Universidade Federal do Pampa – UNIPAMPA
Curso Bacharelado em Engenharia de Computação

Disciplina de Engenharia de Software
Prof. Carlos Michel Betemps

Documentação de Software - Gerenciador Dinâmico de Memória.

Integrantes do grupo: André Magalhães, Douglas Aquino, Ronaldo Amato.

Universidade Federal do Pampa

Este documento visa esclarecer os conceitos principais e métodos abordados durante a solução aplicada para criar um algoritmo paralelo/concorrente de gerenciamento dinâmico de memória. Como abordagem inicial, temos uma análise de requisitos globais do problema, buscando estabelecer os objetivos, com auxílio das ferramentas disponibilizadas pela disciplina de engenharia de software. Esta aplicação, por se tratar de um sistema de modelagem e simulação, inclui diversas classes e objetos separados que interagem entre si, colaborativamente. Diante disto, o modelo ideal para a elaboração do processo de desenvolvimento de software, foi selecionado o modelo de desenvolvimento incremental, auxiliado pela metodologia ágil e suas ferramentas, buscando tornar viável um ambiente de possíveis mudanças referentes à alterações durante a implementação do projeto.

A fim de definir os requisitos de usuário, visto que o software será estruturado para trabalhar de forma paralela, há parâmetros que devem ser definidos anteriormente ao seu início propriamente dito. O software deve fazer a gerência entre as requisições que serão criadas entre as filas de processamento e fazer a alocação e desalocação na memória. O usuário deverá inserir os valores correspondentes ao gerador de requisições, quantidade e tamanho de requisições e dados sobre a alocação em memória, que devem ser fornecidos durante a inicialização do sistema.

Para a elaboração da engenharia de requisitos de sistema, foi feita a formulação de um fluxograma, buscando compreender a forma de execução que pode ser implementada, de forma funcional. O usuário deve inserir dados na inicialização do sistema que são:

- Tamanho da Heap, intervalo de valores para tamanho das variáveis;
- Limite de ocupação de memória como percentual, acima do qual o algoritmo de desalocação de memória deve ser executado, assim como o valor mínimo em que a memória deve ficar livre;
- Número de requisições a serem alocadas na memória, assim como seu tamanho.

Após a inserção destes dados, se iniciará a geração das requisições de forma aleatória, onde cada requisição ocupará um pedaço do vetor de requisições, com tamanho variável dentro do limite estipulado (mínimo e máximo).

Paralelamente, ao perceber a inserção destas requisições na fila de prontos do vetor de requisições, o escalonador iniciará o seu fluxo, escalando as requisições conforme a ordem de chegada na fila, direcionando assim o seu processamento.

Durante este processo, (tipo de escalonador a ser definido, SJF ou RR):

Se SJF: o escalonador verificará o tempo de demanda para execução e definir assim a sua prioridade, visto que as requisições com menor tempo de demanda serão executadas primeiro. Caso exista uma requisição em processamento com a demanda maior no momento de entrada desta nova requisição, será executada a troca de contexto, onde será movido o processo em execução para a fila de waiting, e será dado início ao processamento desta nova requisição.

Se RR: o escalonador será responsável por ordenar as requisições, de acordo com a ordem de chegada, assim como durante a troca de contexto, que deve obedecer um limite definido para execução do quantum do cpu, ordenando assim as filas de requisições ready e waiting.

Enquanto estas operações são executadas, um monitor deve verificar o nível de memória disponível, onde fica responsável por executar as alocações de forma contígua e desalocação do vetor de requisições de forma aleatória.

O número de requisições, ao ser atingido após o processamento, definirá o final do programa.

Como diretrizes deste projeto, procurando atingir um bom nível de qualidade, estes componentes estão sendo arquitetados de forma a trabalharem modularmente, utilizando abstrações de arquitetura a fim de obter-se representações com características funcionais independentes, assim como o seu acoplamento fique em níveis aceitáveis. Cada classe do projeto foi criada com estes conceitos. As classes são:

- PCB
  - Está atribuído a esta classe, armazenar as filas de requisições de processos, assim como os métodos de adição, obtenção e mudança entre as filas.

- Process
  - Responsável por definir individualmente as características de cada processo através de seu construtor, com seu tamanho, tempo de execução e sua identificação.

- RequisitionsArray
  - Vetor de requisições que tem por definição armazenar os processos criados.

