



UNIVERSIDADE FEDERAL DO PAMPA - UNIPAMPA
Curso Bacharelado em Engenharia de Computação
Disciplina de Engenharia de Software

PROFESSOR: Carlos Michel Betemps

DOCUMENTAÇÃO DE UM PRODUTO DE SOFTWARE
Gerenciador de Memória

Giuliana Oliveira de Mattos Leon
Guilherme Domingues Goulart
João Antonio dos Santos Antunes
Thamires Sampaio de Mattos

Bagé, 4 de outubro de 2021

ÍNDICE DETALHADO

PREFÁCIO.....	3
1. INTRODUÇÃO DO DOCUMENTO.....	4
1.1. MÉTODO DE TRABALHO.....	4
2. DESCRIÇÃO GERAL DO PROJETO.....	7
3. REQUISITOS DO SISTEMA.....	8
3.1. REQUISITOS FUNCIONAIS.....	8
3.2. REQUISITOS NÃO FUNCIONAIS.....	9
3.3. PROTÓTIPO.....	10
4. ANÁLISE E DESIGN.....	12
4.1. DIAGRAMA DE CLASSES.....	12
5. IMPLEMENTAÇÃO.....	14
6. TESTES.....	19
7. MANUAL DO USUÁRIO.....	35

Prefácio

O objetivo deste documento é fornecer todas as informações necessárias sobre o desenvolvimento do software de gerenciamento de memória, utilizando os princípios da disciplina de engenharia de software, como a notação UML (Unified Modeling Language), os modelos de desenvolvimento de software, verificação e validação, etc.

Neste documento será citado o modelo de desenvolvimento, bem como a UML da aplicação, os requisitos do sistema, a descrição geral do problema, análise e design, implementação, os planos de testes e um pequeno manual do usuário.

Ao final deste documento, haverá as conclusões e considerações finais sobre o projeto.

1. Introdução

Em sistemas operacionais, um gerenciador de memória é a parte responsável por cuidar de quais partes da memória estão em uso, quais estão livres, alocar processos quando necessário, desalocar processos quando eles não necessitarem mais e gerenciar a troca dos processos entre a memória principal e o disco, que acontece quando a memória principal não é suficiente para manter todos os processos.

Desta maneira, foi desenvolvido um *software* capaz de exemplificar como é o funcionamento de um gerenciador, por meio de alocação e desalocação, tanto da forma sequencial, como da forma paralela. Este software é capaz de pegar uma requisição do vetor de requisições, que é uma fila circular, alocar na heap e fazer a desalocação assim que a heap estiver com sua ocupação acima de uma determinada porcentagem definida pelo usuário, além de ser capaz de realizar compactação na heap caso o nível de fragmentação externa esteja crítico. O usuário será capaz de estabelecer valores para o tamanho da heap, o percentual máximo e mínimo da heap, a quantidade de requisições atendidas e o tamanho máximo e mínimo da variável dinâmica delas. Ao final, será possível ver o tempo de execução da versão sequencial e da versão paralela.

1.1. Método de Trabalho

Para a elaboração deste projeto, foi utilizado o modelo de metodologia ágil. Esta metodologia de desenvolvimento de software tem como foco o próprio projeto, visando sempre melhorias e alterações constantes, baseadas em *feedbacks* dos usuários, clientes e dos próprios desenvolvedores como também, a flexibilidade de escopo do projeto. Juntamente com esta metodologia, foi empregado o método *scrum* que tem como objetivo agregar mais produtividade nos processos. Desta maneira, foram elaboradas etapas a serem desenvolvidas, podendo ocorrer tráfego entre elas.

Em um primeiro momento, foi feito o levantamento dos requisitos necessários para que a aplicação se tornasse funcional para o cliente. Desta maneira, foi feito um levantamento de todos os requisitos com o cliente, sendo eles listados abaixo:

- O algoritmo deverá acessar as requisições para alocação de memória a partir de um vetor em memória (vetor de requisições), implementado como uma fila circular. Cada requisição deve informar ao algoritmo de alocação de memória o tamanho da variável dinâmica a ser alocada, bem como seu respectivo identificador. Um gerador de requisições randômicas deve alimentar o vetor de requisições para que este sempre disponha de requisições suficientes para possibilitar o trabalho em fluxo contínuo do algoritmo de alocação.
- O alocador de memória e o gerador de requisições randômicas devem compor, portanto, duas threads do sistema (uma thread é o alocador de memória e outra thread é o gerador de requisições) e ambos operam de forma paralelo/concorrente sobre o vetor de requisições. Fica a critério de cada equipe optar, para melhora de eficiência e de qualidade de sua solução, por implementar esses dois módulos (alocador e gerador) com mais threads cada um.
- O usuário do sistema deve ser capaz de configurar o tamanho da heap¹ para funcionamento do sistema, assim como o intervalo de valores (mínimo e máximo) para geração randômica do tamanho das variáveis a cada requisição produzida pelo gerador.
- O algoritmo de alocação deve alocar a memória para cada variável na heap sempre de forma contígua.
- O usuário também deve informar o limite de ocupação de memória como um percentual, acima do qual o algoritmo de desalocação de memória deve ser executado. Quando o algoritmo de desalocação for executado, ele deve liberar memória até que o percentual de memória livre fique igual ou abaixo de um outro valor de limitar (também definido pelo usuário).
- O algoritmo de desalocação deve selecionar de forma randômica variáveis

¹ Área na memória a ser usada para alocação das variáveis dinâmicas, cujos tamanhos são informados nas requisições contidas no vetor de requisições.

a serem desalocadas (simulando um fluxo natural de um sistema real com desalocações).

- O algoritmo de desalocação deve executar de forma paralela com o alocador (isto é, as alocações devem continuar ocorrendo durante o tempo de reciclagem da memória).
- O número de variáveis a serem alocadas na memória deve ser informado pelo usuário do sistema, de modo a calibrar o esforço a ser realizado pelo algoritmo alocação e, por conseguinte, o tempo de execução para todo o sistema.
- Fica a critério de cada equipe, implementar um algoritmo de compactação para eliminar a fragmentação externa que se forma à medida que as variáveis vão sendo desalocadas.

