Implementações Sistemas Operacionais - Segunda unidade
Docente: Dr. Leiva Casemiro Oliveira

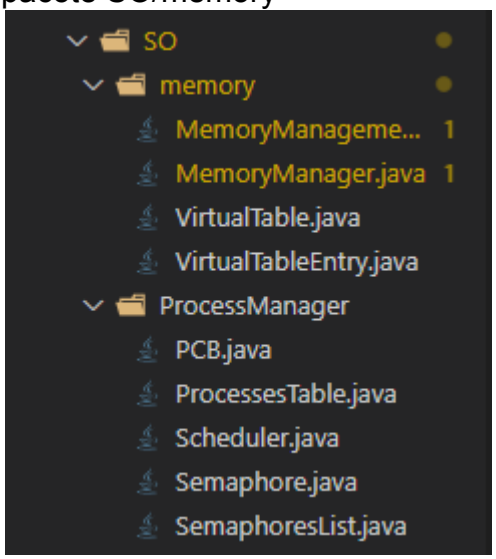
Grupo 2:
João Felipe Barros Silva
Ríad Oliveira de Morais
Ricardo Cezar Fernandes de Melo Junior
Pedro David Rocha Saldanha
Vitor Duarte Bezerra de Oliveira

Arquivos .asm de testes e macros se encontram na pasta testeAssembly

Tarefa 2.1 - 2.2

Tarefa 2.1: Estruturas para o gerenciamento de memória

Pacotes com as classes para o gerenciamento de memória foram criados dentro do pacote SO/memory



Adicione atributos na sua classe PCB (*Process Control Block*) para manter informações a respeito do gerenciamento de memória do processo tais como:

- Registrador de limite superior da memória do processo
- Registrador de limite inferior da memória do processo

```
public class PCB {
    private int enderecoInicio = 0;  // regSuperior;
    private int enderecoFim = 0;     // regInferior;
```

- Os registradores de limites de memória devem ser configurados como:
  - Limite superior: endereço inicial do processo (*label* que o identifica, que também é usado pelo *Fork*)

```java
public static void adicionarProcesso(PCB novoProcesso) {
    novoProcesso.setEstadoProcesso(estadoProcesso: "Pronto");

    if(novoProcesso.getEnderecoFim() == 0) {
        novoProcesso.setEnderecoFim(ultimoEnderecoPrograma);
    }

    int tamanhoLista = listaProcessos.size();
    if(tamanhoLista > 0) {
        PCB penultimoProcesso = listaProcessos.get(tamanhoLista - 1);

        if(penultimoProcesso.getEnderecoFim() == 0) {
            penultimoProcesso.setEnderecoFim(
                novoProcesso.getEnderecoInicio() - 4
            );
        }
    }

    listaProcessos.add(novoProcesso);
}
```

- Limite inferior: endereço imediatamente antes do *label* do próximo processo. Caso seja o último processo, deve ser configurado com o endereço final do programa

```java
public static void setUltimoEnderecoPrograma(
    int ultimoEnderecoProgramaRecebido
) {
    ultimoEnderecoPrograma = ultimoEnderecoProgramaRecebido;
}
```

- Cada vez que o processo for escalonado, além dos registradores físicos do processador, os valores dos registradores de limites da memória também devem atualizados das PCBs de cada processo

Criar uma classe de gerenciador de memória com atributos globais (e seus respectivos métodos *get* e *set*) para o todos os processos, tais como:

- Tamanho de bloco de alocação de memória (tamanho da página virtual) em quantidade de instruções do MIPS
- Quantidade máxima de blocos de alocação por processo
- Configuração do tipo de algoritmo de substituição de páginas da memória virtual

```
public class MemoryManager {
    /*Foram utilizados valores padrões encontrados
    no livro de Tanenbaum para os possíveis tamanhos
    de página. A de 4kb é uma das mais usadas no mercado*/
    private static int tamPagVirtual; //4kb, 8kb, 16kb, 32kb, 64kb

    //----------------------------------------
    /*Quantidade máxima de molduras na memória física.
    A quantidade de molduras também pode ser entendida
    como a quantidade máxima de páginas virtuais mapeadas
    permitida.*/
    private static int qntMaxBlocos; // 4, 8, 16, 32

    //----------------------------------------
    private static String algoritmoSubstituicao;
```

