

**UNIVERSIDADE FEDERAL DE VIÇOSA
CAMPUS FLORESTAL**

Trabalho Prático II Sistemas Operacionais

Artur Souza Papa - 3886

Luciano Belo de Alcântara Júnior - 3897

Maria Theresa Henriques - 3486

Trabalho Prático II apresentado à disciplina
de Sistemas Operacionais do curso de
Ciência da Computação da Universidade
Federal de Viçosa.

**Florestal
26 de Julho de 2022**

CCF 451 - Sistemas Operacionais

Trabalho Prático II

Artur Souza Papa - 3886

Luciano Belo de Alcântara Júnior - 3897

Maria Theresa Henriques - 3486

26 de Julho de 2022

Contents

1	Introdução	3
2	Contexto conceitual	3
2.1	Gerência de memória	3
2.2	Algoritmos de alocação	3
2.2.1	First fit	3
2.2.2	Next fit	3
2.2.3	Best fit	4
2.2.4	Worst fit	4
3	Implementação	4
3.1	Divisão de pastas	4
3.2	Gerenciamento de memória	5
3.2.1	Tamanho e número de blocos da memória	5
3.2.2	Lógica de geração da memória	6
3.2.3	Algoritmos de alocação	7
3.2.4	Lógica comum aos algoritmos	7
3.2.5	First fit	8
3.2.6	Next fit	9
3.2.7	Best fit	10
3.2.8	Worst fit	11
3.2.9	Desalocação e integração da memória	11
3.3	Métricas	12
3.3.1	Fragmentos externos	12

3.3.2	Percurso médio de nós para alocação	13
3.3.3	Percentual de alocação negada	14
3.4	Memória Virtual	15
3.4.1	TLB em detalhes	15
3.4.2	TLB hit	15
3.4.3	TLB miss	15
3.4.4	Paginação	16
3.5	Impressão	16
3.6	Adaptações necessárias	17
3.7	Domínio	17
4	Discussão dos resultados	18
5	Conclusão	19

1 Introdução

Este trabalho tem como objetivo o estudo e aplicação dos conceitos de gerenciamento de memória através do desenvolvimento de uma simulação de alocação de memória principal e a implementação de um mecanismo de memória virtual.

Dessa forma, o trabalho foi dividido em duas etapas, construção conceitual do que se espera e aplicação prática do que foi estudado. Assim, em um primeiro momento apresentaremos os principais conceitos que serão implementados em nossa simulação. Em seguida, serão apresentadas explicações ilustradas das funcionalidades codificadas no projeto.

2 Contexto conceitual

2.1 Gerência de memória

A demanda por mais memória e memórias mais rápidas capazes de armazenar e executar todos os programas desejados em multiprogramação ainda não são uma realidade barata nos dias atuais. Sendo assim, para lidar com adversidades de disponibilidade de espaço dos sistemas faz-se necessário a utilização de funções de gerenciamento de memória.

Para lidar com a alocação de processos neste trabalho, faremos uso de quatro diferentes técnicas quanto à escolha do local de reservar a memória, sendo elas: *first fit*, *next fit*, *best fit* e *worst fit*.

2.2 Algoritmos de alocação

2.2.1 First fit

O algoritmo da primeira alocação, também conhecido como *first fit* possui a função de procurar pelo primeiro lugar na memória, representada por uma lista ou vetor, que possua espaço suficientemente grande para armazenar o processo. A principal vantagem desse algoritmo trata-se do tempo de execução do mesmo, visto que ele gasta o tempo mínimo em operações de pesquisa por espaço. Já sua maior desvantagem é a alta probabilidade de gerar fragmentação interna nos blocos onde os processos foram alocados durante sua execução.

2.2.2 Next fit

Assim como o algoritmo *first fit*, o algoritmo da próxima alocação, também possui a função de procurar pelo primeiro lugar na memória que possua espaço suficientemente grande para armazenar o processo, porém, sempre iniciando a busca a partir da posição seguinte onde o algoritmo alocou o último processo. Igualmente ao algoritmo *first fit*,

apresenta grande vantagem no tempo de execução quando comparado aos outros algoritmos, visto que ele também gasta o tempo mínimo em operações de pesquisa por espaço. Semelhante ao algoritmo supracitado, a maior desvantagem são as maiores chances de geração de fragmentação interna nos blocos onde os processos foram alocados.

2.2.3 Best fit

O algoritmo da melhor alocação, comumente conhecido como *best fit*, busca na estrutura de armazenamento de memória a página que proporciona o espaço mais ajustado ao processo que se busca alocar e que ainda não foi ocupada por outro processo. Este é um algoritmo mais lento, dado que ao contrário dos anteriores, faz-se necessário percorrer toda a lista de blocos de memória para encontrar qual página é a melhor opção dentro dos critérios do algoritmo, ou seja, possui tamanho livre e disponível mais próximo ao tamanho do processo que precisa ser alocado.

