

Administração de Recursos Computacionais: Implementação e Teste dos Métodos de Gerenciamento de Memória

Ellison William M. Guimarães¹, Matheus S. Rodrigues¹, Saionara A. Gomes¹

¹Ciência da Computação – Universidade Estadual de Santa Cruz (UESC)
Campus Soane Nazaré de Andrade – Ilhéus – BA – Brasil

{ellison.guimaraes,theuzkye,saionara.aguiargomes}@gmail.com

***Resumo.** Pessoas estão cada vez mais tendo suas vidas inseridas nos equipamentos eletrônicos, seja das tarefas mais básicas às mais complexas. Junto com isso, há a necessidade de aprimorar os recursos computacionais a ponto de se ajustar ao uso, e um desses recursos é a memória principal. O presente artigo propõe a utilização das técnicas de gerenciamento mais usadas para administrar a memória principal e obter o melhor uso desse recurso. O trabalho contém a implementação e testes de cada um dos métodos utilizados para melhorar o desempenho e a eficiência.*

1. Introdução

Vivemos numa nova era, onde os dispositivos eletrônicos e o homem são integrantes de uma mesma sociedade. Onde se tinha lápis e caderno, e através da evolução tem sido substituído por smartphones ou tablet's. Sempre que pensamos em fazer algo, nos perguntamos como poderíamos inserir em algum dispositivo eletrônico que está mais próximo, seja para uma simples anotação ou uma automatização de uma tarefa corriqueira.

A cada dia que passa os dispositivos eletrônicos são programados a fim de se tornarem ainda mais inteligentes. Com isso, a cada oportunidade que temos, estamos sempre dando a ele as tarefas que costumávamos fazer, sempre fazendo com que ele faça algo por nós. A quantidade de informações transferidas para os computadores está crescendo a todo momento, bem como, a quantidade de atividades simultâneas a serem executadas. Visto a necessidade da sociedade do uso dos recursos tecnológicos, foi preciso ajustar os mesmos para acompanhar o crescente uso.

No computador, temos vários recursos disponíveis e cada um deles com uma função específica. Dentre os principais recursos temos o processador e a memória. O processador, bem como o nome especifica, é utilizado para processar e calcular dados, já a memória são todos os dispositivos responsáveis por armazenar esses dados, seja temporariamente ou permanentemente. E a cada dia que passa os recursos citados precisam evoluir para atender ao público, cada vez mais necessitado de processadores melhores, que processem os dados cada vez rápido a ponto de obter um resultado quase que instantâneo e de memórias com leitura e escrita cada vez mais rápidas e capacidade de armazenamento absurdamente grandes.

Teoricamente, gostaríamos de uma memória privada, infinitamente grande e rápida, barata, e que não fosse volátil, ou seja, não perdesse os dados quando a energia fosse cortada. A realidade é que não estamos evoluídos o bastante para obter uma memória que atenda estas especificações, a tecnologia que temos hoje em dia são limitadas ao pensarmos em algo semelhante, porém, estamos muito avançados quando lembramos do IBM7094, um dos primeiros computadores de aplicação científica e tecnológicas de larga escala da época (1962).

Já que não temos posse do tipo de tecnologia descritos anteriormente, temos a necessidade de utilizar os recursos disponíveis com as tecnologias atuais da melhor maneira possível. Com o que possuímos de tecnologia no momento, as memórias foram divididas em duas categorias: primária e secundária. A primária consiste numa memória de acesso mais rápido, mas de capacidade restrita, volátil e com um custo mais elevado, e a secundária sendo uma memória de acesso mais lento, porém, não volátil, capacidade bastante elevada e de menor custo.

Percebendo-se que rapidez e capacidade de armazenamento nas memórias são grandezas inversamente proporcionais, foi criada a hierarquia de memória dentro do computador. A hierarquia de memória se refere a uma classificação de tipos de memória em função de seu desempenho. Os dados percorrem essa hierarquia a ponto de melhorar o desempenho e ter um processamento ainda melhor para entregar dispositivos ainda mais rápidos para o uso corriqueiro. A hierarquia geralmente é demonstrada através de pirâmides, onde no topo existem alguns megabytes de memória cache (muito rápida, volátil e custo alto), alguns terabytes de armazenamento em disco (baixa velocidade, não volátil e de baixo custo) e entre elas alguns gigabytes de memória principal (volátil, custo médio e velocidade médio) que é exatamente dela que iremos abordar neste artigo.

Visto as dificuldades enfrentadas, devemos administrar nossos recursos computacionais da melhor forma possível. Com isso, nesse artigo iremos conhecer e implementar as técnicas mais utilizadas para o melhor gerenciamento da memória principal (*Random Access Memory* - RAM) do computador.