Após a definição dos requisitos com o *product owner*, passamos para o planejamento das *sprints* do projeto. Esse planejamento deu-se na forma de criação de uma estrutura chamada *Scrum*, sendo essa uma estrutura para implementar processos ágeis em desenvolvimento de *softwares*. Este *framework* consiste em interações curtas de trabalho, sprints e reuniões diárias (*Daily Scrum*) até a finalização do projeto. Com a definição dos *Scrums Masters*, que são os líderes do projeto em cada *sprint*, foi definido um cronograma de atividades definido abaixo, na figura 1.

1	SPRINT 1 - Elaboração do Projeto (SM Thamires)	<ul style="list-style-type: none"> - Backlog do produto - Consulta com Cliente - Definição dos Scrums 	ALTA
2	SPRINT 2 - Definição e Plano do Projeto (SM Giuliana)	<ul style="list-style-type: none"> - Reuniões de Equipe - Delimitação do Problema - Organização do Trabalho - Implementação Back-end (UML) - Revisão da documentação 	ALTA
3	SPRINT 3 - Execução do projeto (SM Guilherme)	<ul style="list-style-type: none"> - Reuniões de Equipe - Implementação Back-end - Teste de Aplicação - Revisão da documentação 	ALTA
4	SPRINT 4 - Controle e desempenho do projeto (SM João)	<ul style="list-style-type: none"> - Reuniões de Equipe - Implementação Back-end - Teste de Aplicação - Revisão da documentação 	ALTA
5	SPRINT 5 - Interface e fechamento do projeto (SM Thamires)	<ul style="list-style-type: none"> - Implementação Front-end - Testes Finais - Relatório 	ALTA

Figura 1 - Cronograma de atividades de cada sprint

Com a elaboração do cronograma de atividades, o próximo passo foi o desenvolvimento de uma UML do programa, que contemplasse todos os requisitos necessários para que o projeto atendesse às necessidades do cliente. Conforme o modelo de metodologia ágil, foi possível alterar a UML a cada modificação da aplicação. Foi elaborado um plano de testes que começou a ser utilizado na *sprint* 3. Um plano de testes trata-se de um documento ao qual se definem escopos e objetivos, além dos requisitos da aplicação. Este plano foi feito com base nos requisitos citados anteriormente e dividido em tabela de requisitos, detalhamento da abordagem de testes, ferramentas, casos de teste e considerações dos resultados.

2. Descrição Geral do Sistema

O projeto tem como propósito uma descrição do funcionamento de um gerenciador de memória dinâmico, fazendo alocação e desalocação sequencialmente e paralelamente, de acordo com a necessidade do usuário. Este sistema deverá gerar requisições e armazená-las em um vetor de requisições, as configurações das requisições como quantidade e tamanho deverão ser informadas pelo usuário. Após, ocorrerá a remoção da primeira requisição da fila deste vetor e a alocação da mesma na heap, na qual o tamanho também será definido pelo usuário. Quando a heap atingir um percentual máximo de espaço alocado, definido também pelo usuário, haverá a desalocação da mesma até chegar a um percentual mínimo, também informado pelo usuário. Se o nível de fragmentação externa ficar crítico, o gerenciador executará uma compactação dos dados, realocando os espaços vazios (buracos) e os espaços ocupados, para que assim a fragmentação externa seja controlada e a heap possua apenas um buraco contíguo de espaços vazios no sistema. É importante salientar que podem ocorrer duas formas deste gerenciamento: sequencial e paralelo. Na forma paralela, é utilizado semáforos para o sincronismo e threads para a programação multithreading.

3. Requisitos do Sistema

No decorrer de desenvolvimento desta aplicação, foi reunido todas informações possíveis para garantir que o projeto atenda exatamente ao que é esperado. Dessa forma, foi definido os requisitos do sistema, estes divididos em requisitos funcionais e requisitos não funcionais.

3.1. Requisitos Funcionais

Nesta seção, serão abordados os requisitos funcionais da aplicação. Requisitos funcionais são todos os problemas e necessidades que devem ser atendidas e resolvidas pelo software por meio de funções ou serviços. Desta maneira, foram listados os seguintes requisitos funcionais:

- 1) O usuário do sistema deve ser capaz de configurar o tamanho do heap para funcionamento do sistema.

- 2) O usuário do sistema deve ser capaz de configurar o intervalo de valores (mínimo e máximo) para geração randômica do tamanho das variáveis a cada requisição produzida pelo gerador.
- 3) O usuário do sistema deve ser capaz de definir o valor de referência do índice de fragmentação externa do sistema.
- 4) O algoritmo de alocação deve alocar a memória para cada variável na heap sempre de forma contígua.
- 5) O algoritmo de alocação deve monitorar o índice de fragmentação externa do sistema e iniciar a execução de um desalocador de memória para liberar espaço na heap **sempre** que o índice de fragmentação estiver acima do valor de referência.
- 6) O algoritmo de desalocação deve selecionar, de forma randômica, variáveis a serem desalocadas (simulando um fluxo natural de um sistema real com desalocações).
- 7) O algoritmo de desalocação deve ser executado de forma paralela com o alocador.
- 8) O número de requisições de alocação a serem alocadas na memória deve ser informado pelo usuário do sistema.
- 9) O usuário deve informar o limite de ocupação de memória como um percentual, acima do qual o algoritmo de desalocação de memória deve ser executado.
- 10) O usuário deve informar o percentual mínimo de espaço livre na memória, é trabalho do desalocador liberar memória até que o percentual de memória livre fique igual ou abaixo deste valor.

3.2. Requisitos Não-Funcionais

Os requisitos não funcionais são todos aqueles relacionados à forma como o software se comportará com relação a desempenho, usabilidade, confiabilidade, segurança, disponibilidade, manutenção e tecnologias envolvidas. Elas são características mínimas de um software, ficando então por conta do desenvolvedor optar ou não pelo uso destes requisitos.

- 1) O sistema deverá ser desenvolvido na linguagem Java.

- 2) É necessário o uso de threads para a execução sequencial.
- 3) É necessário que o alocador funcione de forma sincronizada com o desalocador.
- 4) A aplicação deve rodar em qualquer máquina.
- 5) O sistema deverá processar n requisições em menor tempo possível.
- 6) Os usuários deverão ler o Manual do Usuário, disponível na seção 7, para executar a aplicação.