2.2.4 Worst fit

Similar ao algoritmo da melhor alocação, o algoritmo de pior alocação (*worst fit*), realiza uma varredura na memória em busca de uma página que disponha do maior espaço livre para alocação de um processo, sendo que esse valor tem de ser igual ou superior ao tamanho do processo que se busca alocar. Este também possui como característica o fato de ser um algoritmo mais lento que os dois primeiros apresentados, novamente, dado que faz-se necessário examinar toda a lista de blocos de memória na procura de uma página que melhor se encaixe nas regras do algoritmo.

3 Implementação

Para o desenvolvimento dos códigos, foi utilizado o sistema operacional Linux, a linguagem de programação C, o compilador GCC e o editor de código-fonte Visual Studio Code.

Tanto o sistema operacional quanto a linguagem de programação foram escolhidos devido às restrições de requisitos do projeto, para além das possibilidades que os mesmos oferecem por ser, respectivamente, um sistema de código aberto com ampla documentação e por ser uma linguagem de propósito geral com alta flexibilidade de uso.

3.1 Divisão de pastas

O trabalho foi desenhado com uma estrutura organizacional de pastas de modo a separar os arquivos em: (1)componentes; (2)funções; (3)compartilhados; (4) logs; (5) métricas; e,

por fim, (6) domínio. Com isso, temos seis principais diretórios: *components*, *core*, *shared*, *log*, *metrics* e *domain* (tanto no **src** como em **headers**).

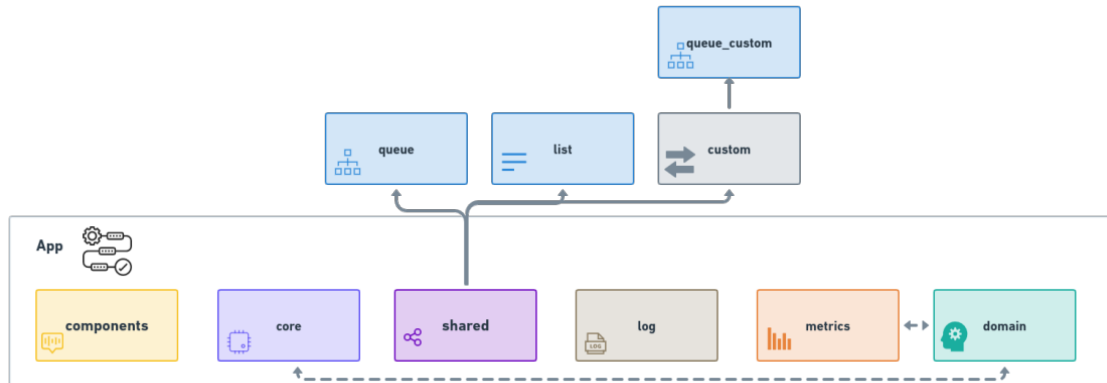


Figura 1: Divisão de pastas no programa

3.2 Gerenciamento de memória

Para realizar esse estudo, primeiro apresentamos os critérios de escolha e configuração quanto ao gerenciamento de memória, seguido das funções necessárias para sua utilização.

3.2.1 Tamanho e número de blocos da memória

O tamanho da memória principal do programa foi escolhido de forma arbitrária, tomando-se como tamanho o valor de 2 MB de memória (MEMSIZE), ou, o equivalente a 2048 kilobytes. Além disso, o tamanho da memória foi dividido em dez blocos, tal escolha deve-se ao fato de na primeira simulação a ser criada para simular os processos termos observado um comportamento de geração de no máximo seis processos na tabela de processos. Logo, tal peculiaridade foi importante na tomada de decisão da quantidade máxima de blocos que o simulador de memória deveria ter, ou seja, sendo esse valor o mais próximo desse valor.

```
1 // 2*1024
2 // Processo tem 2 MB de memória
3 #define MEMSIZE 2048 // em KBs
4 #define SIZE 10
```

Figura 2: Código referente a definição da memória principal

3.2.2 Lógica de geração da memória

No que concerne à configuração da memória, para a construção e configuração da mesma foi construída a função *generate()*, responsável por gerar blocos de memória de forma aleatória. Em pormenores, a função itera até o tamanho definido na memória principal, de modo a incrementar o valor de um kilobyte de memória de forma randômica nos blocos de memória, de maneira a garantir que a soma de todos os blocos seja o valor total de memória principal.

A segunda iteração do algoritmo diz respeito à construção do arquivo com extensão **.dat**, que trata-se de um arquivo de dados genérico encarregado de armazenar informações específicas relacionadas ao programa o qual o originou [1]. Neste contexto, o arquivo será configurado com os tamanhos de bloco e seus respectivos índices na memória como na figura 02.

```
1  for (int i = 0; i < MEMSIZE; i++)
2  {
3      // incrementa 1KB de memória por vez
4      memory[rand() % SIZE]++;
5  }
6
7  for (int i = 0; i < SIZE; i++)
8  {
9      i = SIZE - 1 ?
10     fprintf(file, "%d %d", i, memory[i]) :
11     fprintf(file, "%d %d\n", i, memory[i]);
12 }
```