2. Gerenciamento de memória

Em um sistema multiprogramado, onde a memória é compartilhada entre vários processos, o gerenciamento de memória principal é de extrema importância em um computador. O gerenciador de memória é responsável por controlar espaços livres e alocados disponíveis, proteger cada área ocupada por cada processo e fazer o chaveamento entre os níveis de hierarquias presente na Figura 1. Em um sistema multiprogramado há a necessidade de manter o maior número possível de processos, maximizando o compartilhamento da CPU (*Central Processing Unit*) e dos demais recursos, para entregar um resultado cada vez mais rápido ao usuário para cada uma de suas tarefas.

Os programadores não têm acesso a memória física contida em um computador, é totalmente restrito a manipulação de endereços físicos diretamente, temos acesso somente a endereçamento lógico. Os endereços lógicos são abstrações de endereços físicos. Um dos maiores motivos desse funcionamento é para maior proteção da memória física, pois expor a memória física a programadores e processos, podendo

endereçar qualquer parte da memória utilizada por outros processos, ou até para o sistema operacional, pode danificar ou paralisar o sistema. E por trás disso, o gerenciador de memória junto ao MMU (*Memory Management Unit*) tem a responsabilidade de fazer essa conversão de endereços e gerenciar as trocas entre memórias da hierarquia.

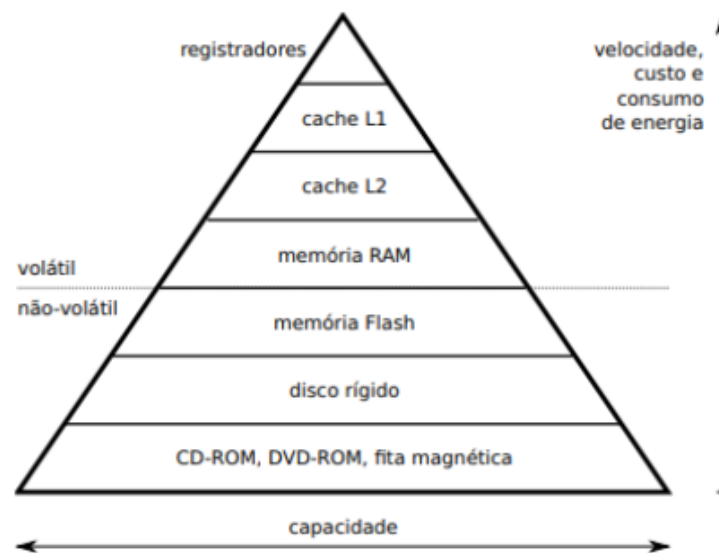


Figura 1. Hierarquia de memórias presente nos computadores.

Constantemente processos concorrem por um espaço na memória principal para serem executados, devemos estudar as melhores maneiras de inserir para que aproveitem a capacidade da melhor forma. Suponha que a memória seja como um vetor sequencial que tem seus índices numerados começando do índice 0 até um índice final (depende da capacidade e configuração), e que cada processo toma uma pequena parte desse vetor. Na Figura 2, temos a representação de memória e dos processos contidos nela. Quando alocamos um espaço de tamanho α na memória para um processo, o espaço tem um endereço inicial (base) e limite (comprimento).

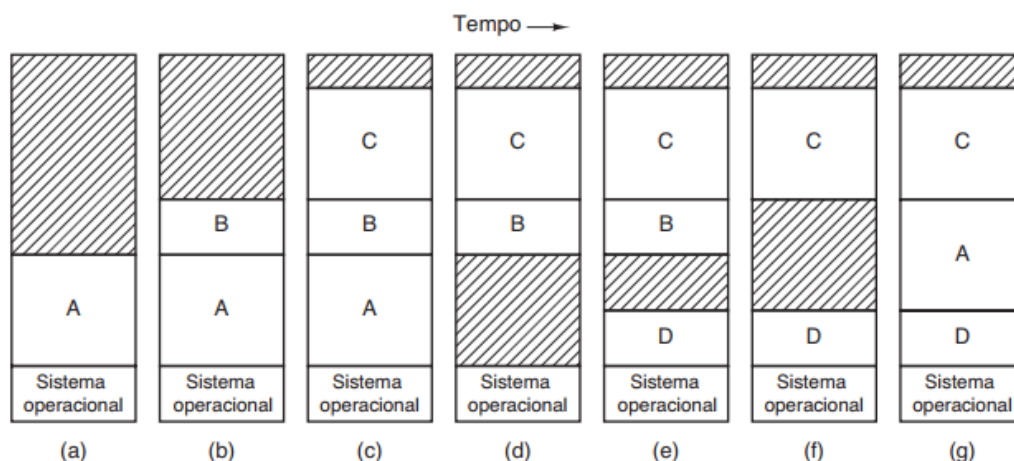


Figura 2. Representação dos processos na memória.