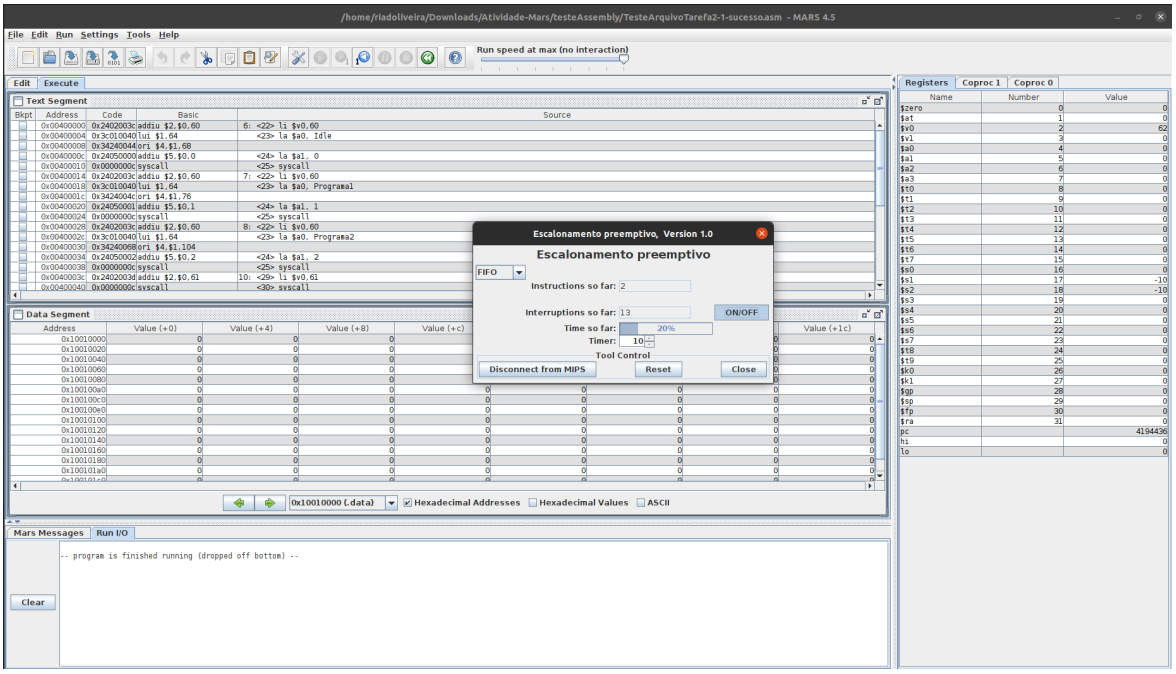
A cada acesso a memória (segmento de código) o gerenciador deve:

- Comparar se endereço acessado está dentro dos limites de memória para o processo em execução
- Se endereço for válido continue executando normalmente
- Caso contrário imprime uma mensagem de erro na saída padrão do MARS, informando a tentativa de endereço a ser acessado e os limites do processo, em seguida pare a simulação.

