REVISTA BRASILEIRA DE MECATRÔNICA
FACULDADE SENAI DE TECNOLOGIA MECATRÔNICA

Desfragmentação de memória em sistemas embarcados

Memory defragmentation in embedded systems

Elielder Belchior de Melo, [1i]
Vinicius Augusto Passarella de Melo, [2 ii]
Leandro Poloni Dantas, [3 iii]
Fernando Simplicio de Sousa, [4 iv]

Data de submissão: (27/04/2022) Data de aprovação: (13/09/2022)

RESUMO
Com a crescente complexidade das aplicações embarcadas, a área de memória dinâmica é utilizada para aumentar a capacidade das aplicações em tempo de execução. No entanto, as sucessivas alocações e liberações de memória causam um efeito chamado fragmentação de memória, que são espaços não alocados e não sequenciais. Sendo assim, este artigo apresenta um novo algoritmo que objetiva retirar as lacunas de memória por conta das desalocações de memória. O processo se dá pela divisão do espaço de memória dinâmica, *heap*, alocação e desalocação de objetos e reorganização do *heap*, com a troca dos objetos entre o *heap* e seu espelho complementar. Com a implementação do algoritmo, alocações e desalocações são feitas sem que ocorram fragmentações na memória e preservando a integridade dos dados. Por outro lado, a maior demanda computacional imposta pelo algoritmo é justificada principalmente em sistemas embarcados complexos e robustos.

Palavras-chave: Fragmentação; Memória Dinâmica; *Heap***.**

ABSTRACT
Due to the increasing complexity of embedded applications, the dynamic memory area has been used to increase the capacity of applications at runtime. However, successive memory allocations and frees can create an effect called memory fragmentation, which is unallocated and nonsequential spaces. Therefore, this article presents a new algorithm that aims to remove memory gaps due to memory deallocations. This process works through the division of dynamic memory space, heap, allocation, and deallocation of objects and reorganization of the heap through the exchange of objects between the heap and its complementary mirror. With the implementation of the algorithm, allocations and deallocations are done without

[1] Pós-graduado em Sistemas Embarcados e Técnico em Eletrônica. E-mail: elielderbelchior@gmail.com
[2] Pós-graduado em Sistemas Embarcados e Técnico em Eletrônica. E-mail: viniciusaugusto_pm@hotmail.com
[3] Doutor em Engenharia Eletrônica, Professor da Faculdade SENAI no curso de Pós-graduação em Sistemas Embarcados. E-mail: leandro.poloni@sp.senai.br
[4] Mestre em Engenharia Elétrica, professor da Faculdade SENAI no curso de Pós-graduação em Sistemas Embarcados. E-mail: fernando.simplicio@sp.senai.br

memory fragmentation and preserving data integrity. On the other hand, the higher computational demand required by the algorithm is mainly justified in complex and robust embedded systems.

Keywords: Memory fragmentation; Dynamic memory; Heap.

1 INTRODUÇÃO

Devido à complexidade das aplicações em sistemas embarcados, a utilização de memória dinâmica, que se refere ao processo de alocação e liberação de memória em tempo de execução nas aplicações é uma possibilidade que amplia a capacidade de aplicações embarcadas.

Diferentemente de alocações estáticas de memória, as alocações dinâmicas possibilitam o uso de uma determinada área de memória para mais de uma aplicação (WILSON et al., 1995), fazendo com que um sistema possa ter mais possibilidades e maior complexidade em um ambiente de menor recurso de memória RAM (*random access memory*).

Juntamente à possibilidade de tornar o sistema embarcado mais complexo (ANDERSSON, 2005; JERSAK, RICHTER & ERNST, 2005), as alocações dinâmicas de memória trazem consigo uma característica que torna o sistema incapaz de operar aplicações com frequentes alocações e liberações de memória e que requerem maior quantidade de recursos, identificada como fragmentação de memória (BOHRA; GABBER, 2001).

Algoritmos e padrões criados para solucionar a fragmentação de memória não têm obtido resultados significativos (FERRES, 2010), pois, apesar de diminuir os efeitos da fragmentação de memória, eles ainda permanecem em áreas pequenas da memória. Com o objetivo de reorganizar a memória e diminuir a fragmentação (BOHRA; GABBER, 2001), foi implementado o espelhamento da região de alocação de memória dinâmica**,** sendo aplicado em conjunto com partes de algoritmos utilizados em *Garbage Collector* (BACON, CHENG, GROVE, 2004; BOEHM, 1995)*,* que é um mecanismo que fornece recuperação automática de memória para blocos de memória não utilizados.

Este artigo tem por objetivo estudar a fragmentação de memória e propor uma nova maneira de alocação dinâmica por reorganização, dividindo o *heap* em duas partes de igual tamanho para que, por espelhamento, o *heap* ativo possa ser alocado na sua parte complementar, quando houver novas alocações e desalocações de memória.

2 REVISÃO DE LITERATURA

Na alocação dinâmica de memória, a memória RAM é utilizada no *heap*, que está diretamente ligado com outro espaço de memória chamado *stack*. O *heap* e o *stack* são inversamente proporcionais um ao outro, quanto maior o uso do *heap* menor o tamanho de *stack* a ser utilizado. O *stack* é a localidade na memória em que as tratativas de chamada e retorno são feitas para que o programa seja executado (FERRES, 2010).

O *heap* é uma área de alocação dinâmica de variáveis. Se um programa utiliza uma lista encadeada por exemplo, ele aloca essa estrutura que cresce dinamicamente no *heap*. Essas alocações são feitas em tempo de execução para que a memória seja utilizada por mais de uma aplicação e não esteja fixa a ser utilizada apenas por um único programa (PUAUT, 2002).

As sucessivas alocações e liberações de memória causam a fragmentação, que impede a operação correta da aplicação pela criação de lacunas entre as alocações ativas na memória,

fazendo com que a memória dinâmica esteja desfigurada a ponto de não permitir mais alocações mesmo com espaços livres (BOHRA; GABBER, 2001). O processo que causa a fragmentação de memória é descrito a seguir.

Um espaço do *heap* inutilizado, na memória RAM, é alocado e a primeira solicitação recebe o primeiro endereço do *heap* com reserva do espaço a ser utilizado pela aplicação. Assim que a aplicação faz o uso da memória dinâmica, a memória é liberada no *heap* (WILSON et al., 1995).

Se um endereço de memória está alocado com um espaço 'x' e outra alocação é feita, ela será alocada no endereço seguinte ao limite da alocação anterior.

Se a primeira alocação é finalizada, com a liberação de memória, o primeiro espaço é deixado livre. Sendo assim, o limite máximo que poderá ser alocado para uma próxima aplicação será o tamanho alocado para a primeira aplicação (FREERTOS, 2021).

Então, há um espaço inutilizado com tamanho 'x' e no endereço seguinte a alocação para a segunda aplicação. Assim se obtém a chamada fragmentação na memória, sem a possibilidade de reorganização, do segundo objeto alocado, para o primeiro endereço.

Veja o exemplo da Figura 1, após algumas alocações e desalocações a memória ficará fragmentada.