Entendendo o funcionamento da memória internamente, Figura 3(a), devemos entender as estruturas mais utilizadas para o gerenciamento: utilização dos Mapas de Bits, Figura 3 (b), que estrutura a memória de forma que seja dividida em unidades de alocação, com cada unidade correspondendo a um bit no mapa de bits, onde se o bit for um, o espaço está ocupado, e se o bit for zero o espaço está livre, assim, podemos nos informar dos espaços que estão vazios e preenchidos. A outra estrutura existente é a lista encadeada, Figura 3(c), que vamos discutir e implementar neste artigo.

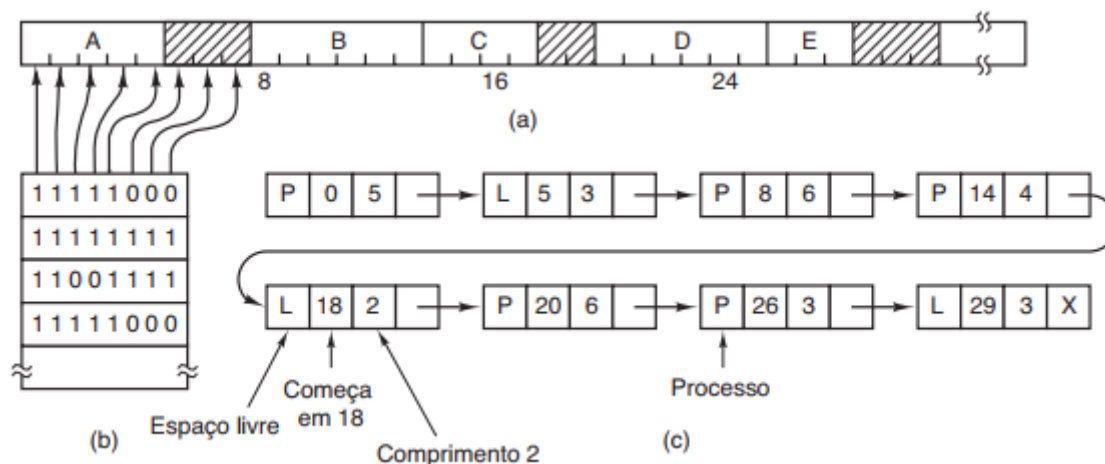


Figura 3. Demonstração dos mapas de bits e lista encadeada.

Na lista encadeada, Figura 3(c), podemos observar que cada bloco alocado contém quatro informações: A primeira quando P (identificação do processo) significa que o espaço está alocado e L quando o espaço está vazio; A segunda informa o endereço inicial (base) na memória; A terceira o comprimento do espaço alocado para o processo; A quarta é um ponteiro ou endereço, que aponta e identifica o próximo bloco na memória.

A memória, como foi esclarecido anteriormente, recebe processos constantemente para serem alocados. Com isso, devemos saber onde encaixar esses novos blocos de processos que chegam para serem armazenados, e finalmente, chegamos ao ponto deste artigo, em que falamos sobre as técnicas de inserção, ou encaixe, de processos na memória principal do computador para a melhor utilização da memória.

3. Gerenciando listas encadeadas com algoritmos de encaixe

Existem vários algoritmos para gerenciar o encaixe dos processos na memória, porém, nesse artigo, iremos abordar e implementar quatro técnicas de encaixe específicas. Veremos cada uma dessas técnicas com uma abordagem teórica explicativa em seguida algoritmos implementados na linguagem Python, com gráficos gerados pelo próprio

para facilitar a leitura da quantidade de espaço alocado por cada processo contido na memória. Utilizarei a estrutura de dado Lista Encadeada Simples para representação da memória.

3.1. Métodos de encaixe

Dentre os métodos de encaixe, iremos apresentar quatro tipos de abordagens, onde cada uma delas tem características específicas e contextos a serem utilizadas. Não existe algoritmo melhor, cada um dos algoritmos tem suas especificações e atendem a determinada função, basta escolher o mais ideal para cada tipo de aplicação diferente. Os quatro métodos são: Primeiro encaixe ou *first fit*; próximo encaixe ou *next fit*; melhor encaixe ou *best fit*; pior encaixe ou *worst fit*.

O algoritmo mais simples é o de primeiro encaixe (*first fit*). Tendo um processo a ser inserido na memória, o gerenciador de memória percorre todos os segmentos procurando o primeiro bloco vazio que couber o processo. Ao encontrar, verifica se o segmento vazio é maior ou igual ao tamanho do processo, se maior, o processo é alocado logo no início do bloco vazio, se igual, o segmento preenche exatamente o bloco vazio.