Figura 3: Código referente a definição da memória principal


Além disso, como necessariamente temos que ter um único vetor, para armazenar as variáveis dos processos simulados e essas devem ficar adjacentes na memória, foi criada a função *init_main_memory()*, que percorre o arquivo **memory.dat** localizado na pasta/*-data*, que define os valores de cada bloco.

```
1  typedef struct
2  {
3      int empty;
4      int allocated;
5  } main_memory;
```

Figura 4: Código referente a estrutura da memória principal

Vale ressaltar que no começo da estrutura cada posição do array de *main_memory*

possui valor *empty*. Não diferente, o tamanho total do bloco também possui valor vazio (*empty*), já que nenhum processo foi alocado durante a inicialização, logo, por definição o valor do campo *allocated* é igual a 0.



```
1  main_memory *memory = (main_memory *)malloc(SIZE * sizeof(main_memory));
2
3  FILE *file;
4  const char *path = "data/memory.dat";
5  int index, value;
6
7  file = fopen(path, "r");
8
9  while (!feof(file))
10 {
11     fscanf(file, "%d %d", &index, &value);
12     memory[index].empty = value;
13     memory[index].allocated = 0;
14 }
15
16 return memory;
```

Figura 5: Código referente a função de inicialização da memória principal `init_main_memory()`

Por fim, a ideia desses campos é termos o controle do tamanho total de cada bloco, sendo que para conhecer o tamanho do bloco basta somar os valores das variáveis inteiras *empty* e *allocated*. Por exemplo, um bloco de tamanho 200 kb, no qual foi alocado um processo de tamanho 136 kb, tem-se que 64 kb de memória estará vazia.

$$\begin{aligned} \text{memory}[x] &= \text{memory}[x].\text{allocated} + \text{memory}[x].\text{empty} \\ 200\text{kb} &= 136\text{kb} + 64\text{kb} \end{aligned}$$

3.2.3 Algoritmos de alocação

Dado que os algoritmos de alocação já foram descritos acima, esta seção se limitará a mencionar as peculiaridades de implementação dos algoritmos citados.

No entanto, informamos de antemão que não entraremos em detalhes nesta seção no que diz respeito às funções referentes às métricas de alocação, sendo elas: *increment_total_allocation_request* e *increment_denied_allocation_request*.

3.2.4 Lógica comum aos algoritmos

Para evitar explicação de detalhes de implementação comuns aos algoritmos descreveremos aqui parte da lógica igualmente usada em todos os algoritmos de alocação.

Nas funções de alocação são realizadas as seguintes ações: primeiro define-se o valor do index com valor de -1, esse valor será posteriormente usado no algoritmo para realizar verificação de alocação ou não do processo na memória. Em seguida, se o valor do index for diferente do que previamente definido, tem-se que o processo conseguiu um bloco válido para ser alocado, logo o mesmo acionará a estrutura de memória, mais especificamente seus vetores referentes aos kilobytes vazios e alocados, para atualizar seus valores no índice indicado. A primeira alteração indicará quanto do bloco de memória selecionado permanecerá não alocado. Já a segunda mudança pode ser descrita como de ação complementar à anterior, ou seja, o vetor receberá no índice selecionado o valor pertinente ao tamanho do processo.

O outro resultado possível na análise condicional tem a ver com a possibilidade do processo não ter encontrado um bloco de memória válido, ou seja, com a comparação inicial resultando em falso. A melhor escolha para se ter controle desses processos não alocados é através do uso da estrutura de dados fila, dessa maneira, os processos não alocados são adicionados um a um em uma fila de reservada a processos que não tiveram sucesso em encontrar um bloco vazio e com o tamanho suficiente para sua alocação.



```
1
2  int index = -1;
3
4
5
6
7  if (index != -1)
8  {
9      memory[index].empty = memory[index].empty - process_size;
10     memory[index].allocated = process_size;
11 }
12 else
13 {
14     to_queue(denied_process, process_size);
15 }
```

Figura 6: Trecho de código comum aos algoritmos de alocação

3.2.5 First fit

Na função de primeira alocação, tem-se quatro valores como parâmetros: (1) o endereço referente a estrutura da memória principal; (2) o endereço referente a estrutura de métricas de alocação; (3) a fila de processos negados; e, (4) o tamanho do processo que deseja-se alocar.

Em seu corpo, no que diz respeito especificamente ao comportamento do *first fit*, temos uma iteração que irá em seu pior cenário percorrer todos os blocos de memória em busca de um que contenha espaço suficiente para que o processo em questão seja alocado. Durante essa busca é realizada uma checagem para verificar se o bloco atual possui o

tamanho necessário para o processo, verificando a variável referente ao tamanho vazio (*empty*), seguidamente a verificação de já alocação ou não do bloco (*allocated == 0*).