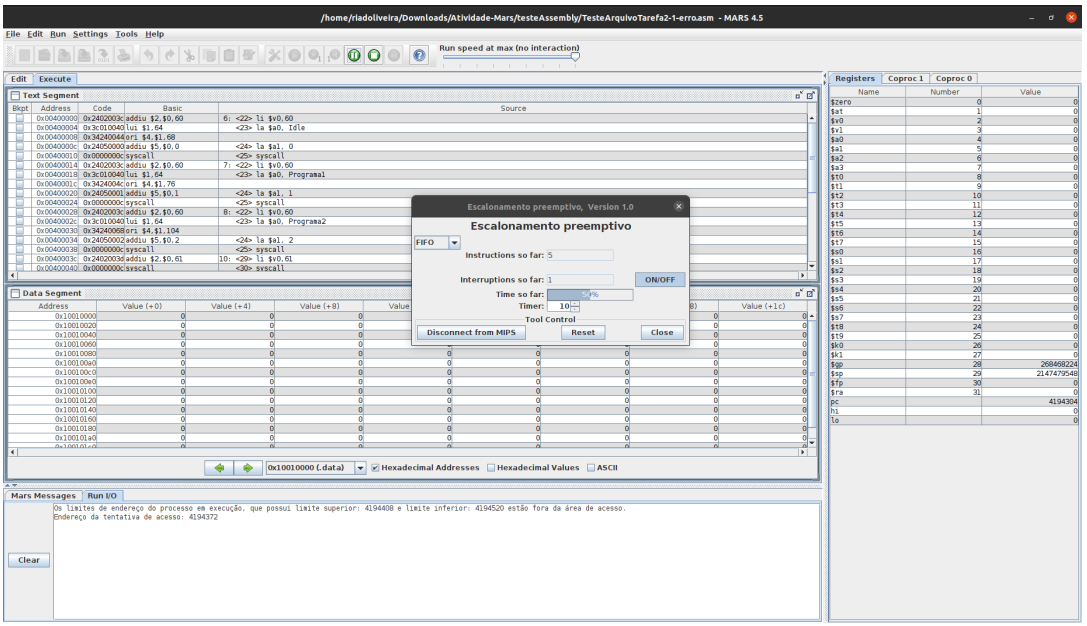
```java
public static void verificarMemoria() {
    PCB procExec = ProcessesTable.getProcessoExecutando();
    if(procExec == null) return;

    int pc = RegisterFile.getProgramCounter();
    if (procExec.getEnderecoInicio() > pc || procExec.getEnderecoFim() < pc){
        SystemIO.printString(
            "Os limites de endereço do processo em execução, que possui limite superior: " +
            procExec.getEnderecoInicio() + " e limite inferior: " +
            procExec.getEnderecoFim() + " estão fora da área de acesso.\n"
        );
        SystemIO.printString("Endereço da tentativa de acesso: " + pc);

        System.out.println(
            "Os limites de endereço do processo em execução, que possui limite superior: " +
            procExec.getEnderecoInicio() + " e limite inferior: " +
            procExec.getEnderecoFim() + " estão fora da área de acesso."
        );
        System.out.println("Endereço da tentativa de acesso: " + pc);

        Simulator.getInstance().stopExecution(new AbstractAction() {
            @Override
            public void actionPerformed(ActionEvent arg0) {
            }
        });
    } else {
        /*A MMU deve verificar os campos "índice"
        e "deslocamento" do endereço virtual para
        mapeá-lo para endereço físico, para isso
        é necessário usar o endereço armazenado
        em PC e dividi-lo nos 2 campos mencionados
        */

        /*Primeiro atribui o endereço atual
        armazenado no pc para o endereço virtual*/
        String enderecoVirtual = Integer.toBinaryString(pc);
    }
}
```

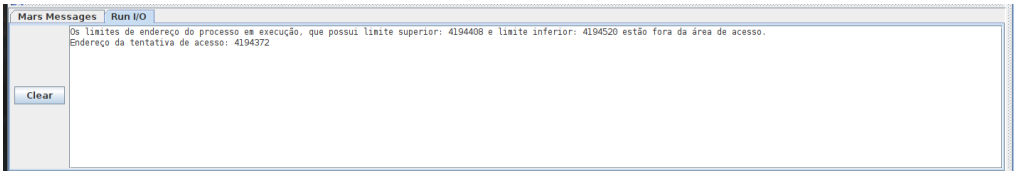
Para validar (Observação: Os testes foram realizados com a tool do último projeto aberta “Preemptive Scheduling”, para que fosse possível efetuar a verificação de memória a cada linha de execução do programa):

- Execute o código a seguir, verificando se os limites de memória são os esperados



- Em seguida modifique o código de um dos Programas para que salte para o código do outro Programa e verifique se o erro é detectado.