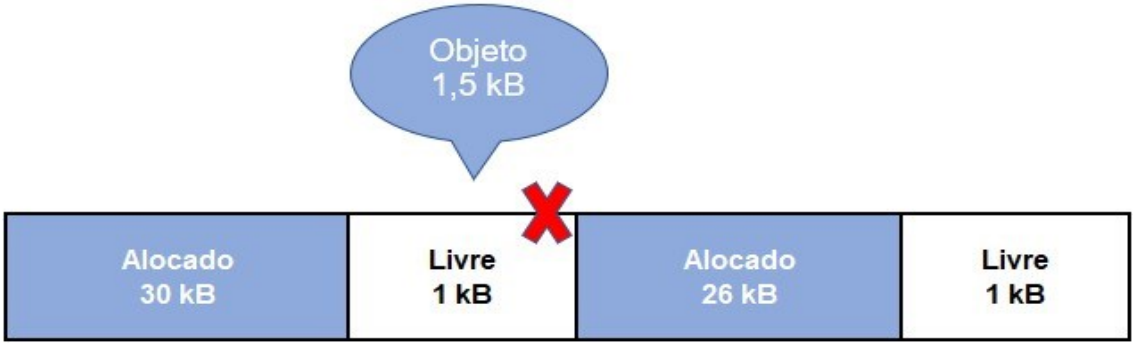
Figura 1 – Demonstração da fragmentação da memória após duas desalocações.

Fonte: Dados pelo autor.

Na prática, em caso de uma nova alocação de um objeto, por exemplo com o tamanho de 1,5 kB, embora com 2 kB disponíveis a alocação não seria possível. Esse é o principal motivo da fragmentação de memória.

A Figura 2 ilustra que apesar do total de memória livre disponível totalize 2 kB, não é possível fazer a alocação, pois como a memória está fragmentada, não existe um espaço que totalize o tamanho do objeto.

Figura 2 – Demonstração de que a alocação não é possível, pois a memória está fragmentada.



Fonte: Dados pelo autor.

Com a ocasião do processo de fragmentação, a memória necessita da reorganização, que requer um processo delicado de armazenamento de endereços.

A Figura 3 mostra como a memória ficaria após o processo de reorganização.

Figura 3 – Demonstração de reorganização do *heap*.

Fonte: Dados pelo autor.

Um dos processos mais utilizados para conhecer, sincronizar as alocações e liberações e reorganizar os objetos na área de memória dinâmica é chamado *garbage collection* (coleta de lixo) (YUASA, 1990).

A coleta de lixo (GC) é um mecanismo que fornece recuperação automática de memória para blocos de memória não utilizados. Os programas alocam memória dinamicamente, mas quando um bloco não é mais necessário, eles não precisam devolvê-lo ao sistema explicitamente com uma chamada "*free*", que realiza a liberação de memória no endereço vinculado até o tamanho vinculado na alocação (ZLATANOV, 2015). O mecanismo de GC se encarrega de reconhecer que um determinado bloco de memória alocada não é mais usado e o coloca de volta na área de memória livre (MASMANO et al., 2004; NEVES, 2015).

Pela definição de *garbage collection* supracitada, sua utilização não é a mais adequada quando nos referimos a sistemas embarcados. Em aplicações embarcadas, o desenvolvimento é feito em sua grande maioria em linguagem C, a qual está diretamente ligada à arquitetura dos microcontroladores e microprocessadores, sendo assim, controlada diretamente pelos processos escolhidos pelo desenvolvedor que precisa lidar com as alocações e os recursos de memória dinâmica utilizada pelas aplicações (BASKIYAR & MEGHANATHAN, 2005; JERSAK; RICHTER & ERNST, 2005).

Uma aplicação que aloca e desaloca objetos da memória frequentemente pode causar fragmentação, ou seja, lacunas que impossibilitam a junção dos blocos livres na memória. A proposta a seguir apresenta uma nova solução para o gerenciamento da memória dinâmica, são elas a divisão do *heap* e a realocação de memória automática com reorganização em tempo de execução.

3 METODOLOGIA

3.1 Detalhamento do algoritmo

A algoritmo foi implementado e testado em um ambiente embarcado no kit de desenvolvimento STM32F746-Disco, com recurso de 1 MB de memória Flash e 320 kB de memória RAM. O ambiente de desenvolvimento utilizado foi a IDE (*integrated development environment*) STM32CubeIDE.

Esse algoritmo tem como objetivo eliminar a fragmentação da memória por meio das seguintes etapas:

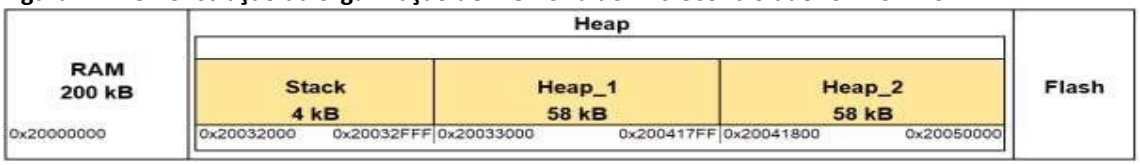
1) Dividir o tamanho total do *heap* em dois.
Criação de dois *heaps* de tamanhos iguais, para fazer um espelhamento de memória. Sempre um deles será o ativo e, o outro, o passivo.

2) Controlar todos os objetos criados pela aplicação.
Por meio de listas, fazer um gerenciamento de todos os objetos criados pela aplicação durante a execução do programa. Isso visa garantir a rastreabilidade dos objetos.

3) Quando um objeto é removido da memória, realizar o chaveamento dos *heaps*. Ao remover um objeto da memória, provavelmente é criada uma fragmentação, sendo assim, o chaveamento do *heap* é executado e todos os objetos existentes que estão ativos no *heap* são novamente alocados no *heap* passivo de forma organizada e sequencial.

Observe o exemplo a seguir: A Figura 4 ilustra a proposta de divisão da memória RAM em quatro segmentos: RAM de uso geral, *stack* e os dois *heaps*.

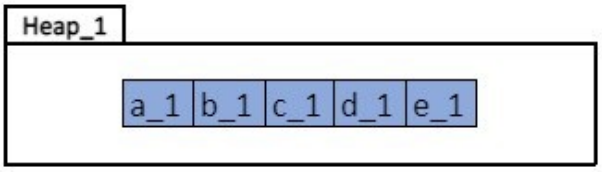
Figura 4 – Demonstração da organização de memória do microcontrolador STM32F767NI



Fonte: Dados pelo autor.

Suponha que o heap_1 seja o *heap* ativo e sua memória ocupada por cinco objetos, como exemplificado na Figura 5.

Figura 5 – Exemplificação dos objetos do *heap*_1.



Fonte: Dados pelo autor.

Objeto: a_1 tamanho: 32 bytes

Objeto: b_1 tamanho: 8 bytes

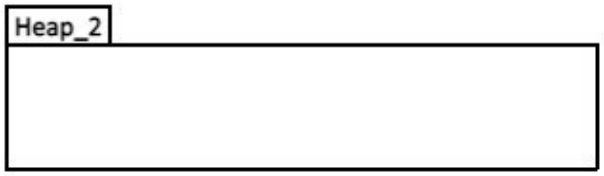
Objeto: c_1 tamanho: 128 bytes

Objeto: d_1 tamanho: 64 bytes

Objeto: e_1 tamanho: 16 bytes

Já o *heap*_2, que é o *heap* passivo, sem alocações em sua região de memória, como demonstrado na Figura 6.