Assim, caso a busca resulte em sucesso, ou seja, encontre dentro dos valores do tamanho da memória um bloco com tamanho grande o suficiente e não alocado, o processo é alocado naquele bloco, salva o valor do índice referente ao bloco e sai da iteração.



```
1  for (int i = 0; i < SIZE; i++) // here, n → number of processes
2  {
3
4      // allocated > 0 ?
5      if (memory[i].empty >= process_size && memory[i].allocated == 0)
6      {
7          index = i;
8          break; // go to the next process in the queue
9      }
10 }
```

Figura 7: Trecho de código do algoritmo first fit

3.2.6 Next fit

O algoritmo de next fit, dentre os algoritmos de alocação, é o único que necessita de uma parametragem a mais para seu funcionamento, sendo esse o parâmetro que determinará a partir de qual posição do bloco de memória o algoritmo deve continuar (*int *next_fit_index*).

Em sua codificação que lhe garante a particularidade de ser o algoritmo da próxima alocação, sua iteração tem o diferencial de a parametragem relativa ao início do laço começar pelo valor passado pela variável *int *next_fit_index*, que contém o índice da posição do último bloco alocado na memória. Como no restante dos algoritmos, a iteração responsável por percorrer a memória realiza suas repetições até o tamanho máximo da memória.

Internamente ao laço de repetição tem-se a presença de dois blocos de condicional, com o último deles possuindo uma ação resposta caso a condição resulte falsa. A primeira condicional verifica se o bloco atual é capaz de armazenar o tamanho do processo criado, além de verificar se o espaço vazio do bloco é suficiente para o processo, também é verificado se o bloco já foi alocado. Caso a condicional resulte verdadeira, o algoritmo executa três comandos: (1) armazenar o valor índice da posição do último bloco alocado; (2) armazenar o valor índice da posição do próximo bloco onde se deverá começar a análise de disponibilidade para alocação; e (3) encerrar a execução do algoritmo.

A segunda estrutura condicional é responsável por verificar se a variável *int *next_fit_index* foi incrementada de tal forma a armazenar um valor de bloco igual ao valor da última posição da memória. Em caso verdadeiro, atualiza-se a variável de modo que a mesma

recebe o valor de zero, permitindo assim que a busca de um espaço na memória para o próximo processo comece a partir do início do vetor de memória. Já caso o processo não consiga ser alocado e a variável com a posição do valor de índice por onde a busca pelo próximo bloco não seja de valor igual ao tamanho máximo permitido para alocação, o algoritmo apenas incrementa a variável **next_fit_index*.

A screenshot of a code editor with a dark background and light-colored text. The code is in C++ and implements the next fit memory allocation algorithm. It starts with a for loop that iterates over memory blocks from index 0 to SIZE-1. Inside the loop, it checks if the current block is empty and its size is greater than or equal to the process size. If so, it updates the next_fit_index to the current index and breaks the loop. If the next_fit_index reaches SIZE-1, it resets it to 0. Otherwise, it increments the next_fit_index by 1.

```
1  for (int i = *(next_fit_index); i < SIZE; i++) // here, n → number of processes
2  {
3
4      // allocated > 0 ?
5      if (memory[*(next_fit_index)].allocated == 0 && memory[*(next_fit_index)].empty >= process_size)
6      {
7          index = i;
8          *(next_fit_index) += 1;
9          break; // go to the next process in the queue
10     }
11
12     if (*(next_fit_index) == SIZE - 1)
13     {
14         *(next_fit_index) = 0;
15     }
16     else
17     {
18         *(next_fit_index) += 1;
19     }
20 }
```

Figura 8: Trecho de código do algoritmo next fit

3.2.7 Best fit

O algoritmo de melhor alocação possui a mesma passagem de parâmetros dos algoritmos *first fit* e *worst fit*, assim, não seremos redundantes aqui ao descrevê-los novamente.

Nos atentando apenas as peculiaridades do *best fit*, os blocos condicionais do algoritmo realiza duas checagens, primeiro se é a primeira vez que o algoritmo percorre a memória e encontra um bloco disponível e com capacidade de armazenar o processo, e segundo se, caso não seja o único bloco capaz de armazenar o processo, compara se os blocos seguintes que também possuem as condições devidas para armazenar o processo contém um espaço mais próximos ao tamanho do processo. Caso seja o primeiro bloco que respeite as restrições de alocação, o valor de índice da posição do processo já é logo salvo, do contrário o valor a ser salvo é o do índice com o valor disponível de memória o mais próximo do tamanho do processo.



```

1  for (int j = 0; j < SIZE; j++)
2  {
3      if (memory[j].allocated == 0 && (memory[j].empty ≥ process_size))
4      {
5          if (index == -1)
6          {
7              index = j;
8          }
9          else if (memory[j].empty < memory[index].empty)
10         {
11             index = j;
12         }
13     }
14 }

```

Figura 9: Trecho de código do algoritmo best fit

3.2.8 Worst fit

Com as descrições prévias dos algoritmos anteriores, foi possível antecipar neles alguns dos detalhes em comum com o algoritmo de pior alocação. Todavia, quanto ao que não foi dito e tem-se como característica principal do algoritmo *worst fit*, pode-se explicar como paralelo antagônico do algoritmo *best fit*. Ou seja, a segunda parte do bloco condicional do algoritmo da pior alocação tem como critério de escolha, caso haja mais de um bloco disponível que respeite as regras de alocação, o bloco de maior valor de espaço livre.



```

1  for (int j = 0; j < SIZE; j++)
2  {
3      if (memory[j].allocated == 0 && (memory[j].empty ≥ process_size))
4      {
5          if (index == -1)
6          {
7              index = j;
8          }
9          else if (memory[j].empty ≥ memory[index].empty)
10         {
11             index = j;
12         }
13     }
14 }