3.3. Protótipo

Neste item será apresentado o protótipo da aplicação, consistindo na interface preliminar contendo seu subconjunto de funcionalidades. A interface final foi desenvolvida na linguagem Java, pela ferramenta Eclipse. Abaixo, na figura 2, encontra-se o seu protótipo, desenvolvido no site Lucidchart, juntamente com o objetivo da tela.

GERENCIADOR DE MEMÓRIA

Configurações Heap:

Tamanho da Heap

Percentual Máximo

Percentual Mínimo

Configurações Vetor Requisição:

Número de Requisições

Configurações Requisição:

Tamanho Mínimo

Tamanho Máximo

Informações Extras:

Ver Heap

Ver Ocupados

Ver Vetor de Requisições

Executar em modo:

Paralelo

Sequencial

Adicionar Dados

Figura 2 - Modelo da interface gráfica

Nas figuras abaixo, será especificado as instruções de cada elemento contido na interface gráfica.

Instruções JTextField

JTextField1: Armazenará a informação do tamanho da heap.

JTextField2: Armazenará a informação de percentual máximo de ocupação da heap.

JTextField3: Armazenará a informação de percentual mínimo de ocupação da heap.

JTextField4: Armazenará a informação de quantas requisições serão atendidas ao total.

JTextField5: Armazenará o tamanho mínimo da variável a ser alocada de cada requisição.

JTextField6: Armazenará o tamanho máximo da variável a ser alocada de cada requisição.

Figura 3 - Instruções dos textos inseridos

Instruções JButton

JButton1: Pegará as informações dadas pelo usuário e as aloca em suas respectivas variáveis. OBS: JButton1 apenas funcionará se o usuário preencher todos os campos e o tamanho máximo da requisição não for maior que o tamanho da heap.

JButton2: Irá imprimir as informações da heap, mais especificamente seus 0's e 1's (0 nas posições livres e 1 nas posições ocupadas).

JButton3: Irá imprimir o vetor de ocupados, organizado por ordem decrescente e mostrando as seguintes informações: Início (endereço da página que ele iniciou a alocação), tamanho (tamanho da variável alocada, quantos blocos ela está ocupando) e ID (identificador da requisição alocada).

JButton4: Irá imprimir o vetor de requisições.

JButton5: Executará o programa em modo paralelo, pode executar apenas se JButton1 tiver sido utilizado de forma correta ao menos 1 vez.

JButton5: Executará o programa em modo sequencial, pode executar apenas se JButton1 tiver sido utilizado de forma correta ao menos 1 vez.

Figura 4 - Instruções dos botões contidos na tela

4. Análise e Design

De acordo com os requisitos levantados, foi possível levantar 2 tipos de diagramas para modelagem da solução do sistema: Diagrama de Classe e Diagrama de Sequência. Cada um desses diagramas expõem partes essenciais da aplicação.

4.1. Diagrama de Classes

Neste item será apresentado o modelo UML (Unified Modeling Language) do diagrama de classes. Este diagrama, figura 3, foi utilizado para a elaboração da estrutura do projeto.

A classe Interface possui um gerador de requisições, então ela gera e insere requisições no vetor de requisição no começo do programa. Ela também possui objetos do tipo semáforo e objetos do tipo heap sequencial, além de incluir algumas informações nessas classes, como percentual máximo e mínimo de ocupação da heap.

No vetor de requisições, será setado os valores de tamanho do vetor baseado no número de requisições que o usuário informou.

O alocador remove uma requisição do vetor de requisições, utilizando semáforos, para alocar na heap e colocar as informações de identificador e tamanho na fila de segmentos ocupados. A classe segmentos possui atributos de início, tamanho, configuração e identificador. Essa classe é utilizada para representar cada requisição, pois cada requisição possui um identificador, um tamanho e sua locação inicia de alguma determinada posição.

A classe semáforo é utilizada para fazer o sincronismo entre o alocador e o desalocador da versão paralela. Ele é responsável por criar uma fila de livres e ocupados na classe segmentos. O semáforo possui métodos para imprimir a fila de ocupados, a fila de livres, verificar se já chegou ao final da fila e de checagem de tempo.

Já na classe heap sequencial, ela é a parte usada para a execução sequencial da aplicação. Então ela possui atributos para percentuais máximo e mínimo, tamanho total do vetor e o seu próprio vetor de heap. Essa classe possui métodos para inserir elementos na heap, criando um novo objeto do tipo segmento e adicionando na fila de ocupados, métodos de imprimir estes segmentos ocupados e livres, a função de compactar quando há buracos na memória e a fragmentação externa está crítica, e a função que soma estes buracos na memória. Mais detalhes sobre cada classe, são encontradas na seção 5 deste artigo.

Resumindo a UML: Tudo começa pela interface, ela recebe as informações iniciais do usuário como tamanho da heap, percentual mínimo e máximo de ocupação, número de requisições e tamanho mínimo e máximo de cada requisição, após isso ela distribui esses dados para todas as classes, exceto alocador e desalocador, além disso ela gera as requisições e as envia para o vetor de requisições. Quando o usuário seleciona o modo de execução em paralelo, o alocador remove a primeira requisição da fila do vetor de requisições e realiza a alocação na heap da versão paralela chamando o método inserir, além de reorganizar as listas de livre e ocupados do tipo segmento, já que agora um ou mais blocos estão ocupados. Quando o gerenciador percebe que a nova requisição retirada para alocação possui uma variável de tamanho maior que o maior buraco da heap, nós possuímos 2 caminhos: Inicialmente chamamos o método somatório para ele nos retornar o valor do somatório de todos os buracos existentes na heap, caso este valor retornado seja maior que o tamanho da variável que queremos alocar na memória, chamamos o método compacta para realizar a compactação e após isso iremos chamar o método inserir normalmente. Caso o somatório nos retorne um valor que seja menor que o tamanho da variável que queremos alocar na memória, o único caminho que nos resta é realizar a desalocação, então liberamos o semáforo do Desalocador para ser possível desalocar. Outra forma do Desalocador ser chamado é se a porcentagem de espaço ocupado na memória for igual ou maior que a porcentagem máxima definida pelo usuário, nos dois casos da desalocação ela é realizada até que a porcentagem de espaço ocupado na memória seja igual ou menor que a porcentagem mínima definida. Quando o usuário seleciona a versão sequencial, a execução quase toda acontece na classe heap sequencial com um looping de inserção na classe interface. Na sequencial não

possuímos threads nem semáforos, todos os comandos de somatório, inserção, desalocar e compactar estão dentro da classe heap sequencial, por conta disso que ela possui ligação na UML apenas com a interface que é quem envia as informações iniciais do usuário para ela e com os segmentos pois ela precisa manter as listas de livres e ocupados da mesma forma que a sequencial faz.