O segundo algoritmo é o próximo encaixe (*next fit*). Ele é muito semelhante ao de primeiro encaixe, a diferença, é que o de primeiro encaixe procura desde o início da memória, e o de próximo encaixe procura a partir do último processo inserido na memória, que teoricamente há sempre um espaço vazio caso a última inserção tenha sido menor que o espaço alocado. Tem desempenho um pouco inferior ao de primeiro encaixe.

Um dos métodos mais utilizados é o de melhor encaixe (*best fit*). Ele percorre toda a memória procurando o menor espaço possível que possa caber o processo desejado, ou, se achar um bloco vazio de tamanho idêntico a ele é alocado no mesmo instante. Ele é muito mais lento que o de primeiro encaixe, justamente por ter que percorrer toda a memória em busca de um segmento ideal. Esse método gera fragmentações extremamente pequenas que na maioria das vezes nunca são utilizadas, desperdiçando mais memória.

E o ultimo algoritmo é o de pior encaixe (*worst fit*). Ele é bastante semelhante ao de melhor encaixe, a diferença, é que o de pior encaixe procura o maior segmento para encaixar o processo. Esse método foi a solução criada para o de melhor encaixe, já que agora são criados fragmentos maiores, dando a oportunidade de outros processos alocarem esses espaços.

Na Figura 3, exemplificamos a inserção de dois segmentos de 14kbytes e 20kbytes na memória com um estado inicial. Foram utilizados todos os métodos de encaixe anteriormente citados. Foi observado que o método de melhor encaixe obteve a menor fragmentação, porém são blocos extremamente pequenos a ponto de ser inúteis, enquanto o de pior encaixe deixou uma fragmentação grande o suficiente para acomodar outro processo. O de primeiro e próximo encaixe obteve quase o mesmo resultado com esse estado de memória e ponteiros para os últimos encaixados.



Figura 3. Exemplos dos algoritmos de encaixe.

3.2. Descrição do problema: Implementação

Em posse dos métodos de gerenciamento de encaixe na memória, iremos implementar uma solução para simular um sistema de gerenciamento de memória utilizando os métodos citados. Será implementado uma Lista Encadeada Simples de blocos livres e alocados. Inicialmente a memória terá tamanho 100M, e os processos inseridos entre 2M, 4M, 8M ou 16M, onde poderá escolher qual dos algoritmos será utilizado para encaixar o processo na memória. O simulador também terá a opção de remover um processo na memória pelo número de identificação, bem como compactar a memória a fim de reorganizar os processos de modo que não fique qualquer espaço livre entre eles.

O simulador precisa unir todos os segmentos em branco que estão juntos, além disso, mostrar informações como: listar todos os processos carregados na memória; mostrar espaço total e espaço total disponível da memória; mostrar na tela a situação atual da memória, espaços ocupados por processos e espaços livres.

3.3. Solução: Algoritmo

Para a construção do algoritmo foi utilizado a linguagem de programação *Python 3* junto com a biblioteca *Matplotlib*, responsável por gerar gráficos em 2D para visualização de dados. Utilizamos a biblioteca para gerar gráficos de setores ou circular, Figura 4(b), para visualização do espaço alocado por cada processo na memória, bem como o espaço total, vazio e ocupado. Utilizamos classes para representar os blocos de memória (*section*) e para representar as instruções realizadas na memória (*memory*). O algoritmo possui uma pequena interface em terminal para facilitar o gerenciamento da memória, Figura 4(a).

A classe `section` é composta pelo método `is_void` que retorna um valor booleano informando se o bloco é vazio ou não, e também é composta pelos atributos: `id` que é a identificação do processo; `name` que é o nome dado ao processo; endereço inicial (`base`) e comprimento (`size`); endereço para o próximo bloco (`nearby`).

A classe `memory` é composta pelos seguintes métodos: o método construtor, que inicializa as variáveis `max_limit` com o tamanho total da memória, a `ids` sendo uma lista vazia (lista de identificação de todos os processos na memória), o `address_initial` que é o endereço para o início da Lista Encadeada Simples e o `last_access` sendo o endereço do último bloco inserido na memória (utilizado apenas para o próximo encaixe, *next fit*); os métodos de encaixe `store_firstfit`, `store_nextfit`, `store_bestfit` e `store_worstfit`; o método `remove` responsável por remover um processo através do `id`; o método `compress` responsável por compactar os processos; o método `join_empty_memory` que une blocos vazios que estão juntos; `generate_id` que gera um novo `id` não repetido; `show_memory` que mostra a situação atual da memória, espaços ocupados e vazios; `show_memory_info` que retorna a quantidade de espaço em branco e usados na memória; `show_programs` que mostra todos os processos presente na memória.