```

Figura 10: Trecho de código do algoritmo worst fit

3.2.9 Desalocação e integração da memória

Antes de entrarmos em detalhes quanto à funcionalidade de desalocação, primeiro é preciso entender alguns detalhes da lógica de desalocação. Toda vez que temos mudança de estado o processo deve ser desalocado, de execução para bloqueado ou pronto. Assim que o processo é desalocado é possível tentar aloca-lo novamente.

Para simular o que exposto acima, temos no algoritmo de desalocação um ponteiro para memória e o índice do processo como parametragem necessária, pois é preciso localizar o processo na tabela para identificar quais seus valores de memória que devem ser desalocados e quanto da memória deve ser restabelecida. Assim, como pode ser visto na figura 11, o algoritmo se certifica que a memória do processo tenha os mesmos valores vazios, tamanho do bloco, antes do processo ter sido alocado no mesmo. Por fim, marca-se o bloco de memória como disponível para alocação, com a *flag* de alocação da memória como zero.

A screenshot of a code editor with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. The code is written in C and defines a function named 'deallocate'. It takes two arguments: 'main_memory' (a pointer to an array of memory blocks) and 'index' (an integer representing the memory block index). The function body consists of five lines: a function signature, an opening curly brace, an assignment that adds the 'allocated' value of the memory block to its 'empty' value, an assignment that sets 'memory[index].allocated' to 0, and a closing curly brace.

```
1 void deallocate(main_memory *memory, int index)
2 {
3     memory[index].empty = memory[index].empty + memory[index].allocated;
4     memory[index].allocated = 0;
5 }
```

Figura 11: Código referente ao algoritmo de desalocação

No que se refere à integração da memória, a melhor escolha para se ter controle de processos não alocados é através do uso de estrutura de dados fila. Assim, quando na fila, os processos que foram previamente desalocados podem ser requisitados para alocação novamente. Dado as características da estrutura fila, o primeiro processo que foi negado será também o primeiro a ser usado nos algoritmos de alocação, e, após ser desenfileirado, caso tenha sua requisição para ser alocado novamente negada, ou o processo será adicionado ao final da fila de processos não alocados. Em outras palavras, o programa faz uso de uma lista de processos bloqueados, para que seja possível enfileirar os processos negados.

3.3 Métricas

Para melhor conhecermos o desempenho do programa foram utilizadas três métricas de desempenho, sendo elas: (1) o número médio de fragmentos externos; (2) o tempo médio de alocação em termos de número médio de nós percorridos na alocação; e, (3) o percentual de vezes que uma requisição de alocação é negada.

3.3.1 Fragmentos externos

Para que seja possível gerarmos os dados necessários para uma análise referente ao número de fragmentos externos gerados pelo simulador de gerência de memória, foi necessário a criação de uma estrutura de dado dedicada e uma função específica para esse objetivo.

Quanto à função para o cálculo médio de fragmentos externos, ela recebe como parametragem o tipo estruturado de dados de métricas, figura 12, para desse modo tornar possível

o acesso a lista destinada ao armazenamento das informações relativas aos fragmentos externos.

```
1 typedef struct
2 {
3     list external_fragmentation;
4     list nodes_traveled;
5     allocation_request allocation_request_metrics;
6 } metrics;
```

Figura 12: Estrutura de dados métricas

A lista supracitada, denominada no programa como *external_fragmentation*, é utilizada por todos os métodos aqui apresentados de alocação. Mais precisamente, ao acionar um algoritmo de alocação, independente de qual, o mesmo fará a contabilização de quanto do bloco utilizado para alocação foi preenchido com o processo e quanto do mesmo permaneceu vazio. Assim, com essa informação, é chamada a função de soma, *sum*, que irá percorrer toda a lista em busca dos valores relativos aos espaços de memória não ocupados pelos processos. Ademais, também faz-se uso da função de comprimento de lista para descobrir o número total de fragmentos gerados, denominada *lengtht*. Com esses dois valores conhecidos, passa-se para última etapa do algoritmo, o retorno da média entre a soma dos valores de todos os espaços de memória não aproveitados eficientemente pelo número total de blocos de memória.