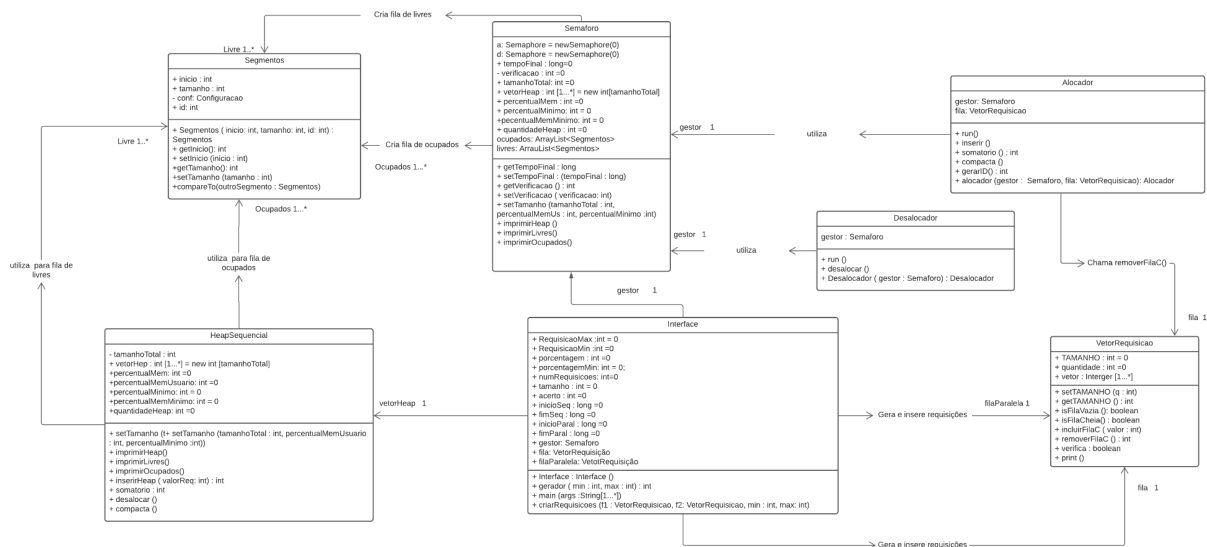


Figura 3 - UML

5. Implementação

Nesta seção será abordado sobre a implementação das classes esboçadas na UML. As classes são as seguintes: alocador, desalocador, heap sequencial, segmentos, semáforo, vetor de requisição e interface.

Antes de começar a explicar o funcionamento de cada classe, vale ressaltar algumas informações. Como a memória implementada é contígua, foi necessário trabalhar com os segmentos de tamanho variáveis. Ou seja, ele vai alocar na memória exatamente o tamanho que ele precisa. Com isso, ficamos com o que é chamado de buracos. Quando a memória fica com esses espaços vazios, eles podem ter diferentes tipos de tamanho, dependendo do tamanho do processo que sai (swap out) e dos processos que entram, dessa maneira, para não ocorrer o

desperdício de memória por fragmentação externa, foi utilizado a compactação de memória, onde passamos por dentro dessa memória, somando todos os espaços vazios, os realocando e assim juntando todos os buracos em um segmento contíguo, para que assim seja possível armazenar requisições que anteriormente não podiam se alocadas, já que o segmento que se formará irá possuir tamanho igual ao somatório de todos os buracos existentes anteriores a compactação. Tendo essas informações ditas, é possível começar a falar de cada classe separadamente.

- 1) Interface: Ela contempla o início do programa. Na interface do usuário, é pedido para o usuário inserir as configurações da heap como tamanho e percentual máximo e mínimo de ocupação, as configurações do vetor de requisição e as configurações da requisição. Após inseridos os valores, é apenas necessário apertar o botão adicionar dados para que os dados sejam inseridos e gerados em seus devidos lugares. Depois de inserir os dados, é necessário escolher a forma de como será executado o gerenciador. É possível também ver o vetor da heap, ver o vetor de ocupados e ver o vetor de requisições. Além disso, ela possui métodos de gerar números para as requisições, método de manipulação de fila, as ações de cada botão presente na interface gráfica e todas as estruturas utilizadas para o pleno funcionamento da aplicação. Dentre essas estruturas, se encontram a estrutura para o semáforo, a estrutura para heap, os vetores de requisições, um para cada tipo (sequencial e paralela), variáveis para requisições e variáveis para medir o tempo. Com relação às requisições, elas são geradas de forma aleatória na classe da interface, como já dito. Quando setado o valor de número de requisições, é chamada a função criarRequisições e passado como parâmetro as 2 filas (paralela e sequencial) e os valores de mínimo e máximo, dessa maneira iremos ter duas filas com os mesmos valores. Na função criarRequisições, é onde ocorre a manipulação da mesma, ou seja, são gerados valores randômicos, com base da função gerador. Quando selecionado o botão de sequencial, é iniciado o tempo para o cálculo de desempenho de cada versão. Então, enquanto tiver elementos no vetor de requisição, eles serão retirados dele e inseridos na heap. Quando não existir mais elementos no vetor de requisição, é informado o valor em

milissegundos da execução sequencial. Quando selecionado o botão de paralela, é iniciado o tempo para o cálculo de desempenho. É instanciado duas threads, uma para o alocador e outra para o desalocador. Então enquanto estiver ocorrendo a inserção lá na classe Alocador, o nosso tempo segue passando. Logo mais, será explicado o funcionamento da classe alocador.