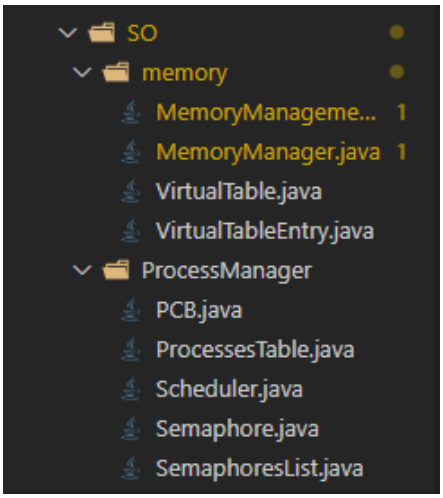
Tarefa 2.2:

- O Gerenciador de Memória Virtual (GMV) será responsável por criar e manter um espaço de endereçamento virtual para cada processo como ilustrado na figura a seguir

```
public static int getTamPagVirtual() {
    return tamPagVirtual;
}

public static void setTamPagVirtual(int tamPagVirtual) {
    MemoryManager.tamPagVirtual = tamPagVirtual;
}
```

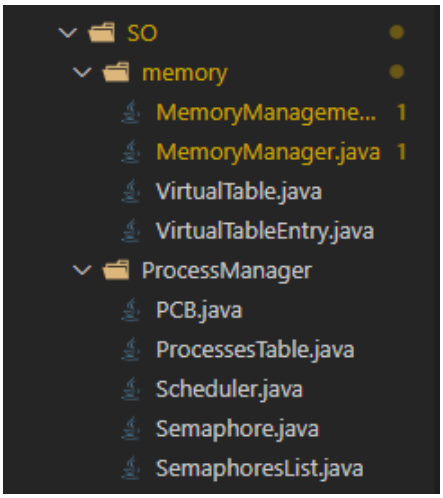
- Assim, cada processo terá uma tabela que mapeia seus endereços virtuais para endereços de memória física (até o limite daquele processo)
- As páginas de endereços virtuais não mapeados estão no disco
- Assim o GMV deve:
    - Criar tabelas de mapeamento de endereços virtuais para os processos
    - Utilizar blocos de alocação (páginas) para troca de informações entre disco e memória física
    - Contabilizar os endereços físicos disponíveis para o processo
    - Mapear endereço virtual (páginas) em endereço físico disponível (molduras)
    - Desmapear um endereço físico para mapear outro que vem do disco quando o limite dos endereços dos processos for atingido
    - Desmapear todos os endereços quando o processo terminar

Classe Gerenciador de Memória virtual (MemoryManager.java) se encontra no seguinte pacote:



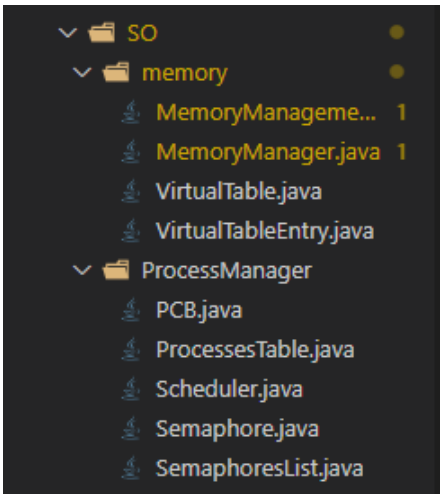
- É necessário implementar uma MMU (*Memory Manegment Unit*) que mantém a tabela de memória virtual para traduzir endereços virtuais em físicos.
- Criar uma classe “entrada da tabela virtual” com, no mínimo, os atributos:
    - Página Referenciada
    - Página Modificada
    - Proteção (bits R, W e X)
    - Bit Presente/ausente
    - Número da moldura mapeada