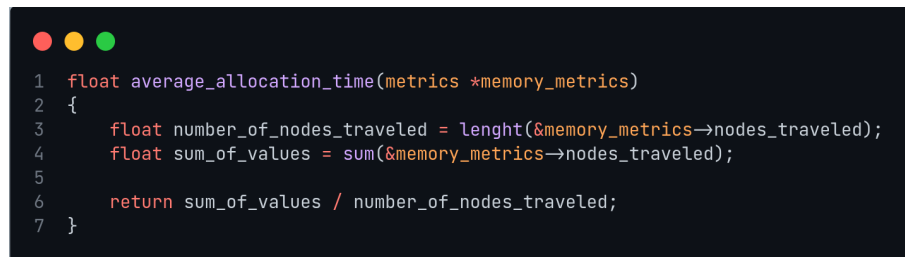
```
1 float average_number_of_external_fragments(metrics *memory_metrics)
2 {
3
4     float number_of_external_fragments = lengtht(&memory_metrics->external_fragmentation);
5     float sum_of_values = sum(&memory_metrics->external_fragmentation);
6
7     return (sum_of_values / number_of_external_fragments);
8 }
```

Figura 13: Função para encontrar a média de fragmentos externos

3.3.2 Percurso médio de nós para alocação

Quanto à métrica utilizada para medir o percurso médio de nós para alocação, a lógica não difere muito do que foi visto no algoritmo acima. Aqui, o algoritmo *average_allocation_time*, ou algoritmo de tempo médio de alocação, também é utilizado por todos os métodos de alocação apresentados neste trabalho.

Esta função possui como parâmetro a estrutura de dados metrics, que nos possibilita acessar a lista contendo o número de nós percorridos por cada processo alocado na memória. Com essa estrutura é possível resgatar duas informações necessárias para o objetivo do algoritmo, que é a soma do total de nós percorridos por todos os processos alocados, e a quantidade de blocos na qual a memória foi dividida. O resultado aqui esperado nada mais é que a divisão do valor total de nós percorridos pelo tamanho total de blocos da memória.

A screenshot of a code editor with a dark background and three colored circles (red, yellow, green) in the top left corner. The code is written in C and defines a function named average_allocation_time. It takes a pointer to a metrics structure as an argument. Inside the function, it calculates the number of nodes traveled and the sum of values from the metrics structure, and then returns the average value.

```
1 float average_allocation_time(metrics *memory_metrics)
2 {
3     float number_of_nodes_traveled = lenght(&memory_metrics->nodes_traveled);
4     float sum_of_values = sum(&memory_metrics->nodes_traveled);
5
6     return sum_of_values / number_of_nodes_traveled;
7 }
```

Figura 14: Função para encontrar a média do tempo de alocação

3.3.3 Percentual de alocação negada

A última métrica diz respeito ao percentual de vezes que uma requisição de alocação foi negada. Neste caso o processo fica bloqueado com uma flag de espera por memória, flag essa denominada admission. Caso haja uma liberação de memória por um processo, a alocação deste processo pode ser tentada novamente.

Em mais detalhes, quando um processo não é alocado ele entra nas métricas de percentual de vezes que uma requisição é negada. Para se obter o percentual total de alocações negadas durante a execução do programa, primeiro recupera-se o valor de número de requisições negadas, com seu valor multiplicado por 100, e em seguida divide-se pelo total de requisições de alocação no programa.

A screenshot of a code editor with a dark background and three colored circles (red, yellow, green) in the top left corner. The code is written in C and defines a function named percentage_of_allocation_request_is_denied. It takes a pointer to a metrics structure as an argument. Inside the function, it calculates the percentage of allocation requests that were denied by multiplying the denied count by 100 and then dividing by the total count.

```
1 float percentage_of_allocation_request_is_denied(metrics *memory_metrics)
2 {
3     float value = (memory_metrics->allocation_request_metrics.denied * 100);
4     return (value / memory_metrics->allocation_request_metrics.total);
5 }
```

Figura 15: Função para encontrar o percentual de alocação negada

3.4 Memória Virtual

Primeiramente, pode-se dizer que a memória virtual é um esquema de alocação de armazenamento no qual a memória secundária pode ser endereçada como se fosse parte da memória principal. Os endereços que um programa pode usar para fazer referência à memória são diferenciados dos endereços que o sistema de memória usa para identificar locais de armazenamento físico, e os endereços gerados pelo programa são traduzidos automaticamente para os endereços de máquina correspondentes.

Dessa maneira, vale dizer que foi utilizado esse conceito em nosso projeto para realizarmos a tarefa B do trabalho prático, neste caso, foi feito um endereçamento para cada bloco da memória principal com estes armazenados na memória virtual, sendo que após um bloco ter sido alocado por um processo seu endereço é traduzido para a memória física imediatamente e os blocos que não possuem processos alocados permanecem com seus endereços na memória virtual. Para isto, vale ressaltar que foi usada a ideia de paginação e **TLB** (*Translation Lookaside buffer*) [2].

Por fim, ressalta-se que foi utilizado uma tabela de páginas de 2^8 entradas, uma TLB de 16 entradas, com um tamanho de página e moldura de 2^8 bytes e uma memória física de 256 x 256 frames.

3.4.1 TLB em detalhes

A TLB atua como uma cache para o MMU que é usado para reduzir o tempo de acesso tirado da memória física. Dependendo da marca e modelo da CPU, pode-se existir mais de uma TLB, ou inclusive múltiplos níveis de TLB como acontece com a memória cache para evitar TLB misses e garantir uma latência de memória tão baixa quanto possível.