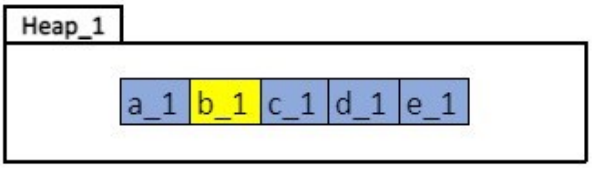
Figura 6 – *Heap*_2 totalmente livre de alocações.



Fonte: Dados pelo autor.

Caso uma desalocação de um objeto alocado no *heap*_1 se torne necessária, por exemplo, o objeto b_1, esse objeto é excluído da memória, no entanto a sua exclusão cria uma lacuna entre os objetos. Isso foi ilustrado na Figura 3 e também na Figura 7.

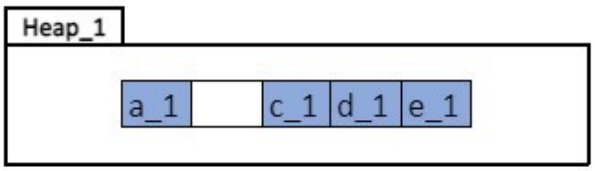
Figura 7 – Remoção do objeto [b_1] do *heap*_1.



Fonte: Dados pelo autor.

Após a exclusão do objeto [b_1] uma fragmentação é observada no *heap*_1, como demonstrado na Figura 8.

Figura 8 – Fragmentação gerada no *heap*_1 após a remoção do objeto [b_1].



Fonte: Dados pelo autor.

Consequentemente, não é possível juntar esse bloco fragmentado, com 8 bytes de tamanho, a um espaço livre no *heap*. Porém, ao realizar o *free,* que é a liberação de memória alocada à aplicação, o *heap* é chaveado, ou seja, o heap_1 passa de ativo para passivo e o heap_2 de passivo para ativo. Além disso, todos os objetos existentes no heap_1 serão alocados novamente no heap_2 de forma sequencial e organizada (Figura 5). E após isso, todos os objetos existentes no heap_1 serão excluídos.

A proposta para solucionar o problema da fragmentação é realizar a divisão do *heap* na metade, para que existam dois *heaps* de mesmo tamanho, ou seja, se o *heap* apresentar 256 bytes de memória total, o heap_1 ficará com 128 bytes e o heap_2 ficará com os outros 128 bytes.

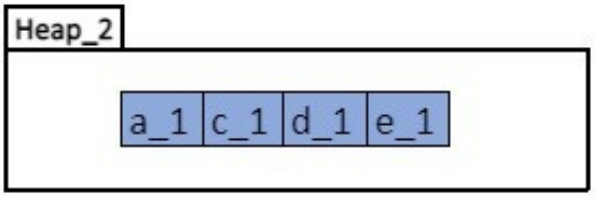
A implementação consiste em utilizar uma das metades do *heap* com uma área ativa da memória, ou seja, essa área ativa é uma das metades utilizada para que a aplicação faça alocação dos objetos. Já a outra região da memória é denominada parte passiva, que só é utilizada quando a aplicação realizar uma desalocação. Com essa divisão é possível fazer um espelho do *heap*, ou seja, quando uma desalocação acontece, todos os objetos do *heap* ativo passam para o *heap* passivo de forma organizada e ordenada.

Para isso, também é necessária a criação de uma lista de objetos, que contém todos os objetos existentes em memória. Com essa lista de objetos, é possível rastrear quais objetos estão em uso e quais foram apagados.

Ao realizar uma desalocação, o algoritmo proposto remove o objeto da lista, além disso, também organiza a lista de objetos. Assim, todos os objetos que foram ordenados na lista são novamente alocados no *heap* passivo, que agora passa ser o *heap* ativo. O chaveamento dos *heaps* permite uma reorganização dos *heaps* quando uma desalocação ocorre. Para que isso aconteça, é necessário ter um espaço para organizar os objetos quando um deles for desalocado, ou seja, isso permite que sempre o *heap* ativo fique organizado sem a fragmentação.

A Figura 9 mostra a disposição das alocações de memória após o chaveamento do heap 1 para heap 2.

Figura 9 – Após o chaveamento de *heap*, a fragmentação foi eliminada.



Fonte: Dados pelo autor.

3.2 Implementação do algoritmo

O algoritmo é constituído de conceitos avançados de programação C, tais como ponteiros e estruturas, listas encadeadas e passagem de valor por referência de barramento, baseados na arquitetura do microcontrolador STM32F746NI.

A proposta a seguir apresenta uma nova solução para o gerenciamento da memória dinâmica, são elas a divisão do *heap* e a realocação de memória automática com reorganização em tempo de execução.

Para inicializar o *heap*, uma seção de memória deve ser fornecida. Como mencionado anteriormente, as regiões de memória são definidas com base no endereço da memória RAM do controlador.

A Figura 4 ilustra a divisão da memória para atribuição dos endereços para o *heap*_1 e *heap*_2.

Quando a função *init_heap()* é chamada, o endereço da estrutura do *heap* deve ser fornecido. Essa função cria um grande espaço com cabeçalho (estrutura *node_t*) e um rodapé (estrutura *footer_t*).

A estrutura *node_t* é denominada de cabeçalho, ela contém informações sobre o detalhamento do bloco. Ela é composta da seguinte forma:

```
typedef struct node_t {    uint
        hole;    uint size;
        struct node_t* next;
        struct node_t* prev;
}node_t;
```

4 RESULTADOS E DISCUSSÕES

Para as aplicações que se utilizam de memória dinâmica para processamento de dados, as sucessivas alocações e desalocações de memória são parte da operação contínua do sistema embarcado.

Nesse cenário de teste, diversos objetos foram criados, de forma a comprovar a teoria mencionada no tópico anterior. Para isso, algumas definições são necessárias. Uma delas é a divisão da memória do controlador STM32F767NI, adotado para a realização dos testes.

Foi necessária a delimitação das regiões de memória RAM, do *heap* e da memória *flash*.

RAM (xrw) : Início = 0x20000000, Tamanho = 200 kB

HEAP (xrw) : Início = 0x20032000, Tamanho = 120 kB

FLASH (rx) : Início = 0x8000000, Tamanho = 1024 kB

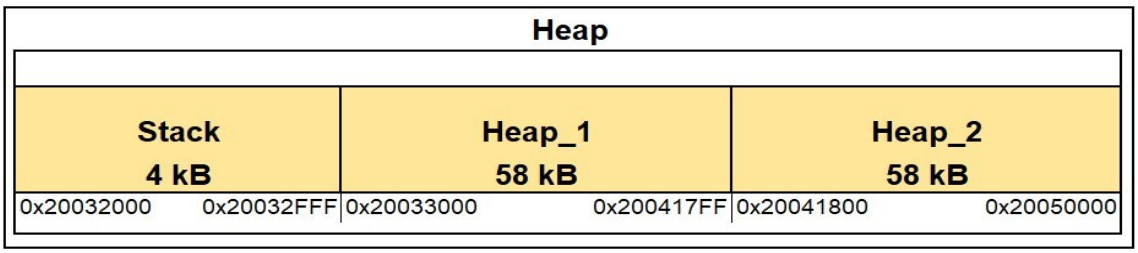
Ou seja, o espaço destinado ao *heap* começa do endereço 0x20032000 indo até o endereço 0x20050000, totalizando 120 kB. É importante ressaltar que o *heap* é parte da memória RAM, sendo parte dos 320 kB totais da memória RAM do controlador referido.

