Trabalho Prático 2 - Sistemas Operacionais

Eduardo Vaz Fagundes Rech - 180075161
Gabriel Henrique Souza de Melo - 180136577
Igor Laranja Borges Taquary - 180122231

Ferramentas e Linguagem

A linguagem utilizada para elaboração do trabalho prático foi TypeScript (type/node) com auxílio das bibliotecas *fs (file system)* para leitura dos arquivos textos *e path* para auxílio das referências aos arquivos.

Para execução do código é necessário o *runtime* do JavaScript (*npm, npx*), além do Node.js.

Executando os seguintes comandos no diretório raiz do projeto:

1. *$ npm install*
2. *$ npx ts-node src/index.ts processes.txt files.txt*

Descrição da Solução

Primeiro são definidas as estruturas utilizadas para a realização do projeto, as quais serão usadas para facilitar a manipulação dos dados e um código mais organizado e compreensível. Listadas nas imagens abaixo:

```ts
// Dados de um Processo executado
export type Process = {
  id: number,
  priority: number,
  process_time: number,
  operations: number
}

/// Tipos de operacao de disco
export enum OperationType {
  CREATE = 0,
  DELETE = 1
}

// <ID_Processo>, <Código_Operação>, <Nome_arquivo>, <se_operacaoCriar_numero_blocos>.
export type FileOperation = {
  process_id: number,
  type: OperationType,
  name: string,
  size?: number
}

// arquivo (a ser identificado por uma letra), número do primeiro bloco gravado, quantidade de blocos ocupados por este arquivo.
export type OldFile = {
  name: string,
  position: number,
  size: number
}

// Tipo de alocacao na memoria
// 1-contígua, 2-encadeada ou 3-indexada
export enum AllocationType {
  CONTIGUA = 1,
  ENCADEADA = 2,
  INDEXADA = 3
}
```

```ts
// Bloco de arquivo salvo no disco
export type DiskFile = {
  name: string,
  allocationType?: AllocationType,
  created_by?: number,
  nextBlock?: number // Usado apenas no caso de alocação encadeada
  blocksList?: number[] // Usado apenas no caso de aloacação indexada no bloco de índice.
}
```

A função a ser executada é a *main,* a qual obtém o nome dos dois arquivos passados como parâmetros na execução do programa e realiza a execução de outras funções, lidando com seus parâmetros.

```
// Funcao principal
function main() {
  // Checa possiveis erros
  try {

    // Parametros de execucao
    // Arquivos de processo
    const processes_file = process.argv[2]
    // Arquivo de lista de operacoes
    const files_file = process.argv[3]

    if(!processes_file || !files_file) {
      throw new Error("Missing input files");
    }

    // Carrega os processos no array no formato Process
    const processes = readProcesses(processes_file)

    // Carrega as informações do disco e os arquivos em arrays
    const { allocationType, diskSize, oldFiles, filesToCreate } = readFiles(files_file)

    const disk = initDisk(diskSize, oldFiles)

    runOperations(disk, processes, allocationType, filesToCreate)

    console.log("Success End")
  } catch (error) {
    console.log("ERROR: " + error.message || error)
  }
}
```

Em resumo, a função *main* faz a leitura dos arquivos de entrada e com as informações recebidas, inicializa os processos, cria o disco, adiciona os blocos iniciais e por último realiza as operações. Ao final da execução indica sucesso ou a existência de algum erro que não pode ser tratado.

Como dito anteriormente o primeiro passo é a manipulação dos arquivos texto usados como entrada do programa, tanto o de processos (process.txt) como o de operações de arquivos (files.txt). O objetivo destas funções é receber um arquivo texto, com a correta estrutura e quebras de linha, e retornar variáveis que possam ser manipuladas pelo nosso programa. Para este fim foram criadas as funções "*readProcesses*", para tratar o arquivo (process.txt), e a "*readFiles*", para tratar o arquivo (files.txt). Nas duas foram utilizadas as bibliotecas fs e path para resgatar o conteúdo dos arquivos.