- 2) Vetor de Requisição: Na classe vetor de requisição, temos a nossa fila circular. Nela serão estabelecidos valores como o tamanho do vetor e a quantidade de elementos no vetor. Nessa classe, possuímos métodos para verificar se a fila está vazia, para verificar se a fila está cheia, para printar e para inserir e remover elementos da fila. No método inserção, primeiro é feito um teste para saber se a fila está cheia, após isso, ele vai rodando o vetor até encontrar uma posição livre para inserir. No método remover, caso a fila não esteja vazia, é criada uma fila nova que receberá os dados do antigo vetor menos o elemento inserido, por isso o tamanho dessa fila nova é -1. Também possuímos métodos para impressão do vetor.
- 3) Segmentos: Na classe segmentos, é onde ficam as informações de início (posição na memória), tamanho e identificador de cada requisição alocada na heap. Essa classe possui dois propósitos, sendo eles: manter os dados de cada processo alocado na heap e servir como base para criarmos listas dos espaços livres e espaços ocupados da heap.
- 4) Semáforo: A classe semáforo é utilizada para o controle de concorrência da aplicação. O semáforo limita o acesso de usuários que querem acessar simultaneamente certos recursos protegidos. Nesta classe, definimos 2 objetos do tipo semáforo, um para o alocador e outro para o desalocador (eles são setados como 0 para que tudo que esteja no *acquire* fique retido até o *release*, por exemplo, se o semáforo fosse em 1, a primeira chamada de *acquire* vai acontecer e o resto será bloqueada até que a primeira seja bloqueada, já com zero é o contrário), além de atributos para verificação, tempo final de alocação, um vetor que será preenchido com as requisições, o

percentual da memória, a quantidade de elementos na heap e 2 *array list* para listar em ordem decrescente os espaços ocupados e livres dela. Além destes atributos, os métodos mais importantes que compõem essa classe são o `setTamanho`, que cria um vetor de tamanho e zera todas as posições desse vetor. Depois disso, ele cria um objeto do tipo `segmento` que recebe o valor total da requisição e adiciona esse novo segmento à fila de livres. É feito também o cálculo de percentual de memória, com base no percentual que foi informado pelo usuário. Os métodos de impressão servem apenas para imprimir os segmentos livres, os segmentos ocupados e a impressão da heap.

- 5) Alocador: A classe alocador utiliza a interface *Runnable* para a criação de threads. No método `inserir`, a primeira coisa que é feita é testar se ainda há elementos no vetor de requisição, se ainda houver, é removido uma requisição do vetor de requisições e passa para a próxima etapa, que é verificado se a quantidade de elementos contidos na heap é maior que o percentual máximo de elementos que eu posso ter nela. Caso a quantidade de elementos contidos na heap realmente seja maior que o percentual máximo definido pelo usuário, o gestor libera o recurso de desalocar do semáforo, assim realizando desalocações até que o percentual de espaços ocupados da memória seja menor ou igual ao mínimo informado pelo usuário. Logo após isso, é salvo o segmento livre de maior espaço na variável `s` que nesse caso seria o segmento que está na posição 0 já que a lista de livres está ordenada por ordem decrescente de tamanho. O método de alocação é o *worst-fit*, ou seja, escolhe a partição que deixa o maior espaço sem utilização. Após salvar o maior segmento livre na variável `s`, é testado se o tamanho da requisição é maior do que o maior buraco disponível, se a resposta for sim, é realizado um somatório do tamanho de todos os buracos disponíveis para ver se compactação seria a solução, se o resultado deste somatório retornar um valor maior que o tamanho informado pela requisição, é realizado a compactação, porém, se mesmo somando todos os espaços livres não é encontrado um valor igual ou maior que o solicitado pela requisição, a única coisa que resta fazer é desalocar. Após tudo isso, será

capaz inserir a requisição na heap, então é passado para a inserção, onde é criado um novo segmento, salvando as informações de posição inicial dela na heap, tamanho da variável e identificador, logo após isso é adicionado na lista de segmentos ocupados e finalmente inserido os dados nos blocos corretos, ao final deste processo também é calculado o novo tamanho disponível na heap, reorganizado a lista de espaços livres e resetado as variáveis de controle. Quanto ao somatório, basicamente é pego a lista de segmentos livres e soma o tamanho de todos os buracos para descobrir se o futuro segmento originado pela compactação irá conseguir armazenar o tamanho da última requisição solicitada. A função de compactação serve para compactar os buracos de memória que ficaram sem requisições. Então essa compactação ocorre enquanto não chegar no final da fila de tamanho total do heap. Enquanto isso estiver acontecendo, será percorrido toda fila de segmentos livres, e somado esses segmentos. Após somado, é percorrido todo o vetor de segmentos ocupados fazendo uma cópia de todas as posições ocupadas. Depois de percorrer cada objeto da lista de ocupados, vai ser inserido no início fila a requisição com o seu tamanho, setar a parte que ficou livre nesse vetor. Após andar em toda extensão de heap e verificar os espaços vazios e ocupados, é reordenado os valores livres no vetor de livres e os valores de ocupados no vetor de ocupados, a lista é ajustada para o seu novo tamanho, e um novo segmento livre é adicionado na fila com posição inicial, tamanho e seu identificador.

- 6) Desalocador: O desalocador é ativo quando a porcentagem de espaço ocupado na heap chega ao valor definido pelo usuário, inicialmente é dado um acquire no semáforo para solicitar acesso, após isso é pego o maior valor da lista de ocupados, e desaloca todo este segmento, realizando o controle de quantos elementos estão alocados na heap agora. Após isso, esse segmento é adicionado à fila de livres e removido da fila de ocupados, é realizado esse processo até a porcentagem de espaço ocupado na heap ser menor ou igual que o mínimo informado pelo usuário.

7) Heap Sequencial: Na heap é feita a declaração de todos os atributos necessários para codificação da mesma. Ela possui grande semelhança com a classe Alocador, tendo apenas a divergência de que enquanto na Alocador é utilizados semáforos para desalocar, aqui temos uma função própria para isso. Na heap é também setados os tamanho dela, junto com o percentual de memória, ambos informados pelo usuário, como já explicados na classe Semáforo. Como já dito, a lógica de codificação da heap é praticamente igual à do Alocador para versão paralela, a única diferença é no método de desalocar. Sendo assim, quando a memória estiver muito ocupada e a quantidade de elementos contidos na heap for maior que o percentual de elementos que pode ter, é chamado o desalocar. Da mesma forma, quando ocorre a tentativa de alocar em um espaço que tentou ocorrer a fragmentação, se mesmo compactando não cabe ali, é desalocado da heap. A heap possui seu próprio método para desalocar uma requisição. Esse método tira da lista de ocupados e coloca na lista de livres. Como nossos segmentos são ordenados de maior a menor, ele pega o maior elemento da fila de ocupados para desocupar. Depois de pegar esta requisição de tamanho X e zerar todos esses tamanhos X, esse novo espaço é adicionado à fila de livres e a mesma é ordenada novamente. Por fim, é removido o espaço.