Dentro dessa região delimitada do *heap*, foi criado um *stack* com 4 kB para realizações de controle e gerenciamento de objetos, tais como listas, alocações entre outros. E o restante ficará destinado para o *heap*.

A Figura 10 mostra um detalhamento da divisão da área do *heap*, indicando todas as regiões necessárias para realizar a implementação do algoritmo estudado.

Assim sendo, a área total do *heap* é de 116 kB, que foi dividida da seguinte forma.

Figura 10 – Detalhamento da divisão da memória (*heap*).



Fonte: Dados pelo autor.

Através da Tabela 1 é possível interpretar a imagem acima de uma forma mais prática, na qual são evidenciados os endereços de memória inicial, final e tamanho da região destacada.

Tabela 1 – Indicação de cada bloco utilizado na memória, com seus respectivos endereços e tamanhos.

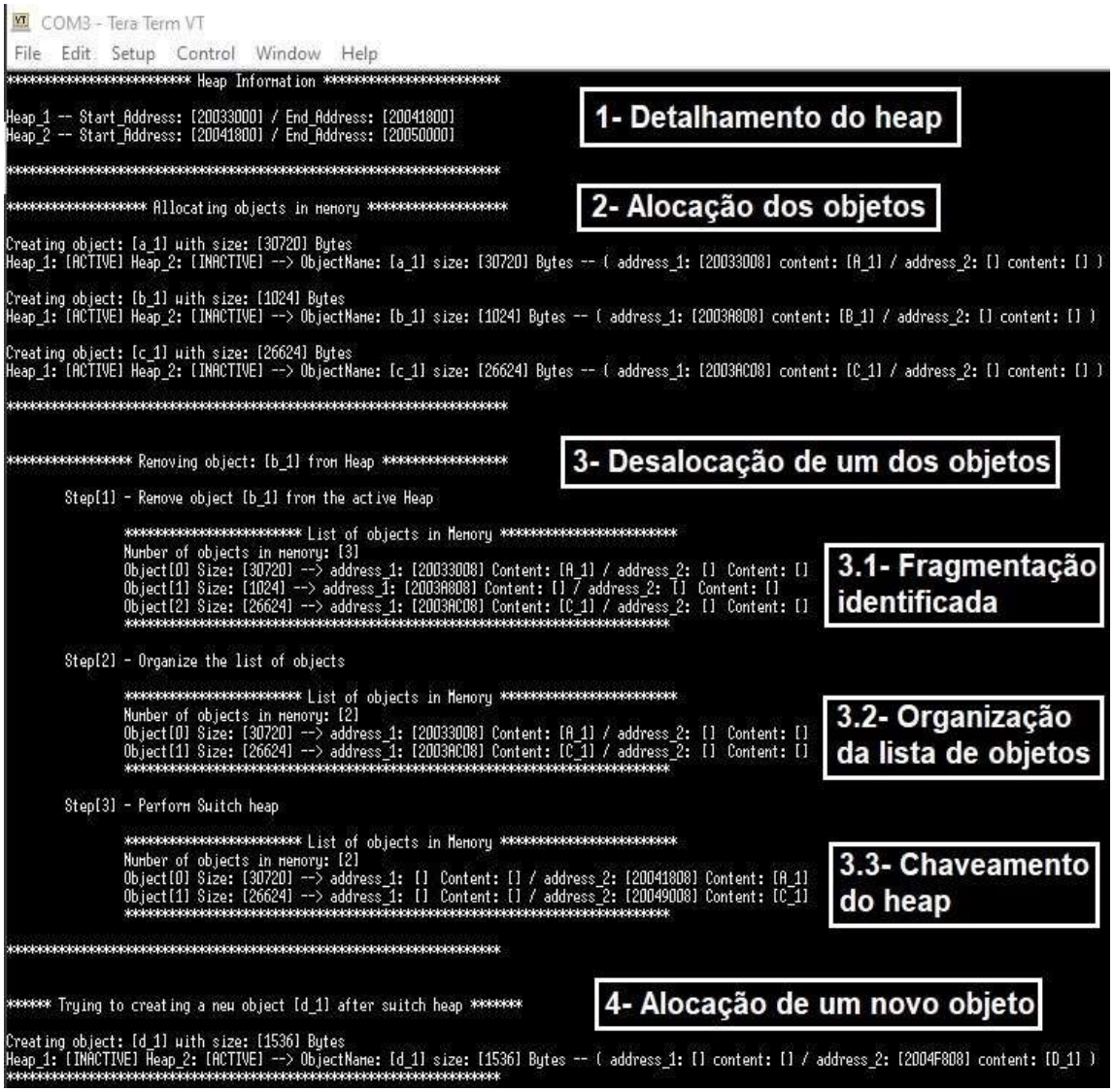
Região de memória	Endereço inicial (Hexa)	Endereço final (Hexa)	Tamanho (kB)
Stack	0x20032000	0x20032FFF	4
Heap_1	0x20033000	0x200417FF	58
Heap_2	0x20041800	0x20050000	58

Fonte: Dados pelo autor.

Para evidenciar o algoritmo proposto, foi realizado um procedimento teste que teve como objetivo gerar uma fragmentação no *heap*_1.

A Figura 11 é uma fotografia do *heap,* que mostra todas as etapas que foram executadas nesse procedimento de teste.

Figura 11 – Console do *software* Teraterm, mostrando todas as etapas realizadas durante o procedimento de teste.



Fonte: Dados pelo autor.

As 4 etapas mencionadas na Figura 11 são descritas e explicadas a seguir.

1- Detalhamento do *heap*

Antes de iniciar qualquer processo de alocação de memória, foi necessária a criação dos dois *heaps*. A Figura 12 tem como objetivo mostrar em tempo de execução os endereços iniciais e finais de ambos os *heaps*.

Figura 12 – Console do software Teraterm, mostrando o detalhamento dos endereços dos *heaps*.

Fonte: Dados pelo autor.

Além disso, com esses endereços foi possível calcular o tamanho dos *heaps*.
[Tamanho do *heap*_1] = endereço_final – endereço_inicial

[Tamanho do *heap*_1] = 0x20041800 – 0x20033000
[Tamanho do *heap*_1] = E800 (Hexa)
Ou seja, o tamanho total do *heap*_1 foi de 58 kB.
O mesmo cálculo pode ser aplicado para o *heap*_2.
[Tamanho do *heap*_2] = endereço_final – endereço_inicial
[Tamanho do *heap*_2] = 0x20050000 – 0x20041800
[Tamanho do *heap*_2] = E800 (Hexa)
Ou seja, o tamanho total do *heap*_2 foi de 58 kB.