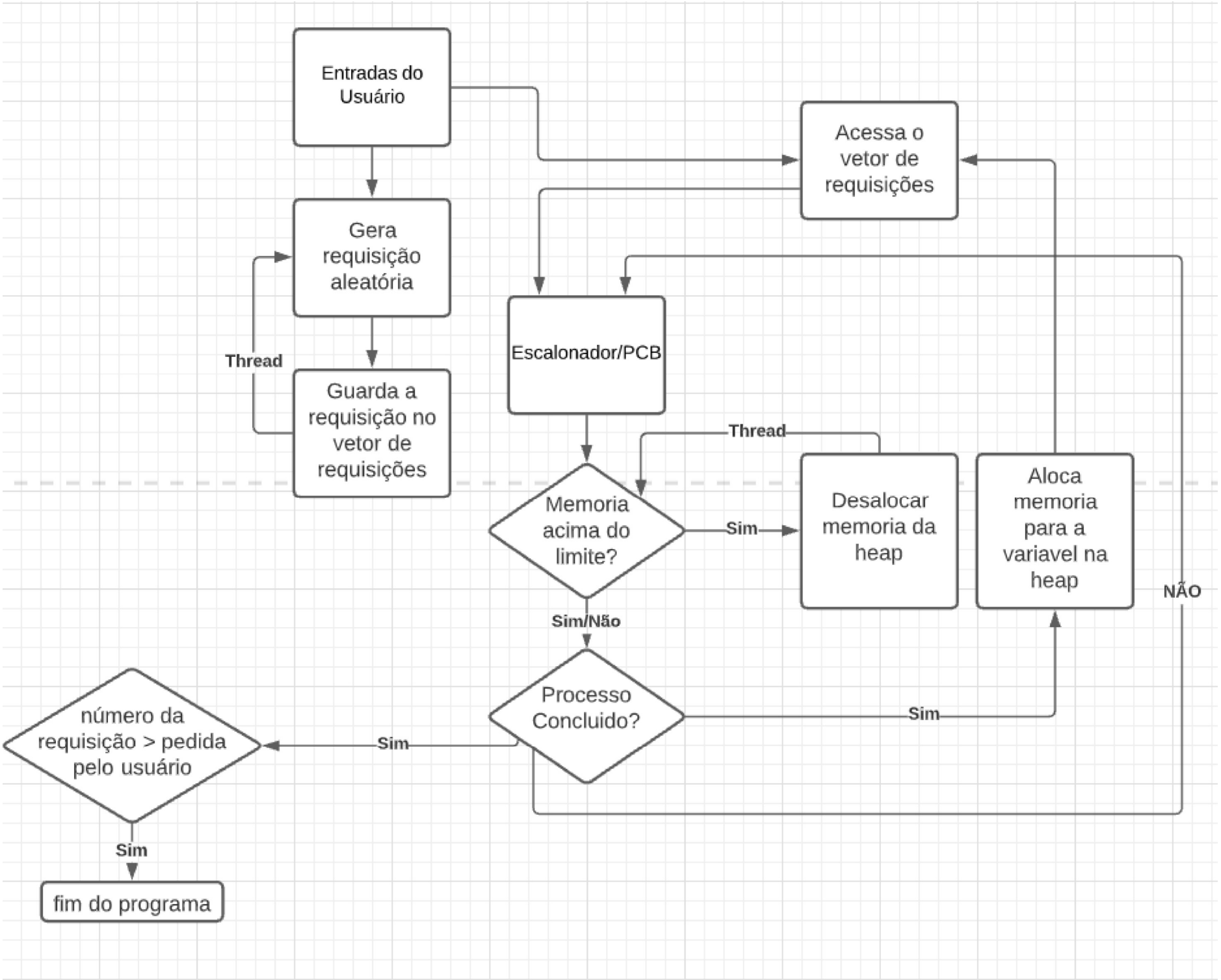
- CircularQueue
  - Superclasse de RequisitionArray, é responsável por garantir a circulação do início e do fim do vetor de requisições.