Classe Virtual Table Entry se encontra (VirtualTableEntry.java) se encontra no seguinte pacote: (muito grande para adicionar prints)



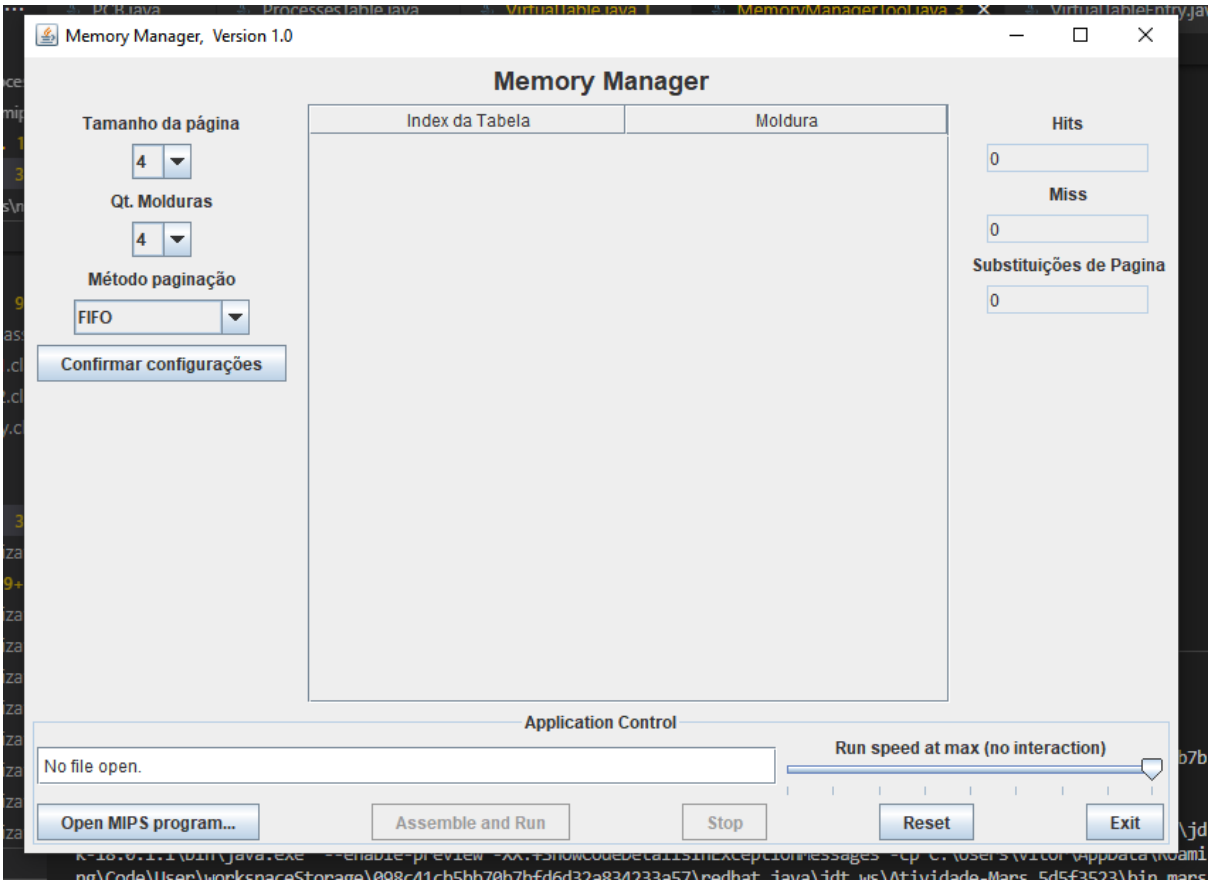
- Criar uma classe “tabela virtual” com uma coleção de “entradas da tabela virtual”
    - Cada processo terá sua própria tabela virtual, então pode-se optar por implementar um objeto “tabela virtual” para cada processo ou 1 único objeto que reserve uma quantidade fixa de “entradas da tabela” para cada processo