3.4.2 TLB hit

Quando o endereço de memória virtual chega, ele precisa ser traduzido para o endereço físico. O primeiro passo é sempre dissecar o endereço virtual em um número de página virtual e o deslocamento de página. Também chamado de *offset*, o deslocamento consiste dos últimos bits do endereço virtual. Os bits de deslocamento não são traduzidos e passados para o endereço de memória física. O *offset* contém bits que podem representar todos os endereços de memória em uma tabela de páginas.

3.4.3 TLB miss

Ainda assim, precisamos pensar em casos que o número da página virtual não é encontrado na **TLB**, nestes momentos, temos o que é chamado de **TLB miss**. Dessa maneira, nessas situações a **TLB** precisa consultar a memória global do sistema para entender qual o número da página física é usado. Alcançar a memória física significa maior

latência quando comparado com com uma **TLB hit**. Se a **TLB** está cheia e a **TLB miss** ocorre, a entrada menos recente da **TLB** é liberada e a nova entrada é colocada ao invés dela.

3.4.4 Paginação

Continuamente, temos o conceito de paginação, que se trata de um mecanismo de armazenamento usado para recuperar processos do armazenamento secundário para a memória principal na forma de páginas. Assim como temos a ideia de dividir cada processo em forma de páginas, vale dizer que a memória principal também será dividida em molduras.

Ademais, uma página do processo deve ser armazenada em um dos quadros da memória. Além disso, observa-se que as páginas podem ser armazenadas em diferentes locais da memória, mas a prioridade é sempre encontrar as molduras contíguas.

Outrossim, vale dizer que as páginas do processo são trazidas para a memória principal apenas quando são necessárias, caso contrário, elas residem no armazenamento secundário.

Desse modo, nota-se que diferentes sistemas operacionais definem tamanhos específicos de molduras, entretanto o tamanho de cada moldura precisa ser igual. Considerando o fato de que as páginas são mapeadas para os quadros em paginação, o tamanho da página precisa ser igual ao tamanho da moldura[3].

3.5 Impressão

As principais modificações no que diz respeito aos processos de impressão foi a criação da funcionalidade para a memória principal, sendo que cada algoritmo de alocação terá seu próprio processo de impressão para memória conforme pode ser visto na imagem abaixo para o algoritmo *first fit*.

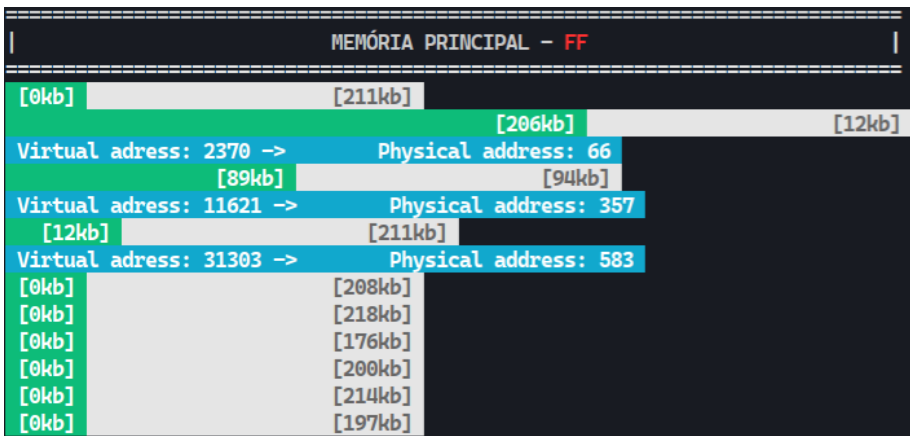


Figura 16: Memória Principal e Virtual - FF

Além disso, foi implementado a funcionalidade de impressão para os parâmetros de performance, sendo eles: (1) número médio de fragmentos externos, (2) tempo médio de

alocação e (3) percentual de alocação negada.

PERFORMANCE PARAMETERS - FF			
(1)	(2)	(3)	
1702.00	2.50	0.00%	
LEGEND			
(1)	- AVERAGE NUMBER OF EXTERNAL FRAGMENTS		
(2)	- AVERAGE ALLOCATION TIME		
(3)	- PERCENTAGE OF THE ALLOCATION REQUEST IS DENIED		

Figura 17: Parâmetros de performance

Por fim, foram feitas outras funcionalidades de impressão auxiliares, sendo algumas delas: menu para escolha dos algoritmos de alocação, processos negados em cada algoritmo de alocação e tamanho de um novo processo a ser alocado.

3.6 Adaptações necessárias

Assim como esperado, foi necessário adaptar a estrutura de gerenciamento para este trabalho. A principal alteração deu-se na estrutura de dados *process_table* onde foi preciso adicionar um array de inteiros (*memory_index*), isso porque, foi necessário cada algoritmo de alocação ter sua própria memória. Com isso dito, um processo pode ser salvo em várias memórias, e para isso é necessário que possamos recuperar os índices desse processo em cada uma das memórias reservadas aos diferentes algoritmos de alocação. Por convenção a posição [0] é referente ao *first fit*, a posição [1] é referente ao *next fit*, a posição [2] é referente ao *best fit* e a posição [3] é referente ao *worst fit*.