2- Alocação dos objetos

O procedimento de teste mencionado anteriormente teve como objetivo gerar uma fragmentação no *heap*_1. Para isso, 3 alocações foram necessárias:

- Uma alocação de um objeto denominado [a_1] com tamanho de 30 kB.

- Uma outra alocação de um objeto denominado [b_1] com tamanho de 1 kB. - E finalmente uma última alocação de um objeto denominado [c_1] com tamanho de aproximadamente 26 kB.

Ao somar os tamanhos de todos os objetos, será obtido um valor de 57 kB, que é muito próximo do tamanho total do heap_1, sobrando um espaço de aproximadamente 1 kB.

A Figura 13 a seguir mostrará uma evidência das alocações descritas acima. Também mostrará que o *Heap*_1 é o *heap* ativo e, por conta disso, todos os objetos serão alocados no endereço_1 e, consequentemente, o endereço 2, relativo ao *heap*_2, estará vazio.

Figura 13 – Console do *software* Teraterm, mostrando os objetos alocados juntamente com seus respectivos endereços na memória do *heap*_1.

Fonte: Dados pelo autor.

A Tabela 2 tem como objetivo simplificar a visualização do que está sendo exibido na Figura 13, ou seja, com ela será possível interpretar os endereços iniciais, finais e o tamanho dos objetos criados na memória.

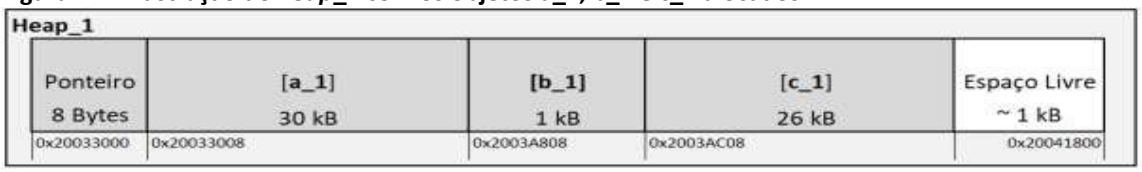
Tabela 2 – Detalhamento de cada objeto criado no *heap*_1 com seus respectivos endereços e tamanhos.

Objetos	Endereço inicial (Hexa)	Endereço final (Hexa)	Tamanho (kB)
a_1	0x20033008	0x2003A807	30
b_1	0x2003A808	0x2003AC07	1
c_1	0x2003AC08	0x20041408	26

Fonte: Dados pelo autor.

A partir disso, com a Figura 14, é possível visualizar heap_1 com os objetos alocados e distribuídos ao logo de todo o seu espaço.

Figura 14 – Ilustração do *heap*_1 com os objetos a_1, b_1 e c_1 alocados.



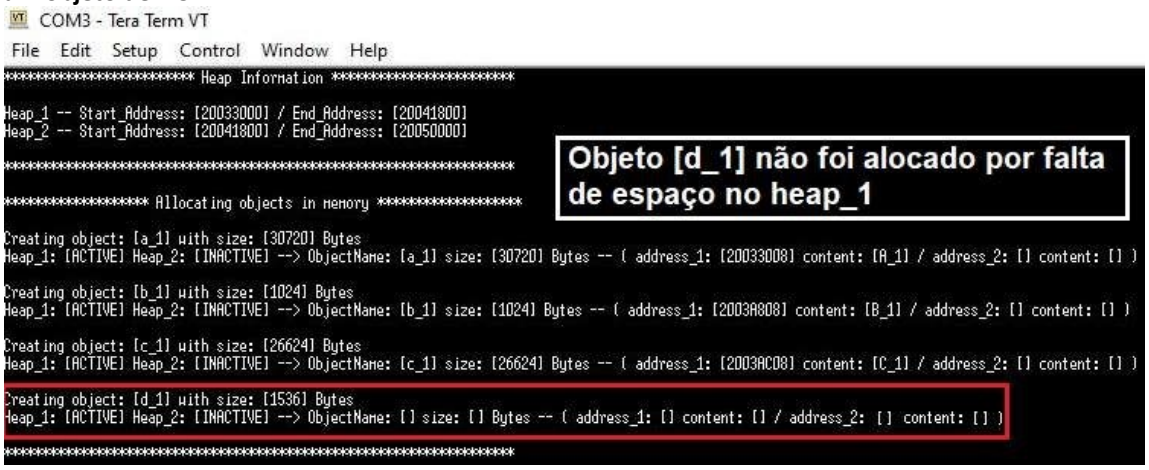
Fonte: Dados pelo autor.

Os primeiros 8 *bytes* alocados em memória são destinados ao ponteiro, pois ele guarda em si um endereço de memória, e este tem sempre o mesmo tamanho independentemente do tipo de dados com o qual se deve interpretar o objeto naquele endereço.

A partir dessas três alocações descritas acima, o *heap*_1 comportou três objetos alocados.

Um outro procedimento de teste foi realizado com o objetivo de alocar um objeto com tamanho maior que o espaço disponível no *heap*_1. Para isso, foi realizada a tentativa de alocação de um quarto objeto denominado [d_1] com tamanho de 1,5 kB. Veja a Figura 15 a seguir, a qual representa o mesmo cenário descrito anteriormente com os mesmos objetos, apenas se diferenciando pela inclusão do objeto [d_1].

Figura 15 – Console do *software* Teraterm, evidenciando a de falta de espaço no *heap*_1 ao tentar alocar um objeto de 1.5kB.



Fonte: Dados pelo autor.

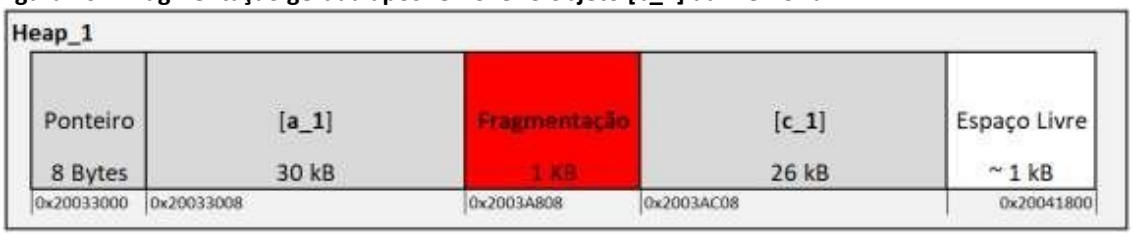
Lembrando que esse cenário de teste foi apenas uma comprovação de que não foi possível alocar um objeto maior que o espaço disponível no *heap*_1.

Nas próximas etapas, que serão detalhadas, deve-se considerar o primeiro cenário de teste, que representa os três objetos alocados.

3- Remoção de um dos objetos

Nessa etapa, o objeto [b_1] foi removido da memória. Com essa remoção, uma fragmentação foi gerada, conforme detalhado na Figura 16 a seguir.

Figura 16 – Fragmentação gerada após remover o objeto [b_1] da memória.