Classe Virtual Table se encontra (VirtualTable.java) se encontra no seguinte pacote: (muito grande para adicionar prints)



- O gerenciador de memória deve implementar memória virtual (considerando apenas dos endereços do segmento de código) simulada por meio de uma ferramenta que deve ser conectada ao MIPS no momento da execução do código *assembly*
- Sua ferramenta deve aproveitar o trabalho anterior e permitir realizar as seguintes configurações via interface gráfica:
    - Tamanho de bloco de alocação de memória (tamanho da página virtual) em quantidade de instruções do MIPS
    - Quantidade máxima de blocos de alocação por processo
    - Configuração do tipo de algoritmo de substituição de páginas da memória virtual (NRU, FIFO, Segunda chance, LRU)
    - As configurações não devem ser modificadas após iniciada a simulação (sugere-se haver um botão de confirmação das configurações para que fiquem desabilitadas logo após esse botão ser clicado e assim permanecer durante toda simulação)
  - Sua ferramenta também deve mostrar o estado atual da memória virtual de cada processo e as configurações escolhidas
    - O estado atual pode ser apenas do processo em execução ou de toda a memória (todos os processos)
    - O estado atual consiste em exibir informações sobre quais páginas virtuais estão mapeadas e em que molduras estão mapeadas, a quantidade de páginas virtuais disponíveis e se está acessando um bloco da memória física ou do disco. Inspire-se na figura a seguir
    - A ferramenta deve mostrar as configurações atuais
  - E ainda deve exibir a seguintes informações (totais e por processo)
    - Quantidade de hits na memória
    - Quantidade de miss na memória (acessos ao disco)
    - Quantidade de substituições de blocos
- Não há necessidade de modificações na memória do simulador

Print da ferramenta construída:



- Quando um processo é criado (*fork*), deve ser criada uma tabela de memória virtual para esse processo mas nenhuma moldura deve ser mapeada (ainda não acontece nenhuma transferidos do disco para memória) e os registradores de limite são setados

  Método chamado pela SyscallFork (Método da ProcessesTable), atualizada para a criação da VirtualTable no PCB:

```
public static void criarProcesso(int enderecoInicio, int prioridade){
    PCB novoProcesso = new PCB(
        enderecoInicio, prioridade,
        new VirtualTable(MemoryManager.getQntMaxBlocos())
    );
    adicionarProcesso(novoProcesso);
}
```

- A cada acesso a memória, a rifica se a página está alocada na tabela (*hit*) e se o endereço é válido
  - Se estiver traduz para o endereço físico, verifica se o endereço está nos limites do espaço de endereçamento

        - Se não estiver imprime uma mensagem de erro no MARS e a simulação deve parar
    - Se estiver dentro dos limites e se tal página está presente na memória e se é válida então permite o acesso.
    - Se o endereço não estiver mapeado para memória (*miss*)
        - Verifica se a quantidade máxima de páginas alocadas foi atingida
            - Se sim, faz a substituição (usando o algoritmo de substituição de página configura) pelo novo bloco
            - Se não, aloca o novo bloco em um espaço vazio
    - Vale salientar que uma página virtual deve conter mais de 1 endereço físico (de acordo com a configuração da ferramenta antes da simulação), logo, endereços físicos simultâneos podem estar dentro de uma mesma página
        - Assim, a MMU deve verificar os campos “índice” e “deslocamento” do endereço virtual para mapeá-lo para endereço físico, assim como na figura a seguir
        - Para isso é necessário usado o endereço armazenado em PC e dividi-lo nos 2 campos mencionados