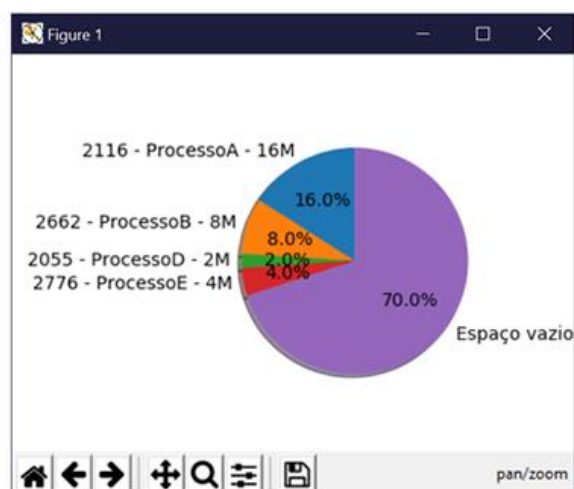
O programa conta com uma pequena interface em terminal para facilitar a utilização dos métodos implementados e descritos no problema do tópico anterior. Na Figura 4(a) podemos ver as opções disponíveis: carregar um programa na memória, onde é necessário informar o nome do processo, o comprimento total e o método de encaixe; listar programas carregados na memória; remover um programa na memória através do `id`; mostrar espaço total e disponível na memória; mostrar estado atual da memória, com processos e espaços vazios; compactação de memória de forma que não fique espaços vazios entre os processos.

Toda vez que um método de encaixe é chamado, no corpo do procedimento é chamado o método `join_empty_memory` para unir os blocos vazios que estão juntos.

```
*****
*          SIMULADOR MEMÓRIA          *
*****
1 - Carregar um programa na memória
2 - Listar programas carregados na memória
3 - Remover um programa da memória
4 - Mostrar espaço total disponível na memória
5 - Mostrar estado atual da memória
6 - Compactar memória
7 - Sair
*****
Opção: 5

Show Memory:
ID      NOME      BASE      TAMANHO
2116    ProcessoA  0         16
2662    ProcessoB  16        8
2560    ProcessoC  24        8
2055    ProcessoD  32        2
2776    ProcessoE  34        4
0       Void       38        62
```

(a) Interface apresentada pelo algoritmo



(b) Gráficos gerados pelo *Matplotlib*

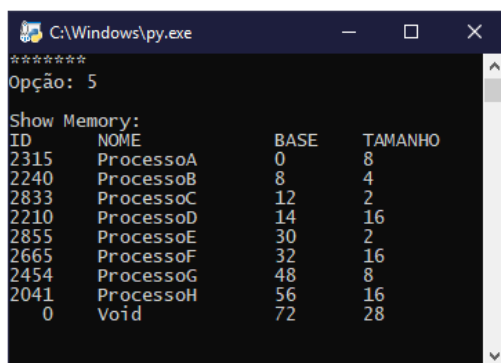
Figura 4. Demonstração do algoritmo de simulação desenvolvido.

O código completo da implementação em *Python* e todas as instruções necessárias para a execução do programa, encontra-se na página do *GitHub* (<https://github.com/ellisonguimaraes/SO-MemorySimulator>). A implementação conta com quinhentas e cinquenta e duas linhas de código, totalmente comentadas. O arquivo *readme.md* contém instruções completas de como instalar o *Python* e a biblioteca *Matplotlib* e também as instruções necessárias para executar o arquivo *memory.py*.

4. Resultados e testes

Para testar o algoritmo simulador de gerenciamento de memória, iremos inserir uma lista de processos na memória e observar como irá se comportar. Iniciaremos a memória com oito processos de tamanhos variados, que podem ser observados na Figura 5(a). O primeiro passo será remover três desses processos, o segundo passo inserir um processo para cada método de encaixe existente para analisarmos seu funcionamento, no terceiro passo iremos testar a compactação da memória. As opções do menu de visualização da memória serão vistas ao decorrer dos testes.

Antes de remover alguns dos processos, iremos utilizar a opção cinco do menu para exibir o estado atual da memória, ou seja, espaços ocupados por processos e espaços vazios, Figura 5(a). Agora iremos remover os processos **A**, **D** e **G** através dos números de identificação 2315, 2210 e 2454 respectivamente e em seguida listar os processos e espaços livres usando a opção cinco do menu, Figura 5(b). Após a remoção, foi gerado três espaços vazios, vamos utilizar eles para aplicar os métodos de encaixe.