Fonte: Dados pelo autor.

Com o algoritmo proposto neste artigo, algumas etapas foram executadas ao remover um objeto do *heap,* para não gerar essa fragmentação na memória.

A Figura 17 mostra as 3 etapas executadas durante a remoção de qualquer objeto do *heap*.

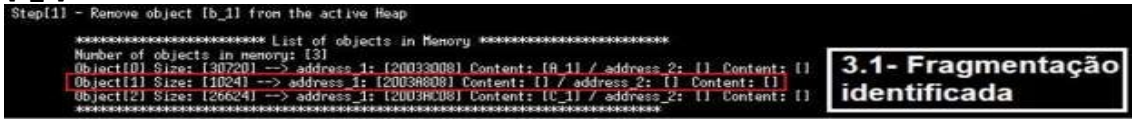
Figura 17 – Console do *software* Teraterm, mostrando as etapas executadas durante a remoção de um objeto no *heap*.

Fonte: Dados pelo autor.

4.1 Fragmentação identificada

Ao deletar o objeto [b_1], ocorreu a chamada fragmentação, que pode ser identificada na Figura 18. Repare que o conteúdo do endereço 0x2003A808 está vazio.

Figura 18 – Console do *software* Teraterm, mostrando a fragmentação gerada após a remoção do objeto [b_1].



Fonte: Dados pelo autor.

4.2 Organização da lista de objetos

Após isso, foi necessário organizar a lista de objetos para que não fossem transportados para o heap_2 objetos fragmentados, ou seja, o intuito dessa etapa foi identificar os objetos inativos em memória. A Figura 19 mostra a lista de objetos organizada.

Figura 19 – Console do software Teraterm, mostrando a lista de objetos organizada.

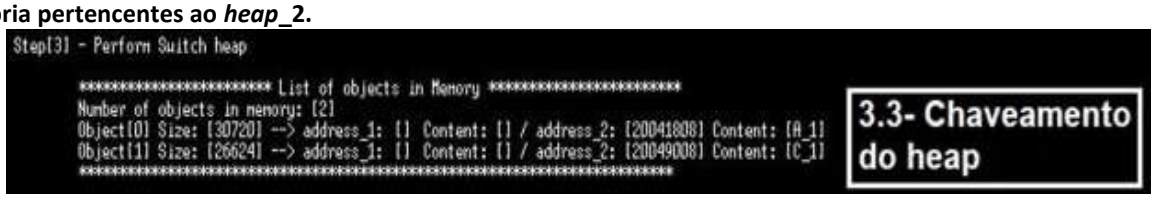
Fonte: Dados pelo autor.

Após a organização mencionada, a lista passou a conter 2 objetos apenas. No entanto, a fragmentação ainda existia e apresentava um tamanho de 1 kB. Em seguida, foi executado o chaveamento do *heap* para remover essa lacuna.

4.3 Chaveamento do *heap*

Todos os objetos do *heap*_1 foram transportados para o *heap*_2 de forma organizada e sequencial. Para isso, a lista de objetos foi utilizada, pois nela os objetos já estavam organizados. Veja, na Figura 20, que os dois objetos em memória foram realocados para novos endereços pertencentes ao *heap*_2.

Figura 20 – Console do *software* Teraterm, mostrando os objetos alocados em novos endereços de memória pertencentes ao *heap*_2.

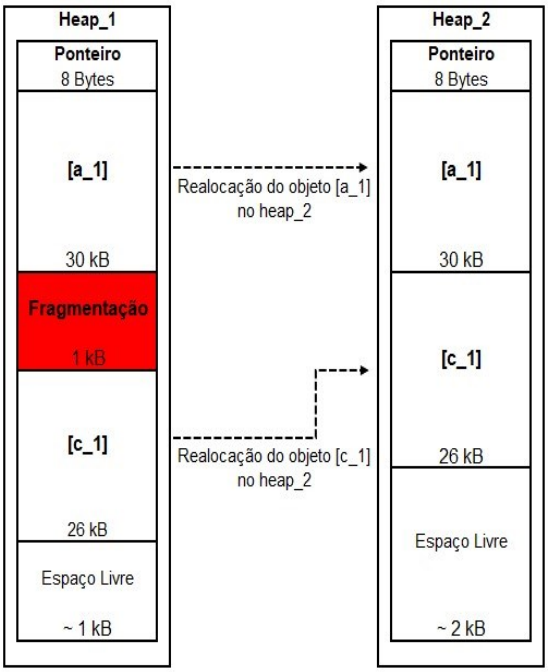


Fonte: Dados pelo autor.

Basicamente a ideia de chaveamento funciona ao coletar todos os objetos existentes no *heap*_1 e transferi-los para o *heap*_2, sem transferir a fragmentação.

Veja na Figura 21 a seguir esse processo.

Figura 21 – Representação do chaveamento dos objetos do *heap*_1 para o *heap*_2.



Fonte: Dados pelo autor.

Repare que, no *heap*_2, foram criados os dois objetos a_1 e c_1, que ocupavam um espaço total de 56 kB. Portanto, o espaço de 1 kB, que anteriormente era um espaço fragmentado, agora foi mesclado com o espaço livre do final do *heap*, totalizando aproximadamente 2 kB.

Após a execução do chaveamento do *heap*, o *heap*_1 é totalmente apagado e seu estado muda para passivo. Já o *heap*_2 muda seu estado para ativo.

Conforme mencionado anteriormente, o objeto [b_1] havia sido excluído do *heap*_1, ocasionando uma fragmentação de 1 kB. Além disso, também existia um espaço livre de 1 kB no final do *heap_1*, totalizando um tamanho de aproximadamente 2 kB. No entanto, não seria possível alocar um objeto de 1,5 kB, pois o *heap*_1 estava fragmentado e apresentava dois blocos de 1 kB, impossibilitando a alocação mencionada.

Com o chaveamento do *heap*, esses blocos fragmentados foram juntados criando um bloco livre de 2 kB.

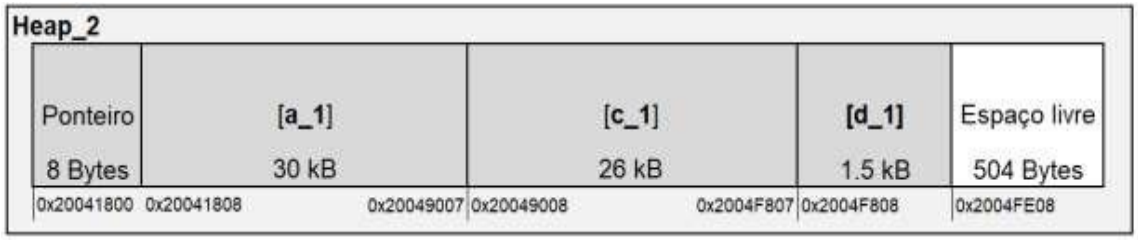
Como forma de comprovação, no cenário de teste proposto foi feita a alocação de um objeto de 1,5 kB, conforme retratado na Figura 22.

Figura 22 – Console do software Teraterm, mostrando o objeto [d_1] tamanho 1,5 kB, criado com sucesso.