- Como as páginas mapeiam n instruções e como cada instrução possui 4 bytes, uma sugestão para calcular em qual página um endereço virtual pode está mapeado, é dividir tal endereço pela quantidade de bytes dentro da página.
    - Ex: Suponha que a tabela de um processo tenha apenas 2 páginas virtuais (página 0 e página 1) e cada página armazena 2 instruções. Suponha também que as instruções desse processo estejam nos seguintes endereços: 0x00400000, 0x00400004, 0x00400008 e 0x0040000c. Para calcular a página para onde o endereço é mapeado basta dividir o endereço por 8 (quantidade de bytes de 2 instruções). Assim:
        - Quando o 1º endereço for solicitado, a MMU o mapearia para a página zero. O 2º endereço também seria mapeado para a página zero.
        - O 3º e 4º endereços serão mapeados para a página 1 e assim por diante.
    - Uma sugestão para o deslocamento dentro da página é dividir o endereço pelo tamanho (4 bytes) e o resultado módulo quantidade de endereços dentro da página.
        - Ex: Considerando o mesmo exemplo, para o 1º endereço temos: (0x00400000 / 4) mod 2 = 0; para o 2º endereço: (0x00400004 / 4) mod 2 = 1; para o 3º endereço: (0x00400008 / 4) mod 2 = 0; e para o 4º endereço: (0x0040000c / 4) mod 2 = 1;

- Se preferir o deslocamento em bytes, não realiza a divisão por 4, apenas o módulo por 16 (quantidade de bytes nas 2 entradas da tabela de página virtuais). E o resultado seria: para o 1º endereço: 0; para o 2º endereço: 4; para o 3º endereço: 8; para o 4º endereço: 12;

- Implementar os algoritmos de substituição de página: NRU, FIFO, Segunda chance, LRU, os quais serão invocados de acordo com a configuração inicial no momento que for necessário.

```
private VirtualTableEntry obterElementoNRU() {
    List<VirtualTableEntry> tabelaProvisoria = new ArrayList<>();
    tabelaProvisoria.addAll(tabelaEntradas);

    Collections.sort(tabelaProvisoria, new Comparator<VirtualTableEntry>() {
        @Override
        public int compare(VirtualTableEntry left, VirtualTableEntry right) {
            int leftValue = (left.getPaginaReferenciada() ? 1 : 0) +
            (left.getPaginaModificada() ? 1 : 0);

            int rightValue = (right.getPaginaReferenciada() ? 1 : 0) +
            (right.getPaginaModificada() ? 1 : 0);

            if(leftValue == rightValue) return 0;
            if(leftValue < rightValue) return -1;
            return 1;
        }
    });

    return tabelaProvisoria.get(index: 0);
}
```

```
  private VirtualTableEntry obterElementoFIFO() {
      return tabelaEntradas.get(index: 0);
  }
```

```
private VirtualTableEntry obterElementoSegundaChance() {
    VirtualTableEntry elementoIterativo = obterElementoFIFO();

    while(elementoIterativo.getPaginaReferenciada()) {
        tabelaEntradas.remove(elementoIterativo);
        tabelaEntradas.add(elementoIterativo);
        elementoIterativo = obterElementoFIFO();
    }

    return elementoIterativo;
}
```

```java
private VirtualTableEntry obterElementoLRU() {
    List<VirtualTableEntry> tabelaProvisoria = new ArrayList<>();
    tabelaProvisoria.addAll(tabelaEntradas);

    Collections.sort(tabelaProvisoria, new Comparator<VirtualTableEntry>() {
        @Override
        public int compare(VirtualTableEntry left, VirtualTableEntry right) {
            Date leftValue = left.getUltimaUtilizacao();
            Date rightValue = right.getUltimaUtilizacao();

            return leftValue.compareTo(rightValue);
        }
    });

    return tabelaProvisoria.get(index: 0);
}
```

- Para validar seu gerenciador de memória
    - Utilize o código de teste dos trabalhos anteriores modificando os parâmetros da ferramenta e compare os resultados
    - Crie mais processos sem laços infinitos
    - Simule situações de erros que podem ser detectados pela ferramenta