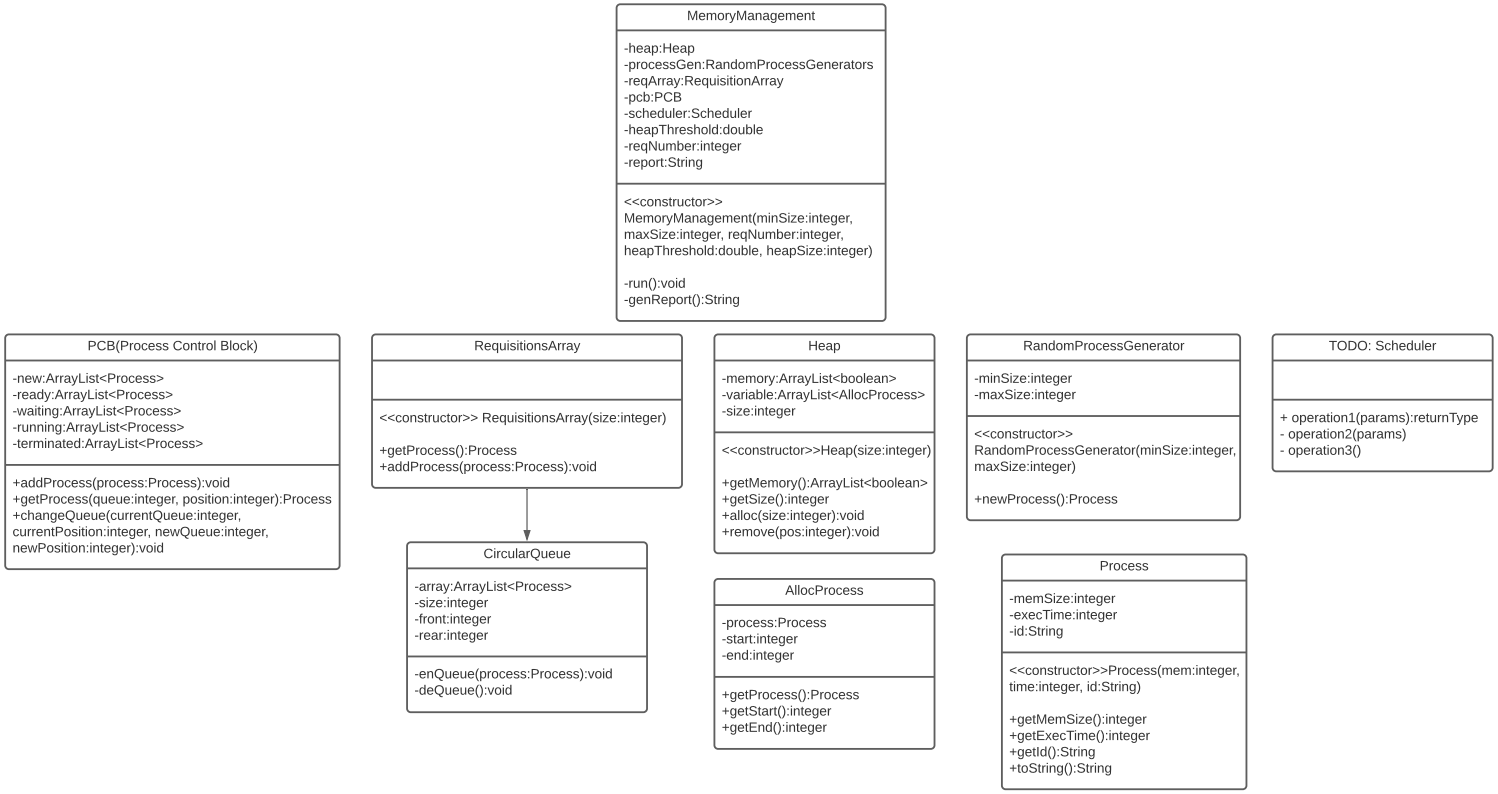
- RandomProcessGenerator
  - Classe responsável por definir as características dos objetos da classe Process, sendo responsável pela criação destes com as definições recebidas pelo usuário.

- AllocProcess
    - Esta classe representa os processos alocados em memória, criados anteriormente pela RandomProcessGenerator, contendo sua posição de início e fim no vetor de memória, assim como o processo que deu sua origem.

- Heap
    - Possui um ArrayList que simulará o funcionamento da memória, com tamanho definido também pelo usuário ao iniciar o sistema. Possui os métodos de alocação e desalocação internos, assim como vetores que representam a memória e os processos que ali estão alocados.
- Scheduler
    - a definir.

- MemoryManagement
    - Responsável pela união dos demais componentes em um sistema como um todo.


Entradas do Usuário
Acessa o vetor de requisições
Gera requisição aleatória
Thread
Guarda a requisição no vetor de requisições
Escalonador/PCB
Thread
Memoria acima do limite?
Sim
Desalocar memoria da heap
Aloca memoria para a variavel na heap
NÃO
Sim/Não
Processo Concluido?
Sim
número da requisição > pedida pelo usuário
Sim
Sim
fim do programa


MemoryManagement
-heap:Heap
-processGen:RandomProcessGenerators
-reqArray:RequisitionArray
-pcb:PCB
-scheduler:Scheduler
-heapThreshold:double
-reqNumber:integer
-report:String
<<constructor>>
MemoryManagement(minSize:integer, maxSize:integer, reqNumber:integer, heapThreshold:double, heapSize:integer)
-run():void
-genReport():String
PCB(Process Control Block)
-new:ArrayList<Process>
-ready:ArrayList<Process>
-waiting:ArrayList<Process>
-running:ArrayList<Process>
-terminated:ArrayList<Process>
+addProcess(process:Process):void
+getProcess(queue:integer, position:integer):Process
+changeQueue(currentQueue:integer, currentPosition:integer, newQueue:integer, newPosition:integer):void
RequisitionsArray
<<constructor>> RequisitionsArray(size:integer)
+getProcess():Process
+addProcess(process:Process):void
CircularQueue
-array:ArrayList<Process>
-size:integer
-front:integer
-rear:integer
-enQueue(process:Process):void
-deQueue():void
Heap
-memory:ArrayList<boolean>
-variable:ArrayList<AllocProcess>
-size:integer
<<constructor>>Heap(size:integer)
+getMemory():ArrayList<boolean>
+getSize():integer
+alloc(size:integer):void
+remove(pos:integer):void
AllocProcess
-process:Process
-start:integer
-end:integer
+getProcess():Process
+getStart():integer
+getEnd():integer
RandomProcessGenerator
-minSize:integer
-maxSize:integer
<<constructor>>
RandomProcessGenerator(minSize:integer, maxSize:integer)
+newProcess():Process
Process
-memSize:integer
-execTime:integer
-id:String
<<constructor>>Process(mem:integer, time:integer, id:String)
+getMemSize():integer
+getExecTime():integer
+getId():String
+toString():String
TODO: Scheduler
+ operation1(params):returnType
- operation2(params)
- operation3()