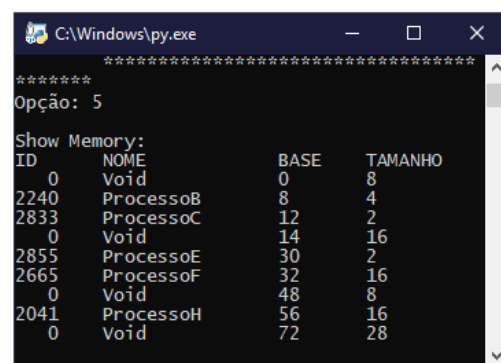


```

*****
Opção: 5

Show Memory:
ID      NOME      BASE  TAMANHO
2315    ProcessoA    0      8
2240    ProcessoB    8      4
2833    ProcessoC   12      2
2210    ProcessoD   14     16
2855    ProcessoE   30      2
2665    ProcessoF   32     16
2454    ProcessoG   48      8
2041    ProcessoH   56     16
0       Void      72     28
  
```

(a) Estado inicial

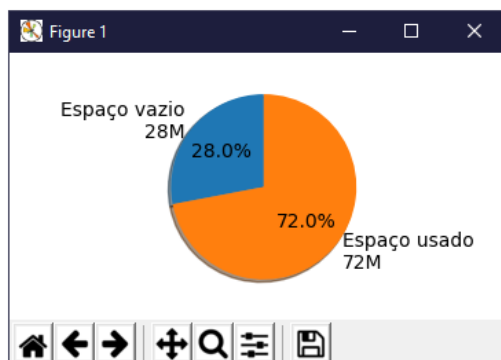


```

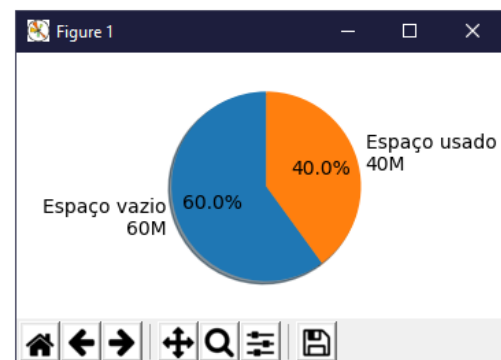
*****
Opção: 5

Show Memory:
ID      NOME      BASE  TAMANHO
0       Void      0      8
2240    ProcessoB    8      4
2833    ProcessoC   12      2
0       Void     14     16
2855    ProcessoE   30      2
2665    ProcessoF   32     16
0       Void     48      8
2041    ProcessoH   56     16
0       Void     72     28
  
```

(b) Após remoção dos processos



(c) Espaço usado e vazio no estado inicial



(d) Espaço usado e vazio depois da remoção

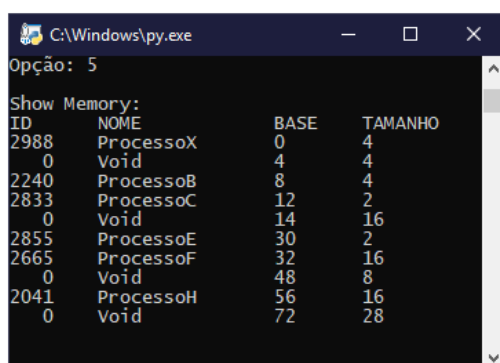
Figura 5. Primeiro passo do teste das funcionalidades.

Iremos aplicar inicialmente um processo de nome *ProcessoX* de tamanho 4M através do método de primeiro encaixe. Poderá ser observado na Figura 6(a) que foi adicionado ao início da memória, onde se tinha um segmento vazio de 8M, no qual, uma parte dele foi tomado pelo processo e a outra parte ficou vazia com 4M.

Através do método de melhor encaixe, iremos adicionar o processo de nome *ProcessoY* de tamanho 8M na memória. Ele foi encaixado no menor espaço vazio que o coubesse, entre o processo **F** e o processo **H**, Figura 6(b). Sabendo-se que o ponteiro do último segmento adicionado é o do processo **Y**, iremos adicionar o processo de nome *ProcessoZ* de tamanho 16M através do método do próximo encaixe. E foi observado na Figura 6(c) que ele foi encaixado após o processo **H**, já que seria o primeiro encaixe após a inserção anterior.

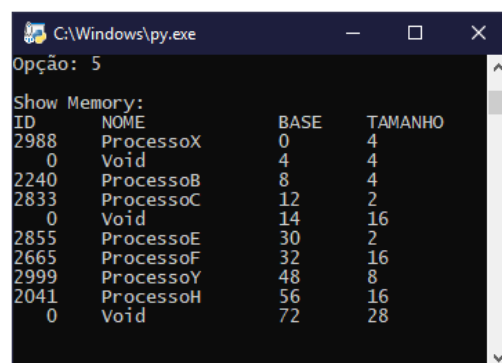
Finalmente testaremos o método de pior encaixe. Iremos adicionar um processo de nome *ProcessoK* de tamanho 2M, que procura o maior segmento vazio para alocar, e obtemos como resultado, visto na Figura 6(d), que ele foi alocado entre os processos **C** e **E**, gerando um fragmento relativamente grande de 14M.