6. Testes

Esta seção descreve todos os testes realizados a fim de verificar, de forma adequada, a qualidade do software projetado. Para isso, foram montados planos de testes, juntamente com um cronograma.

6.1. Plano de Testes

O planejamento do plano de testes deu-se pela subdivisão do mesmo em categorias: tabela de requisitos, detalhamento da abordagem dos testes,

ferramentas, casos de teste e considerações dos resultados. A seguir, estão as tabelas utilizadas em cada uma das subdivisões.

Requisito	Prioridade	Descrição
RF001	Essencial	O usuário do sistema deve ser capaz de configurar o tamanho da heap para funcionamento do sistema.
RF002	Essencial	O usuário do sistema deve ser capaz de configurar o intervalo de valores (mínimo e máximo) para geração randômica do tamanho das variáveis a cada requisição produzida pelo gerador.
RF003	Essencial	O algoritmo de alocação deve alocar a memória para cada variável na heap sempre de forma contígua.
RF004	Essencial	O algoritmo de alocação deve monitorar o índice de porcentagem de espaços ocupados na heap, iniciando a execução de um desalocador de memória para liberar espaço na heap sempre que a porcentagem estiver acima do percentual máximo informado pelo usuário.
RF005	Essencial	O algoritmo de desalocação deve selecionar, de forma randômica, variáveis a serem desalocadas (simulando um fluxo natural de um sistema real com desalocações).
RF006	Essencial	O algoritmo de desalocação deve executar de forma paralela com o alocador.

RF007	Essencial	O número de requisições de alocação a serem alocadas na memória deve ser informado pelo usuário do sistema.
RF008	Essencial	O usuário deve informar o limite máximo de ocupação de memória como um percentual, acima do qual o algoritmo de desalocação de memória deve ser executado.
RF009	Essencial	O usuário deve informar o percentual mínimo de espaço livre na memória.
RF010	Essencial	O desalocador deve liberar memória até que o percentual de memória livre fique igual ou abaixo da porcentagem mínima informada.
RF011	Essencial	O vetor deve ser implementado como uma fila circular.
RF012	Essencial	Cada requisição deve informar ao algoritmo de alocação de memória o tamanho da variável dinâmica a ser alocada, bem como seu respectivo identificador.
RF013	Essencial	Um gerador de requisições randômicas deve alimentar o vetor de requisições.
RF014	Essencial	O alocador de memória e o gerador de requisições randômicas devem compor duas threads do sistema (uma thread é o alocador de memória e outra thread é o gerador de requisições).
RF015	Essencial	implementar um algoritmo de compactação para eliminar a

		fragmentação externa que se forma à medida que as variáveis vão sendo desalocadas.
--	--	--

Tabela 1 - Tabela de Requisitos

Tipo de teste:	Funcional
Subtipo de teste:	Requisitos
Requisitos que motivaram:	RF001
Objetivo do teste:	Testar se o tamanho da heap informado pelo usuário está sendo armazenado corretamente em sua respectiva variável.

Tipo de teste:	Funcional
Subtipo de teste:	Requisitos
Requisitos que motivaram:	RF001
Objetivo do teste:	Testar se a heap está realmente sendo gerada com o tamanho especificado.

Tipo de teste:	Funcional
Subtipo de teste:	Requisitos
Requisitos que motivaram:	RF002
Objetivo do teste:	Testar se o intervalo de valores (mínimo e máximo) para geração randômica informados pelo usuário estão sendo armazenados em suas respectivas variáveis.

Tipo de teste:	Funcional
Subtipo de teste:	Requisitos
Requisitos que motivaram:	RF002
Objetivo do teste:	Testar se o intervalo de valores (mínimo e máximo) informados pelo usuário estão sendo utilizados da forma correta, gerando variáveis que possuem tamanho de valores existentes entre o mínimo e o máximo informado.

Tipo de teste:	Funcional
Subtipo de teste:	Requisitos
Requisitos que motivaram:	RF003
Objetivo do teste:	Verificar se o algoritmo de alocação está realizando a alocação da heap a cada requisição removida do vetor de requisições.

Tipo de teste:	Funcional
Subtipo de teste:	Requisitos
Requisitos que motivaram:	RF003
Objetivo do teste:	Verificar se o algoritmo de alocação está alocando corretamente as informações da requisição (Tamanho e ID)

Tipo de teste:	Funcional
-----------------------	-----------

Subtipo de teste:	Requisitos
Requisitos que motivaram	RF003
Objetivo do teste:	Verificar se o algoritmo de alocação está realizando a alocação de forma contígua.

Tipo de teste:	Funcional
Subtipo de teste:	Requisitos
Requisitos que motivaram:	RF004
Objetivo do teste:	Testar se o algoritmo de alocação está inicializando a execução de um desalocador de memória quando a porcentagem de ocupação da heap atinge um valor igual ou maior que seu percentual máximo informado pelo usuário.

Tipo de teste:	Funcional
Subtipo de teste:	Requisitos
Requisitos que motivaram:	RF005
Objetivo do teste:	Testar se o desalocador está realizando sua função de liberar espaço na heap sempre que for executado pelo algoritmo de alocação.

Tipo de teste:	Funcional
Subtipo de teste:	Requisitos

Requisitos que motivaram:	RF006
Objetivo do teste:	Verificar se o algoritmo de desalocação e o alocador estão sendo executados de forma paralela.

Tipo de teste:	Funcional
Subtipo de teste:	Requisitos
Requisitos que motivaram:	RF007
Objetivo do teste:	Testar se o número de requisições de alocação informado pelo usuário está sendo armazenado em sua respectiva variável.

Tipo de teste:	Funcional
Subtipo de teste:	Requisitos
Requisitos que motivaram:	RF007
Objetivo do teste:	Testar se o número de requisições de alocação informado pelo usuário está sendo utilizado de forma correta, ou seja, se o vetor realmente possui este número específico de requisições armazenadas.