Além disso, caso um processo não consiga ser alocado na memória o seu estado será definido como em **admissão** e para isso, foi inserido no *enum* de estados o campo **ADMISSION**.

3.7 Domínio

O termo *Domain-Driven Design* (DDD) foi criado por **Eric Evans** em seu livro intitulado “*Domain-Driven Design: Tackling Complexity in the Heart of Software*”, onde ele fala extensivamente sobre esses princípios e como eles podem ser aplicados e suportados em código, mostrando diversos padrões de projeto de forma detalhada e suas ideias

para um melhor desenvolvimento de software após duas décadas percebendo os mesmos problemas [4].

A camada de Domínio é a camada foco do **DDD** já que é o coração do negócio em que você está trabalhando. É baseado em um conjunto de ideias, conhecimentos e processos de negócio. É a razão do negócio existir. Sem o domínio de todo o sistema, todos os processos auxiliares, não servirão para nada.

Por mais que tenhamos tais definições, quais os benefícios e utilidades para este projeto? A camada de Domínio será responsável pela integração entre o Gerenciador e toda a regra de utilização para os algoritmos de alocação: quais algoritmos utilizar, quando utilizá-los, gerenciar processos de desalocação, dentre outras finalidades.

4 Discussão dos resultados

Por fim, ressalta-se que após a execução do programa foi possível coletar os dados e indicadores de desempenho para cada técnica, sendo os indicadores: número médio de fragmentos externos; tempo médio de alocação em termos de número médio de nós percorridos na alocação; e, o percentual de vezes que uma requisição de alocação é negada.

Desta forma, foram coletados os dados após três execuções onde obtivemos os seguintes resultados:

	(1)	(2)	(3)
<u>first fit</u>	1865.83	1.50	33.33%
<u>next fit</u>	1865.83	1.50	33.33%
<u>best fit</u>	1865.83	4.25	33.33%
<u>worst fit</u>	1865.83	4.75	33.33%

Tabela 1: Tabela da primeira execução

	(1)	(2)	(3)
<u>first fit</u>	1848.00	1.00	50.00%
<u>next fit</u>	1848.00	1.00	50.00%
<u>best fit</u>	1848.00	6.33	50.00%
<u>worst fit</u>	1848.00	5.00	50.00%

Tabela 2: Tabela da segunda execução

	(1)	(2)	(3)
<u>first fit</u>	1730.17	2.00	16.67%
<u>next fit</u>	1730.17	2.00	16.67%
<u>best fit</u>	1730.17	3.20	16.67%
<u>worst fit</u>	1730.17	5.80	16.67%

Tabela 3: Tabela da terceira execução

A partir da tabela da primeira execução nota-se que os algoritmos *first fit* e *next fit* foram os que apresentaram os melhores resultados, seguidos do *best fit* e *worst fit*, respectivamente. Por fim, observando a segunda e terceira tabela, pôde-se notar que o ranqueamento dos algoritmos não mudou em ambos os casos, mesmo com a mudança de resultados nas três métricas coletadas.

5 Conclusão

Posto isso, o grupo pôde enxergar de forma mais clara todo o funcionamento de um sistema de gerenciamento de memória de um sistema operacional, bem como ponderar sobre suas aplicações, além de expandir a ambientação com a linguagem C e o sistema operacional Linux.

Quanto aos resultados obtidos pelas métricas propostas neste trabalho, nos surpreendemos com os valores obtidos de fragmentação externa, que nos deixou a indagar quais seriam as melhorias que poderiam ainda ser feitas no programa.

Por conseguinte podemos ressaltar que o material apresentado na disciplina foi de suma importância para o desenvolvimento do projeto, haja vista que as explicações dadas pelo professor nos ajudaram bastante a entender as etapas a serem executadas, na construção dos códigos e disponibilização de alguns sites para serem sanadas certas dúvidas acerca da simulação de gerenciamento de memória.

Em adição, é válido dizer que apesar das dificuldades na implementação do código, o trio foi capaz de superar e corrigir quaisquer erros no desenvolvimento do algoritmo. Por fim, verificou-se a assertiva para o objetivo do projeto em implementar uma simulação de gerenciamento de memória.

References

- [1] Dat. <https://maisgeek.com/o-que-e-um-arquivo-dat-e-como-faco-para-abrir-um/>. (Accessed on 01/06/2022).
- [2] Virtual-physical. <https://blogs.vmware.com/vsphere/2020/03/how-is-virtual-memory-translated-to-physical-memory.html>. (Accessed on 01/06/2022).
- [3] Paginacao. <https://www.javatpoint.com/os-paging-with-example#:~:text=In%20operating%20Systems%2C%20Paging%20is,in%20the%20form%20of%20frames>. (Accessed on 01/06/2022).
- [4] Virtual-physical. <https://www.devmedia.com.br/ddd-domain-driven-design-com-net/14416#:~:text=0%20termo%20Domain%2DDriven%20Design,de%20forma%20detalhada%20e%20suas>. (Accessed on 15/07/2022).