block is divided into four, then *n*-layer and *n+1*-layer are formed according to the size of the memory block; Multi-level doubly linked lists are used to manage free memory blocks scattered at various levels. Obviously, according to the first point in the above features, the memory structure can be implemented by a quad-tree. At the same time, the state tags of the leafs in this quad-tree can simulate the free and occupied memory blocks. Once we realize the core of the buddy memory allocation algorithm, it is easy to understand what the algorithm says.

When a reasonable memory block request comes in, the algorithm will look for a free memory block that contains the most suitable size on the corresponding free doubly linked list. If the appropriate memory block can be found directly, it is then allocated to the requesting thread. The splitting operation to a larger free memory block makes the algorithm become complicated when none of suitable memory block exists. The complex process is like this: a larger memory block is taken from the upper layer free list and cracked until the requested memory block is produced, meanwhile, free blocks not used in this cracking process are added to the corresponding hierarchy of free lists, in the end, the requested memory block can be allocated to the thread. The algorithm has to make sure two *suitable*: the free memory block to be allocated must be bigger than the requested memory block, but the smallest one among all suitable memory blocks; the free memory block to be cracked also must be the smallest among all suitable memory blocks. These two points further reduce the in-block and out-block fragments.

The releasing process on allocated memory block could be easy except the additional merge operation. The first step is that the allocated memory block will be returned to corresponding hierarchy of free list. Then if there are four free sub-blocks and they all belong to a larger memory block, these free memory blocks will be merged into a larger memory block. The process is repeated until the entire block of memory can no longer be merged. This progress is called merge operation. The merge operation is reasonable and necessary for the considering of reducing debris and providing adequate resources.

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.

Base on the cognition to the buddy memory algorithm above, we design the fundamental structure of a quad-tree: *'a tree* = *Leaf* (*L: 'a*) | *Node* (*LL: 'a tree*) (*LR: 'a tree*) (*RL: 'a tree*) (*RR: 'a tree*), with two pieces of information carried by the leaf: memory blocks' state (indicated as *ALLOC* and *FREE* respectively); a contiguous address occupied by a memory block (indicated as *ID*). We begin with three operations with certain algorithms mentioned above: **alloc**, **free**, **schedule**. In addition, we introduce some auxiliary variables and definitions to our model, such as abstract *Thread* that contains the running states; simplified collection of resources *State* that simulates every running state of the whole system; sequential linked list *execution* that represents the executive processes and etc,.

Our goal is to offer a simple specification of buddy memory algorithm such that:

1. The specification must be consistent with the implementation detail of the buddy memory algorithm in Zephyr OS's memory management system, which means we need to use the theory proof method to prove that we have built a correct model.
2. We need to verify whether the specification has relevant security properties, so that the algorithm can be guaranteed in the actual application process.
3. From the perspective of algorithm, it is necessary to analyze the complexity in order to provide improvement advice for industrial products.

3.1 Memory Model Axioms

The most basic guarantee for a model which is designed according to some algorithm, is that this model meets the functional requirements of the algorithm. The common method to prove functional correctness is to use preconditions and postconditions. Before designing these conditions, we explain a few auxiliary concepts and functions: The **level** used in the quad-tree is mapped to the size of memory block. For example, root node's level is zero and its size is 1024MB, then the next level of this tree is one, and they're 256MB in size. Function **exists_freelevel** tells us if there is any memory block whose size is greater than or equal to the requested block size. Function **freesets_maxlevel** gives us some memory block which is the smallest one among all blocks whose size are greater than or equal to the requested block size. Then we give preconditions and postconditions for functional correctness like this:

Definition 1. *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 2. *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 ≠ nil ⟶ Status (current_thread) = RUNNING) ∧ (waiting_queue ≠ {} ∧ t ∈ waiting_queue ⟶ 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 3. *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 Security Model of Memory

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: "$[s] \in$ *execution*" |
init_exe: "[*es* $\in$ *execution*; *fst* (*k_mem_pool_define s nam nlv num*) = *t*; *t* = *es* ! *0*] $\Longrightarrow$ *s* # *es* $\in$ *execution*" |
alloc_exe: "[*es* $\in$ *execution*; *po* $\in$ *set* (*pools s*); *fst* (*k_mem_pool_alloc s po lv ti*) = *t*; *t* = *es* ! *0*] $\Longrightarrow$ *s* # *es* $\in$ *execution*" |
free_exe: "[*es* $\in$ *execution*; *po* $\in$ *set* (*pools s*); *fst* (*k_mem_pool_free s po num*) = *t*; *t* = *es* ! *0*] $\Longrightarrow$ *s* # *es* $\in$ *execution*" |
tick_exe: "[*es* $\in$ *execution*; *time_tick s* = *t*; *t* = *es* ! *0*] $\Longrightarrow$ *s* # *es* $\in$ *execution*"

|

schedule_exe: "[*es* $\in$ *execution*; *schedule s* = *t*; *t* = *es* ! *0*] $\Longrightarrow$ *s* # *es* $\in$ *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* $\equiv$ ($\exists$*s* $\in$ *execution*. *ts* ! *0* = *s* $\wedge$ *last ts* = *t*)"
definition "*reachable0 s* $\equiv$ *reachable s0 s*"
definition "*ReachStates* $\equiv$ {*s*. *reachable0 s*}"

4.2 Security Proofs

5 Related Work

6 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.