Fonte: Dados pelo autor.

Portanto, a disposição de objetos ocorreu de forma sequencial, conforme retratado na Figura 23.

Figura 23 – Alocação do objeto [d_1] no *heap*_2.



Fonte: Dados pelo autor.

Caso esse procedimento de teste fosse realizado utilizando o algoritmo convencional de alocação de memória, o objeto d_1 não seria alocado por falta de espaço devido à fragmentação gerada.

Observe que sem o controle dos objetos e sem o chaveamento do *heap*, não seria possível realizar a alocação do objeto [d_1]. Isso mostra uma vantagem no algoritmo proposto neste artigo em relação ao modelo convencional de alocação de memória.

Lembrando que esse cenário de teste levou em conta apenas um único objeto; ao imaginar um cenário mais complexo, pode-se dizer que o algoritmo proposto apresenta um ganho significativo no quesito fragmentação na memória no decorrer do tempo.

No entanto, ao comparar a performance entre o algoritmo de chaveamento do *heap* com o método convencional de alocação de memória, percebe-se que existe um custo operacional elevado para gerenciar e controlar esses objetos em memória. Para realizar essa

comparação, foram realizados alguns testes com intuito de medir o tempo de alocação e desalocação dos objetos em memória.

1- Alocação de um único objeto com tamanhos distintos.

O objetivo foi realizar uma comparação para verificar o tempo de execução dos algoritmos em relação a uma única alocação entre objetos de tamanhos diferentes, conforme mostrado na Tabela 3.

Tabela 3 – Comparação entre o método convencional de alocação com o algoritmo de chaveamento do *heap*, no quesito alocação de um único objeto com tamanhos diferentes.

Alocação objetos com tamanhos distintos		
Tamanho do objeto para ser alocado (bytes)	Método convencional (pulsos de clock)	Algoritmo de chaveamento do heap (pulsos de clock)
32	778	3097
64	778	2961
128	778	2870
256	748	2822
512	707	2746

Fonte: Dados pelo autor.

2- Desalocação de um único objeto com tamanhos distintos.

O objetivo foi realizar uma comparação para verificar o tempo de execução dos algoritmos em relação a uma única desalocação entre objetos de tamanhos diferentes, conforme mostrado na Tabela 4.

Tabela 4 – Comparação entre o método convencional de alocação com o algoritmo de chaveamento do *heap*, no quesito desalocação de um único objeto com tamanhos diferentes.

Desalocação objetos com tamanhos distintos		
Tamanho do objeto para ser desalocado (bytes)	Método convencional (pulsos de clock)	Algoritmo de chaveamento do *heap* (pulsos de clock)
32	263	1330
64	263	1386
128	263	1414
256	262	1443
512	262	1472

Fonte Dados pelo autor.

3- Alocação de ‘n’ objetos.

O objetivo foi verificar o tempo de execução após ‘n’ alocações, conforme mostrado na Tabela 5.

Tabela 5 – Comparação entre o método convencional de alocação com o algoritmo de chaveamento do *heap*, no quesito quantidade de objetos a serem alocados.

Tempo de Alocação de vários objetos consecutivos em memória		
Número de objetos	Método convencional (pulsos de clock)	Algoritmo de chaveamento do *heap* (pulsos de clock)
1	872	2972
10	5199	29116
100	43089	258627

Fonte: Dados pelo autor.

4- Desalocação de um objeto após 'n' alocações.

O objetivo foi verificar o tempo de execução de uma desalocação após alocações sucessivas, conforme mostrado na Tabela 6.

Tabela 6 – Comparação entre o método convencional de alocação com o algoritmo de chaveamento do *heap*, no quesito desalocação de um único objeto após 'n' objetos alocados.

Desalocação de um único objeto, após 'n' alocações		
Número de objetos alocados	Método convencional (pulsos de clock)	Algoritmo de chaveamento do *heap* (pulsos de clock)
1	275	1343
10	275	42269
100	275	466251

Fonte: Dados pelo autor.

Ao analisar os resultados dos testes realizados, podemos inferir que o algoritmo opera da maneira esperada, pensando na proposta de desfragmentação de memória, as alocações e desalocações são feitas e não ocorrem fragmentações na memória, novas alocações podem ser feitas e a integridade dos dados é preservada.

Também é possível inferir que a performance é reduzida se comparada com um simples "*malloc*". O *malloc* realiza a alocação de memória em um endereço da memória do *heap* e possibilita a manipulação dos dados armazenados nesse espaço. O algoritmo apresentado se utiliza de diversos fluxos de controle e gerenciamento dos objetos em memória por meio de listas encadeadas. Para que o algoritmo seja utilizado em um sistema embarcado, é necessário estruturá-lo como uma biblioteca, onde chamadas de suas funções seriam utilizadas no lugar dos métodos convencionais utilizados para uso de memória dinâmica.

A solução proposta se utiliza de uma nova sequência de métodos, quando comparada com o modelo tradicional de alocações e liberações de memória. Os métodos implementados trabalham em conjunto, liberação, reorganização e troca de endereços dos objetos, para que os espaços alocados estejam sempre organizados e para que não haja lacunas entre eles. O modelo convencional de alocação e liberações de memória dinâmica se utiliza de processos

apenas de passagem de endereço para uso do espaço de memória designado. A complexidade do algoritmo proposto se encontra nas sucessivas operações para que não possibilite a criação de lacunas na memória.

Uma situação hipotética de uso do algoritmo apresentado seria uma aplicação que monitorasse eventos e enviasse e-mails criptografados a cada ocorrência. Para gerar a criptografia, a memória dinâmica seria utilizada, e também proveria recursos para o sistema operacional de tempo real. As sucessivas alocações e desalocações e a fila de eventos para envio de e-mails poderiam causar fragmentação, se utilizado o algoritmo padrão. Porém, o algoritmo proposto permitiria que a aplicação operasse de maneira adequada, permitindo o envio dos e-mails eliminando o risco de fragmentação do *heap* e trabalhando para que o *heap* permanecesse organizado durante a execução da aplicação.

Por fim, destacamos que a escolha por algoritmos mais sofisticados, como o proposto, traz consigo efeitos que devem ser avaliados de acordo com cada aplicação. O desempenho da aplicação é uma das características que sofrem alteração em relação à aplicação do método convencional de alocação e desalocação de memória.

5 CONCLUSÃO

Tendo como base a fragmentação de memória nas alocações e desalocações, o estudo apresentado possibilita que aplicações continuem a trabalhar com memória dinâmica sem que os efeitos de fragmentação de memória causem erros ao longo do tempo.

Algoritmos de reorganização e desfragmentação de memória para sistemas embarcados não são comuns, sua implementação está condicionada aos recursos e ao comportamento da aplicação. Em aplicações críticas, nas quais o recurso de desfragmentação de memória evita a parada da aplicação, o uso do algoritmo proposto se torna relevante. Porém, seu uso está condicionado à divisão da região de memória, originalmente alocado para um *heap,* entre dois *heaps* e faz uso de listas encadeadas que permitem o espelhamento dos *heaps* para a realização da troca dos endereços dos dados alocados.