Anteriormente como visto no gráfico da Figura 5(d), a memória tinha um espaço vazio de 60M e um espaço usado de 40M. Depois do teste dos quatro métodos de encaixe, a memória ficou com 30M de espaço vazio e 70M de espaço usado, como visto na Figura 7(a) e na Figura 7(c) podemos ver o gráfico em setores dos processos alocados na memória usando a opção 2 do menu da Figura 4(a).



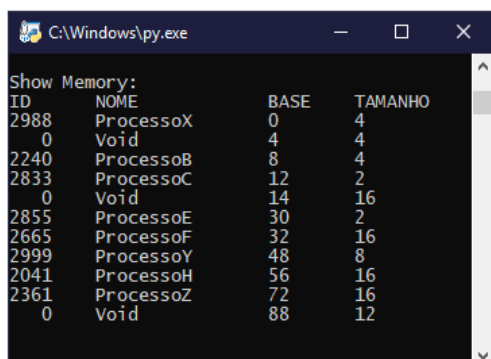
ID	NOME	BASE	TAMANHO
2988	ProcessoX	0	4
0	Void	4	4
2240	ProcessoB	8	4
2833	ProcessoC	12	2
0	Void	14	16
2855	ProcessoE	30	2
2665	ProcessoF	32	16
0	Void	48	8
2041	ProcessoH	56	16
0	Void	72	28

(a) Após adicionar o *processoX*.



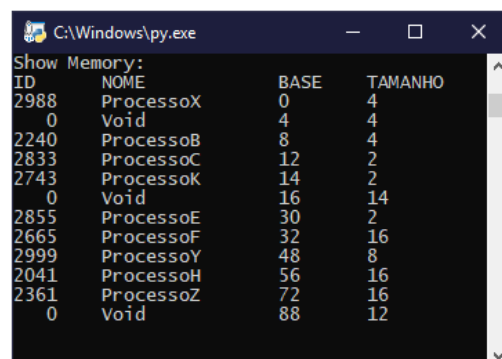
ID	NOME	BASE	TAMANHO
2988	ProcessoX	0	4
0	Void	4	4
2240	ProcessoB	8	4
2833	ProcessoC	12	2
0	Void	14	16
2855	ProcessoE	30	2
2665	ProcessoF	32	16
2999	ProcessoY	48	8
2041	ProcessoH	56	16
0	Void	72	28

(b) Após adicionar o *processoY*.



ID	NOME	BASE	TAMANHO
2988	ProcessoX	0	4
0	Void	4	4
2240	ProcessoB	8	4
2833	ProcessoC	12	2
0	Void	14	16
2855	ProcessoE	30	2
2665	ProcessoF	32	16
2999	ProcessoY	48	8
2041	ProcessoH	56	16
2361	ProcessoZ	72	16
0	Void	88	12

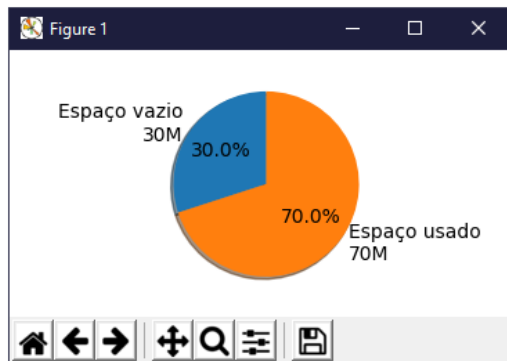
(c) Após adicionar o *processoZ*.



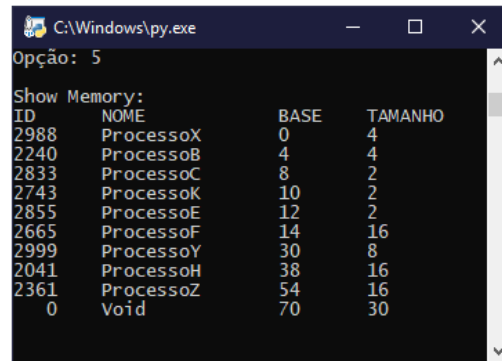
ID	NOME	BASE	TAMANHO
2988	ProcessoX	0	4
0	Void	4	4
2240	ProcessoB	8	4
2833	ProcessoC	12	2
2743	ProcessoK	14	2
0	Void	16	14
2855	ProcessoE	30	2
2665	ProcessoF	32	16
2999	ProcessoY	48	8
2041	ProcessoH	56	16
2361	ProcessoZ	72	16
0	Void	88	12

(d) Após adicionar o *processoK*.