Tipo de teste:	Funcional
Subtipo de teste:	Requisitos
Requisitos que motivaram:	RF008
Objetivo do teste:	Verificar se o percentual máximo definido pelo usuário para o limite de ocupação da

	heap está sendo armazenado em sua respectiva variável.
--	--

Tipo de teste:	Funcional
Subtipo de teste:	Requisitos
Requisitos que motivaram:	RF008
Objetivo do teste:	Verificar se o percentual definido pelo usuário para o limite de ocupação da heap está sendo utilizado de forma correta, ou seja, se o alocador está ativando a desalocação quando o percentual atual de ocupação da heap atinge o valor máximo definido.

Tipo de teste:	Funcional
Subtipo de teste:	Requisitos
Requisitos que motivaram:	RF009
Objetivo do teste:	Verificar se o percentual definido pelo usuário para o valor mínimo de espaço ocupado na heap está sendo armazenado em sua respectiva variável.

Tipo de teste:	Funcional
Subtipo de teste:	Requisitos
Requisitos que motivaram:	RF009
Objetivo do teste:	Verificar se o percentual definido pelo usuário para o valor mínimo de espaço

	ocupado na heap está sendo utilizado corretamente na desalocação, ou seja, se o desalocador está realizando a desalocação dos espaços na heap até que atinja uma porcentagem de ocupação menor ou igual ao mínimo definido pelo usuário.
--	--

Tipo de teste:	Funcional
Subtipo de teste:	Requisitos
Requisitos que motivaram:	RF012
Objetivo do teste:	Verificar se cada requisição está informando ao algoritmo de alocação o tamanho de sua variável dinâmica a ser alocada, bem como seu respectivo identificador.

Tipo de teste:	Funcional
Subtipo de teste:	Requisitos
Requisitos que motivaram:	RF013
Objetivo do teste:	Verificar se a geração de cada requisição está sendo feita de forma randômica.

Tipo de teste:	Funcional
Subtipo de teste:	Requisitos
Requisitos que motivaram:	RF013
Objetivo do teste:	Verificar se cada requisição está realmente sendo armazenada no vetor de requisições.

Tipo de teste:	Funcional
Subtipo de teste:	Requisitos
Requisitos que motivaram:	RF015
Objetivo do teste:	Verificar se o algoritmo de compactação está sendo executado quando detecta-se a necessidade, ou seja, quando o somatório retorna um valor maior que o tamanho da requisição a ser alocada.

Tipo de teste:	Funcional
Subtipo de teste:	Requisitos
Requisitos que motivaram:	RF015
Objetivo do teste:	Verificar se o algoritmo de compactação está fazendo o que deveria, ou seja, realocando todos os espaços para que exista apenas um buraco contíguo na heap ao invés de diversos menores divididos pela heap.

Tabelas 2 - Detalhamento da abordagem de testes

Caso de uso	ID	Passos	Resultado Esperado
UC001 - Testar se a heap está realmente sendo gerada com o tamanho especificado (RF001).	1	Na interface do programa, coloque o tamanho da heap em 10 e preencha os outros dados normalmente.	
	2	Clique em “Adicionar dados” e após	Tanto a impressão da

		isso clique em “imprimir heap”.	heap paralela quanto da heap sequencial devem mostrar que suas páginas vão de 0 a 9.
--	--	---------------------------------	--

UC002 - Testar se o intervalo de valores (mínimo e máximo) informados pelo usuário estão sendo utilizados da forma correta, gerando variáveis que possuem tamanho de valores existentes entre o mínimo e o máximo informado (RF002).	1	Na interface do programa, coloque o tamanho mínimo da requisição em 1 e o tamanho máximo em 5, preencha os outros dados normalmente.	
	2	Clique em “Adicionar dados” e após isso clique em “imprimir vetor de requisições”.	Os valores informados de cada requisição deverá ser um número existente de 1 a 5.

UC003 - Verificar se o algoritmo de alocação está realizando a alocação da heap a cada requisição removida do vetor de requisições (RF003).	1	Rode a versão de testes do programa, preencha as informações como bem entender, contanto que não utilize valores = 0.	
	2	Ao executar o programa sequencialmente e paralelamente, compare o print da heap, lista de ocupados e vetor de requisições a cada alocação realizada.	A cada ciclo em que uma requisição é removida do vetor de requisições, esta deverá aparecer em algum espaço da heap.

UC004 - Verificar se o algoritmo de alocação está alocando corretamente as informações da requisição (Tamanho e ID) (RF003).	1	Rode a versão de testes do programa, preencha as informações como bem entender, contanto que não utilize valores = 0.	
	2	Ao executar o programa sequencialmente e paralelamente, compare o print da heap, lista de ocupados e vetor de requisições a cada alocação realizada.	Quando uma requisição é removida do vetor de requisições, seu tamanho deverá ser armazenado na lista de ocupados, juntamente com sua ID e a posição inicial em que ela está armazenada atualmente na heap.

UC005 - Verificar se o algoritmo de alocação está realizando a alocação de forma contígua (RF003).	1	Rode a versão de testes do programa, coloque uma heap de tamanho 10, 5 requisições, porcentagem máxima de 90, mínima de 40 e o tamanho da variável dinâmica sendo algo entre 1 e 2.	
	2	Ao executar o programa sequencialmente e paralelamente, compare o print da heap, lista de ocupados e vetor de requisições a cada alocação realizada.	Quando uma requisição é removida do vetor de requisições, a mesma deverá ser alocada de forma contígua na memória, ou seja, enquanto não houver buracos na heap o software deverá alocar de forma contígua cada uma das variáveis,

			respeitando seus respectivos tamanhos.
--	--	--	--

UC006 - Testar se o algoritmo de alocação está inicializando a execução de um desalocador de memória quando a porcentagem de ocupação da heap atinge um valor igual ou maior que seu percentual máximo informado pelo usuário (RF004 e RF008).	1	Rode a versão de testes do programa, coloque uma heap de tamanho 10, 5 requisições, porcentagem máxima de 50, mínima de 30 e o tamanho da variável dinâmica sendo 1 (tanto o máximo quanto o mínimo).	
	2	Ao executar o programa sequencialmente e paralelamente, compare o print da heap, lista de ocupados e vetor de requisições a cada alocação realizada.	Quando o print mostrar que 5 ou mais espaços estão ocupados na heap, no próximo ciclo um print deverá aparecer na tela com a mensagem “O desalocador foi chamado, porcentagem de ocupação da heap ultrapassou os 50%”