Agora é preciso fazer a inicialização da estrutura que representará o disco bem como criar os arquivos iniciais de forma contígua, para isso é utilizada a função *initDisk*, cuja implementação pode ser vista abaixo:

```
// Inicializa o vetor que representa o disco
function initDisk(diskSize: number, initialFiles: OldFile[]) {
  const disk = new Array<DiskFile>(diskSize).fill({name:'0'})
  console.log("diskSize = " + diskSize)
  // Arquivos iniciais são alocados de forma contigua
  // Preenche o vetor do disco com os arquivos iniciais
  initialFiles.forEach( file => {
    const free_index = disk.findIndex( v => v.name === '0')
    if(free_index === -1 || (free_index+file.size) > diskSize) throw new Error("No space for initial allocation");
    disk.fill({ name: file.name, allocationType: AllocationType.CONTIGUA }, file.position, file.position+file.size)
  })

  console.log("Initial disk state: ")
  printDisk(disk)
  console.log("\n")
  return disk
}
```

Em seguida, o programa faz o gerenciamento geral dos processos de arquivos a serem executados. Primeiro é feita algumas verificações de erros previstos. Em seguida garante que o processo gaste 1 unidade de tempo de processamento e então realiza o processo, diferindo entre as implementações para as chamadas *DELETE* e *CREATE*.

```
function runOperations(disk: DiskFile[], processes: Process[], allocationType: number, operations: FileOperation[]) {

  operations.forEach( (operation, i) => {
    // Manipulacao e tratamento dos erros
    try {
      // Busca processo que realizará operação
      const processIndex = processes.findIndex( p => p.id === operation.process_id)
      // Verifica se o processo existe na lista de processos
      if(processIndex === -1)
        throw new Error(`O processo ${operation.process_id} não existe.`);
      // Verifica se o processo ainda pode executar operações
      if(processes[processIndex].process_time <= 0)
        throw new Error(`O processo ${operation.process_id} já encerrou a sua execução.`);

      // Processo executará operação (-1 tempo de processamento)
      processes[processIndex].process_time = processes[processIndex].process_time - 1;

      if(operation.type === OperationType.DELETE) {
        // Operação de deleção de arquivo
        disk = deleteFile(disk, processes[processIndex], operation)
        printOperationResult(i, true, `Arquivo ${operation.name} deletado pelo processo ${operation.process_id}`)
      } else if (operation.type === OperationType.CREATE) {
        // Operação de criação de arquivo
        disk = createFile(disk, allocationType, processes[processIndex], operation)
        printOperationResult(i, true, `Arquivo ${operation.name} criado pelo processo ${operation.process_id}`)
      } else {
        throw new Error("Tipo de operação desconhecida");
      }
    } catch (error) {
      // Informa o erro no monitor
      printOperationResult(i, false, error)
    }
    printDisk(disk)
    console.log(".")
  })

}
```

A função para criação de arquivo depende de qual tipo de alocação o disco está usando, porém antes de fazer as alocações é feita uma verificação se o disco tem o espaço necessário para alocar o arquivo.

```
export function createFile(disk: DiskFile[], allocationType: AllocationType, process: Process, operation: FileOperation) {
  // Obtem os espaços livres no disco
  const freeDiskSpaces = getFreeDiskSpaces(disk)
  // Verifica qual o tipo de alocacao do arquivo
  switch (allocationType) {
    case AllocationType.CONTIGUA:
      // Verifica se tem espaço continuo disponivel
      const availableIndex = findObjectValueGte( freeDiskSpaces, operation.size )
      if(availableIndex === undefined)
        throw new Error("Falta de espaço em disco para alocação contigua.");
      return contiguousCreate(disk, operation, availableIndex);

    case AllocationType.ENCADEADA:
      // Verifica se tem espaço total disponivel
      if( operation.size > sumObjectValues( freeDiskSpaces ) )
        throw new Error("Falta de espaço em disco");

      return chainedCreate(disk, operation);

    case AllocationType.INDEXADA:
      // Verifica se tem espaço disponivel
      if( operation.size > sumObjectValues( freeDiskSpaces ) )
        throw new Error("Falta de espaço em disco");

      return indexedCreate(disk, operation);

    default:
      throw new Error("Tipo de alocação do arquivo não conhecida");

  }
}
```

As seguintes imagens mostram as implementações dos processos de criação separados pelo tipo de alocação.