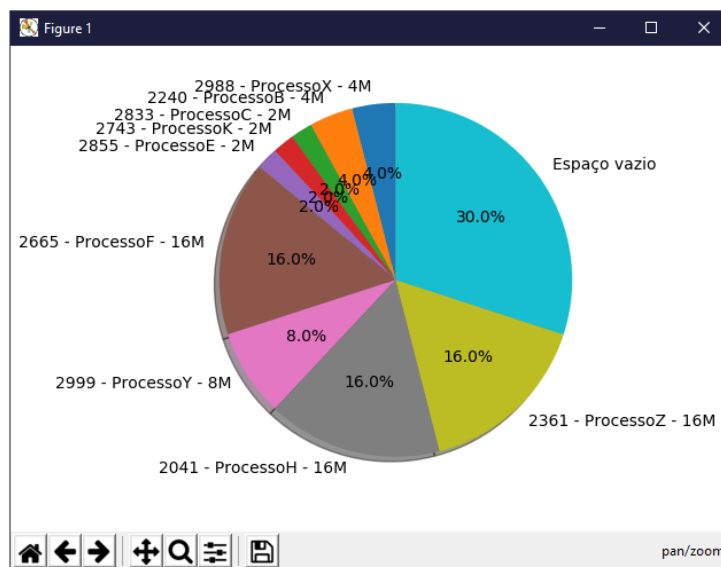
Figura 6. Situação de memória após testes dos métodos de encaixe.



(a) Espaço vazio e espaço usado após inserção



(b) Após a compactação da memória



(c) Processos na memória

Figura 7. Informações da capacidade da memória pós inserção de processos

Para finalizar os testes, realizaremos a compactação da memória, ou seja, reorganizar os processos de modo que não exista qualquer espaço livre entre eles, e de modo geral, levando todo o espaço vazio para o fim da memória. A opção 6, no menu da Figura 4(a), corresponde a compactação da memória, e ao ser usada, os processos são reorganizados visto na Figura 7(b).

5. Conclusões

Vimos que a administração dos recursos computacionais é de extrema importância para obter o melhor dos recursos disponíveis. Quando se trata do gerenciamento de memória se torna ainda mais preocupante, pois o mau uso dela pode gerar problemas graves ao sistema operacional. Por esse motivo, e com base em grandes estudos sobre como se organizam os dados e processos na memória, foram criadas técnicas para eficácia deste recurso.

Sabendo-se que com a nossa tecnologia atual, os recursos computacionais ainda são bastante limitados, mas com planejamento, organização e uma gestão precisa dos recursos computacionais, conseguimos realizar a maior parte dos objetivos necessários para as tarefas diárias que atribuímos aos dispositivos, sem afetar a sua integridade.

Os métodos de encaixe são uma excelente alternativa para a inserção de processos na memória. Através deste artigo podemos concluir que não existe o melhor método, e sim aquele ideal para o tipo de aplicação, basta, através de análises, descobrir o qual resulta num melhor desempenho do sistema. E também, através da implementação dos métodos observamos que quanto mais complexo o algoritmo for, mais tempo da CPU vai ser necessário, assim, a escolha de métodos mais simples e que tenha um resultado médio, pode ser a melhor escolha.

Quanto a parte prática, todos os objetivos foram cumpridos. O teste dos métodos de encaixe e da compactação da memória resultaram da melhor forma possível. A utilização de Listas Encadeadas Simples para representação da memória facilitou muito os objetivos requeridos no problema inicial. A linguagem de programação utilizada foi uma ótima escolha, principalmente pela disponibilidade de gerar gráficos facilmente.

Referências

- Stallings, William. (2005) Sistemas Operativos / traducción y revisión técnica José Mariá, Fernando Pérez, María de los Santos, Victor Robles y Francisco Javier – 5ª. ed. – Madrid: Pearson Educación España.
- Tanenbaum, Andrew S. (2016) Sistemas operacionais modernos / Andrew S. Tanenbaum, Herbert Bos; tradução Jorge Ritter; revisão técnica Raphael Y. de Camargo. – 4. ed. – São Paulo: Pearson Education do Brasil.
- Toy, Wing; Zee, Benjamin (1986). Computer Hardware/Software Architecture. Prentice Hall. p. 30. ISBN 0-13-163502-6.
- Basic pie chart, Matplotlib. Disponível em: <https://matplotlib.org/3.1.1/gallery/pie_and_polar_charts/pie_features.html>. Acesso em: 17 de ago. de 2019.