Ao realizar a comparação, descrita nos resultados e discussões, verifica-se que o método tradicional não permite que a aplicação realize a alocação de memória, pois a memória está fragmentada. O novo método possibilita que a memória seja reagrupada e, assim, permite que a memória seja completamente alocada, sem lacunas. Quando o desempenho computacional é levado em conta, pode-se observar o aumento no trabalho de processamento que o algoritmo de desfragmentação de memória impõe ao processador.

Conclui-se, portanto, que a solução proposta para o tratamento de memória dinâmica é adequada principalmente para sistemas que têm processadores mais robustos, com maior poder de processamento e que executam aplicações complexas. O algoritmo oferece uma boa alternativa para que uma aplicação permaneça em execução mesmo havendo muitos processos dependentes de memória dinâmica, sem que haja a fragmentação da mesma.

Como proposta para trabalho futuro, destacamos a conversão de todas as funções criadas em uma biblioteca especializada em alocação e desalocação de memória dinâmica, que venha operar no lugar das funções convencionais e garanta a manipulação dos dados de forma transparente ao programador de sistemas embarcados com a garantia da manutenção da integridade dos dados.

REFERÊNCIAS

ANDERSSON, J. **Modeling the temporal behavior of complex embedded systems:** a reverse engineering approach. These (Department of Computer Science and Engineering) Malardalen University, Vasteras, Sweden, 2005. Disponível em: http://www.ipr.mdh.se/pdf_publications/763.pdf. Acesso em: 04 set. 2022.

BACON, D. F.; CHENG, P.; GROVE, D. Garbage collection for embedded systems. In: EMSOFT 2004 - ACM INTERNATIONAL CONFERENCE ON EMBEDDED SOFTWARE, 14. **Anais**...2004.

BASKIYAR, S.; MEGHANATHAN, N. **A survey of contemporary real-time operating systems**. **Informatica (Ljubljana)**, 2005. Disponível em: https://www.researchgate.net/publication/220166717_A_Survey_of_Contemporary_Real-time_Operating_Systems. Acesso em: 4 set. 2022

BOEHM, H. J. Dynamic memory allocation and garbage collection. **Computers in physics**, v. 9, n.3, p. 297-303, 1995. Disponível em: https://aip.scitation.org/doi/pdf/10.1063/1.4823407. Acesso em: 04 set. 2022.

BOHRA, A.; GABBER, E. Are mallocs free of fragmentation. **Usenix01**, 2001.

FERRES, L. Memory management in C: the heap and the stack. **Department of Computer. Science Universidad de Concepción**, 7 out. 2010. Disponível em: https://cs.gmu.edu/~zduric/cs262/Slides/teoX.pdf. Acesso em: 04 set. 2022.

JERSAK, M.; RICHTER, K.; ERNST, R. Performance analysis for complex embedded applications. **International Journal of Embedded Systems**, v. 1, n. 1–2, 2005. Disponível em: https://www.inderscienceonline.com/doi/abs/10.1504/IJES.2005.008807. Acesso em: 04 set. 2022.

MASMANO, M. et al. TLSF: A new dynamic memory allocator for real-time systems. - Euromicro Conference on Real-Time Systems, 16. **Proceedings**... 2004. Catania: IEEE, 2004. Disponível em: https://ieeexplore.ieee.org/document/1311009. Acesso em: 04 set. 2022.

FREERTOS. **Memory Management**. 2021. Disponível em: https://www.freertos.org/a00111.html. Acesso em: 4 set. 2022.

NEVES, F. TLSF: Gerenciador de memória em tempo real para sistemas embarcados. **Embarcados**, 2015. Disponível em: https://embarcados.com.br/tlsf/. Acesso em: 04 set. 2022.

PUAUT, I. Real-time performance of dynamic memory allocation algorithms. Euromicro Conference on Real-Time Systems, 14. **Proceedings...**2002. Vianna: IEEE, 2002. Disponível em: https://ieeexplore.ieee.org/document/1019184. Acesso em: 4 set. 2022

WILSON, P. R. et al. **Dynamic storage allocation: A survey and critical review**. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). **Anais**...1995.

YUASA, T. Real-time garbage collection on general-purpose machines. **The Journal of Systems and Software**, v. 11, n. 3, p. 181-198, 1990. Disponível em: https://www.sciencedirect.com/science/article/abs/pii/016412129090084Y. Acesso em: 04 set. 2022.

ZLATANOV, N. **Dynamic memory allocation and fragmentation**. ESC: Santa Clara, 2015. Disponível em: https://www.researchgate.net/publication/295010953_Dynamic_Memory_Allocation_and_Fragmentation. Acesso em: 04 set. 2022.

AGRADECIMENTOS

A Deus.
Aos professores e colegas de curso de Pós-Graduação em Sistemas Embarcados, que contribuíram para a realização deste trabalho com muita dedicação e conhecimento.
Agradecimentos especiais às nossas famílias, pela paciência e carinho.
À toda a equipe da Faculdade de Tecnologia SENAI "Anchieta".

SOBRE O(S)AUTOR(ES)

[i] **ELIELDER BELCHIOR DE MELO**

Graduou-se em Engenharia Eletrônica pela Universidade São Judas Tadeu (2017) fez especialização em Sistemas Embarcados pela Faculdade SENAI Anchieta (2021). Trabalha com Sistemas Embarcados desde 2016, projetou dispositivos para área médica, energia e pagamentos. Ocupa o cargo de Engenheiro de Firmware na Gertec.

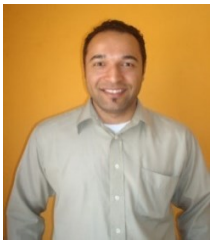
[ii] **VINICIUS AUGUSTO PASSARELLA DE MELO**

Graduou-se em Engenharia Eletrônica pela Universidade São Judas Tadeu (2017) fez especialização em Sistemas Embarcados pela Faculdade SENAI Anchieta (2021). Trabalhou diversos projetos de desenvolvimento e arquitetura de software com foco em transporte metroviário (Alstom Transporte) e em projetos de POS (Ponto de venda) McDonald's (Acrelec Brasil). Ocupa o cargo de Engenheiro Sênior na Acrelec Brasil.

[iii] **LEANDRO POLONI DANTAS**

Engenheiro (2004) e Doutor (2018) em Engenharia Elétrica pelo Centro Universitário FEI. Atuou por 15 anos na indústria eletrônica no desenvolvimento de novos produtos para diferentes segmentos. Desde 2009, vem lecionando em cursos de pós-graduação, graduação e de nível técnico em diferentes instituições paulistanas. Atualmente é professor da Faculdade SENAI Anchieta e do Insper.

[iv] **FERNANDO SIMPLICIO DE SOUSA**



Doutorando em Energia pela UFABC. Mestre em Engenharia Elétrica (2017) e Pós-Graduação (lato sensu) na instituição de Ensino Mackenzie - SP (2010). Professor da Faculdade SENAI Anchieta na graduação em Tecnologia em Eletrônica Industrial e curso de Pós-Graduação em Sistemas Embarcados. Autor do livro Programação BASIC para Microcontroladores 8051 - Ano/2006 pela editora Erica.