A primeira função é para a alocação contígua, em seguida a alocação encadeada e por último a alocação indexada.

```
function contiguousCreate(disk: DiskFile[], operation: FileOperation, firstIndex: string) {
  for (let index = Number(firstIndex); index < (Number(firstIndex) + operation.size); index++) {
    disk[index] = {
      name: operation.name,
      allocationType: AllocationType.CONTIGUA,
      created_by: operation.process_id
    }
  }
  return disk
}

function chainedCreate(disk: DiskFile[], operation: FileOperation) {
  // Tamanho final do arquivo (somado com os 10%)
  const total_size = operation.size + Math.ceil(operation.size / 10);
  for (let block = 0; block < total_size; block++) {
    const freeIndex = disk.findIndex( b => b.name === "0");
    const nextFreeIndex = block >= total_size - 1 ?
      undefined : disk.findIndex( (b,i) => (b.name === "0" && i > freeIndex ))

    disk[freeIndex] = {
      name: operation.name,
      created_by: operation.process_id,
      allocationType: AllocationType.ENCADEADA,
      nextBlock: nextFreeIndex
    }
  }
  return disk
}
```

```
}

// Cria arquivo de alocacao indexada
function indexedCreate(disk: DiskFile[], operation: FileOperation) {
  // Busca primeiro bloco livre
  const firstBlockIndex = disk.findIndex( b => b.name === "0");
  // Cria bloco de indice
  disk[firstBlockIndex] = {
    name: operation.name + "I",
    allocationType: AllocationType.INDEXADA,
    created_by: operation.process_id,
    blocksList: []
  }

  const blocksList = []
  // Cria blocos de dados
  for (let i = 0; i < operation.size; i++) {
    const freeIndex = disk.findIndex( b => b.name === "0");
    disk[freeIndex] = {
      name: operation.name,
      created_by: operation.process_id,
      allocationType: AllocationType.INDEXADA,
    }
    blocksList.push(freeIndex)
  }

  // Salva endereços dos blocos de dados no bloco de indice
  disk[firstBlockIndex].blocksList = blocksList
  return disk
}
```

Para a remoção de arquivo primeiro é verificado se o arquivo foi achado no disco, em seguida verifica se o processo tem permissão para realizar a ação de deletar o arquivo em questão. Então é chamada a função para sua respectiva forma de alocação.

As seguintes imagens mostram as implementações dos processos de remoção separados pelo tipo de alocação.

A primeira função é para a alocação contígua, em seguida a alocação encadeada e por último a alocação indexada.

```
// Faz a delecao contigua
function contiguousDelete(disk: DiskFile[], firstIndex: number, filename: string) {
  for (let i = firstIndex; disk[i].name === filename; i++) {
    disk[i] = { name: "0" }
  }
  return disk
}

// Faz a delecao encadeada
function chainedDelete(disk: DiskFile[], firstIndex: number) {
  let i = firstIndex
  while (i !== undefined) {
    const aux_i = i
    i = disk[i].nextBlock
    disk[aux_i] = { name: "0" }
  }
  return disk
}

// Faz a delecao indexada
function indexedDelete(disk: DiskFile[], firstIndex: number) {
  const index_block = disk[firstIndex]
  if(index_block.name.charAt(1) !== "I")
    throw new Error("Não é um bloco de indice");

  if(!index_block.blocksList)
    throw new Error("Bloco de indice não possui lista de blocos");

  // Apagar blocos de dados
  index_block.blocksList.forEach((block) => {
    disk[block] = { name: "0" }
  })

  // Apagar bloco de indice
  disk[firstIndex] = { name: "0" }

  return disk
}
```

Principais Dificuldades

A priori, o projeto seria desenvolvido na linguagem C++, porém na fase de planejamento foi notada algumas dificuldades que seriam enfrentadas durante o desenvolvimento. Problemas principalmente para manipulação de estruturas de dados, por isso a adoção da linguagem TypeScript que facilita bastante a criação e manipulação de estruturas.

Outra dificuldade foi interromper uma função ao encontrar uma falha para que não aconteça uma alteração parcial no disco e atrapalhe todo o processo. A solução encontrada foi trabalhar com tratamento de exceção disponibilizado pela linguagem, que interrompe uma função e retorna o comportamento indevido.

Funções dos Membros

- Eduardo Vaz Fagundes Rech: Funções de manipulação de arquivos .txt e variáveis de gerência do disco.
- Gabriel Henrique Souza de Melo: Função de inicialização do disco, criação e deleção por alocação contígua.
- Igor Laranja Borges Taquary: Funções de criação e deleção por alocação indexada e encadeada.

Bibliografia

[1] Stalling, W. Operating Systems Internals and Design Principles, Pearson Education, 2009.

[2] Martin, R. C. Clean Code. A Handbook of Agile Software Craftsmanship, Prentice Hall, 2008.