UC007 - Testar se o desalocador está realmente desalocando a heap quando é chamado, além de verificar se o percentual definido pelo usuário para o valor mínimo de espaço ocupado na heap está sendo utilizado corretamente	1	Rode a versão de testes do programa, coloque uma heap de tamanho 10, 5 requisições, porcentagem máxima de 50, mínima de 30 e o tamanho da variável dinâmica sendo 1 (tanto o máximo quanto o mínimo).	
	2	Ao executar o programa sequencialmente e paralelamente, compare o print da heap, lista de ocupados e vetor de requisições a cada alocação realizada.	Quando o print mostrar que 5 ou mais espaços estão ocupados na heap, anote o ID destes e compare com o próximo print do vetor de

na desalocação (RF005 e RF009).			ocupados, ao menos um ID do ciclo anterior deverá ter desaparecido da lista (desalocado) e seus respectivos blocos estarão vazios. A desalocação apenas irá ser finalizada quando a heap possuir apenas 3 ou menos blocos ocupados (30%)
---------------------------------	--	--	--

UC007 - Testar se o número de requisições de alocação informado pelo usuário está sendo utilizado de forma correta, ou seja, se o vetor realmente possui este número específico de requisições armazenadas (RF007).	1	Na interface do programa, coloque o número de requisições em 10, preencha os outros dados normalmente.	
	2	Clique em “Adicionar dados” e após isso clique em “imprimir vetor de requisições”.	O vetor de requisições impresso deverá possuir exatamente 10 espaços, com todos preenchidos com requisições de tamanhos distintos.

UC008 - Verificar se cada requisição está informando ao algoritmo de alocação o tamanho de sua variável dinâmica a ser alocada, bem como seu respectivo identificador (RF012).	1	Na interface do programa, coloque o número de requisições em 10, tamanho da requisição entre 1 e 5, preencha os outros dados normalmente.	
	2	Clique em “Adicionar dados”, depois disso rode o programa em modo paralelo e modo sequencial, comparando o print da heap, vetor de ocupados e vetor de requisições a cada	Veja se o tamanho da variável da primeira requisição removida do vetor de requisições está na lista de

		ciclo.	ocupados após a primeira alocação, juntamente com seu identificador.
--	--	--------	--

UC009 - Verificar se o somatório está realizando o cálculo corretamente.	1	Na interface do programa, coloque o número de requisições em 10, tamanho da requisição entre 1 e 5, preencha os outros dados normalmente.	
	2	Clique em “Adicionar dados”, depois disso rode o programa em modo paralelo e modo sequencial.	Analisando o print da heap, conte quantos blocos vazios ela possui (blocos com valor = 0), após isso, compare este valor com o valor informado pelo somatório, ambos precisam ser iguais.

UC010 - Verificar se a compactação está sendo realizada de forma correta (RF015)	1	Rode a versão de testes do programa, coloque uma heap de tamanho 10, 5 requisições, porcentagem máxima de 50, mínima de 30 e o tamanho da variável dinâmica sendo algo entre 1 e 5.	
	2	Clique em “Adicionar dados” e rode o programa tanto de forma sequencial quanto paralela	Uma mensagem falando “compactação realizada!” deverá aparecer na tela caso uma situação que necessite de compactação ocorra,

			compare o print da heap realizado previamente à compactação e o print atual, verifique se todos os espaços ocupados agora estão contíguos e os buracos (fragmentação externa) agora se tornaram apenas um grande buraco contíguo.
--	--	--	---

UC011 - Verificar se a interface mostra um aviso de erro caso o usuário não preencha algum campo da interface.	1	Na interface do programa, preencha todos os dados exceto um (você precisará fazer isso 6 vezes, uma para cada campo).	
	2	Clique em “Adicionar dados”.	Uma mensagem falando “Há campos de textos vazios, insira dados” deverá aparecer na tela.

UC012 - Verificar se a interface mostra um aviso de erro caso o usuário coloque um tamanho de variável máximo maior que o tamanho da heap informado	1	Na interface do programa, preencha todos os dados normalmente, porém coloque o tamanho máximo da variável da requisição maior que o tamanho da heap.	
	2	Clique em “Adicionar dados”.	Uma mensagem falando “Tamanho da requisição maior que o permitido” deverá aparecer na tela.

UC013 - Verificar se a interface mostra um aviso de erro caso o usuário coloque um tamanho de variável mínimo maior que o tamanho de variável máximo informado	1	Na interface do programa, preencha todos os dados normalmente, porém coloque o tamanho mínimo da variável da requisição maior que o tamanho máximo.	
	2	Clique em “Adicionar dados”.	Uma mensagem falando “Tamanho mínimo da requisição está maior que o permitido (maior que o tamanho máximo definido)” deverá aparecer na tela.


Tabela 3 - Casos de Teste

7. Manual do Usuário

Nesta seção, será descrito o passo a passo para a utilização da aplicação. Para melhor entendimento, será utilizado imagens.

Ao iniciar o sistema, possuímos a tela que faz a configuração da heap, do vetor de requisição e da própria requisição, como ilustra a figura 6. Dessa maneira, o usuário deve informar qual tamanho da heap (em bloco) ele deseja, bem como seus percentuais máximos e mínimos de ocupação, o número de requisições que será atendido pela memória, como também os valores de tamanho máximos e mínimos (em bloco) que cada requisição pode vir a possuir. Depois de inserir estes dados, o usuário deve clicar no botão “adicionar dados”. Inseridos os dados, o usuário pode escolher a forma de execução: sequencial e paralela. Ao final da execução do programa, uma janela será aberta contendo o tempo final de execução da opção escolhida. É possível também ver o vetor da heap, ver a lista de segmentos ocupados com informações como seu identificador, início e tamanho e é

possível ver o estado atual do vetor de requisições, basta apenas clicar no botão corresponde a cada uma destas opções.



Gerenciador de Memória

GERENCIADOR DE MEMORIA

Configurações Heap:

TAMANHO DA HEAP

PERCENTUAL MÁXIMO

PERCENTUAL MÍNIMO

Configurações Vetor Requisição:

NÚMERO DE REQUISIÇÕES

Configurações Requisição

TAMANHO MÍNIMO

TAMANHO MÁXIMO

ADICIONAR DADOS

Informações Extras:

VER HEAP

VER OCUPADOS

VER VETOR DE REQUISIÇÕES...

EXECUTAR EM MODO

PARALELO SEQUENCIAL

Figura 6 - Interface Gráfica