File System Simulator

João Marco Maciel da Silva 7577598
Renan Fichberg 7991131

Decisões de Projeto

Manipulação do Arquivo Binário

Para manipular o arquivo binário, tivemos de pensar em um *layout* para guardar as informações e recuperá-las em um momento futuro. Tal *layout* está descrito no arquivo *headers/partition.h*.

Basicamente, grande parte das definições daquele arquivo são para manipular o ponteiro da *stream* de um FILE (*stdlib.h*) usando a função *fseek*.

A seguir, uma breve explicação da ideia central do layout

Layout da Partição

As posições de caracteres de [0, 76000[(primeiros 76kB do arquivo) são reservadas para o Superbloco, o *bitmap* (para gerenciamento de memória), a tabela de alocação de arquivos (FAT), o nome do root ("/") e as datas de criação, modificação e acesso do *root* (para um sistema de arquivos que não é novo, a sua data de criação original, bem como as outras duas datas, são mantidas do jeito que foram deixadas. Isso vale para os arquivos e diretórios do usuário, também).

Layout da Partição

Depois disso, os *bytes* das posições [76000, 100000000 (= 100MB)[são reservados para diretórios e arquivos do usuário.

Com relação a tais diretórios e arquivos, os seus primeiros 1100 *bytes* [0, 1100[são reservados para metadados do arquivos e os últimos 2900 *bytes* são reservados para dados (no caso de arquivos texto).

Cada bloco do arquivo binário, por definição, tem portanto, um tamanho 1100 + 2900 = 4kB e quando um certo arquivo não cabe em apenas um bloco, ele usa blocos adicionais (e através das suas informações do FAT, guardadas nos seus metadados junto das informações do FAT guardadas nos primeiros 76kB do arquivo binário, conseguimos recuperar o sistema de arquivos no estado em que ele foi deixado pela última vez).

Outras Decisões

I) Os diretórios, apesar de não terem nos seus metadados um campo reservado para "tamanho", assim como os arquivos de texto vazios, eles também são guardados no bitmap e no FAT, ocupando apenas UM índice de ambas as tabelas.

II) Apesar dos arquivos de texto e dos diretórios terem esta mesma característica descrita em (I), os 2900 bytes da área de dados de um diretório não são considerados como desperdiçados, ao contrário de um arquivo de texto vazio. Ou seja, o comando 'df' para um sistema de arquivos só com um diretório apontará zero bytes desperdiçados enquanto um outro sistema arquivos, só com um arquivo de texto vazio (criado com o comando touch) na raiz terá 2900 bytes apontados como desperdiçados ao rodar o mesmo comando.

Outras Decisões

III) Ao iniciar um novo arquivo binário, um caractere '\n' é colocado na posição 100000000 do arquivo, portanto, os arquivos na realidade tem 100MB + 1B.

Este caractere serve apenas para que o sistema de arquivos mostre sempre a sua capacidade de 100MB.

O tanto que de fato está sendo usado pode ser visto usando o comando 'df', já mencionado anteriormente.

Tal decisão de projeto foi escolhida para melhorar a manipulação do arquivo (e não correr riscos de receber erros de barramento (SIGBUS)), considerando casos em que podem ter "blocos esparsos" (= blocos desalocados entre blocos alocados).

Outras Decisões

Este caractere é inacessível e existirá no arquivo binário até que o mesmo seja deletado.

A performance da leitura de arquivo não é afetada por isso (o programa sabe quando já leu tudo o que deveria. Um arquivo não é lido sempre até encontrar este caracter final).

Ainda, esta implementação é a que mais pareceu fazer sentido, imaginando que dispositivos móveis como pendrives, mídias como CDs e até mesmos discos rígidos tem tamanho fixo.

Comando extra time

Comando para auxiliar os testes;

Funciona de maneira diferente do time do Linux;

$time 0 marca o tempo inicial;

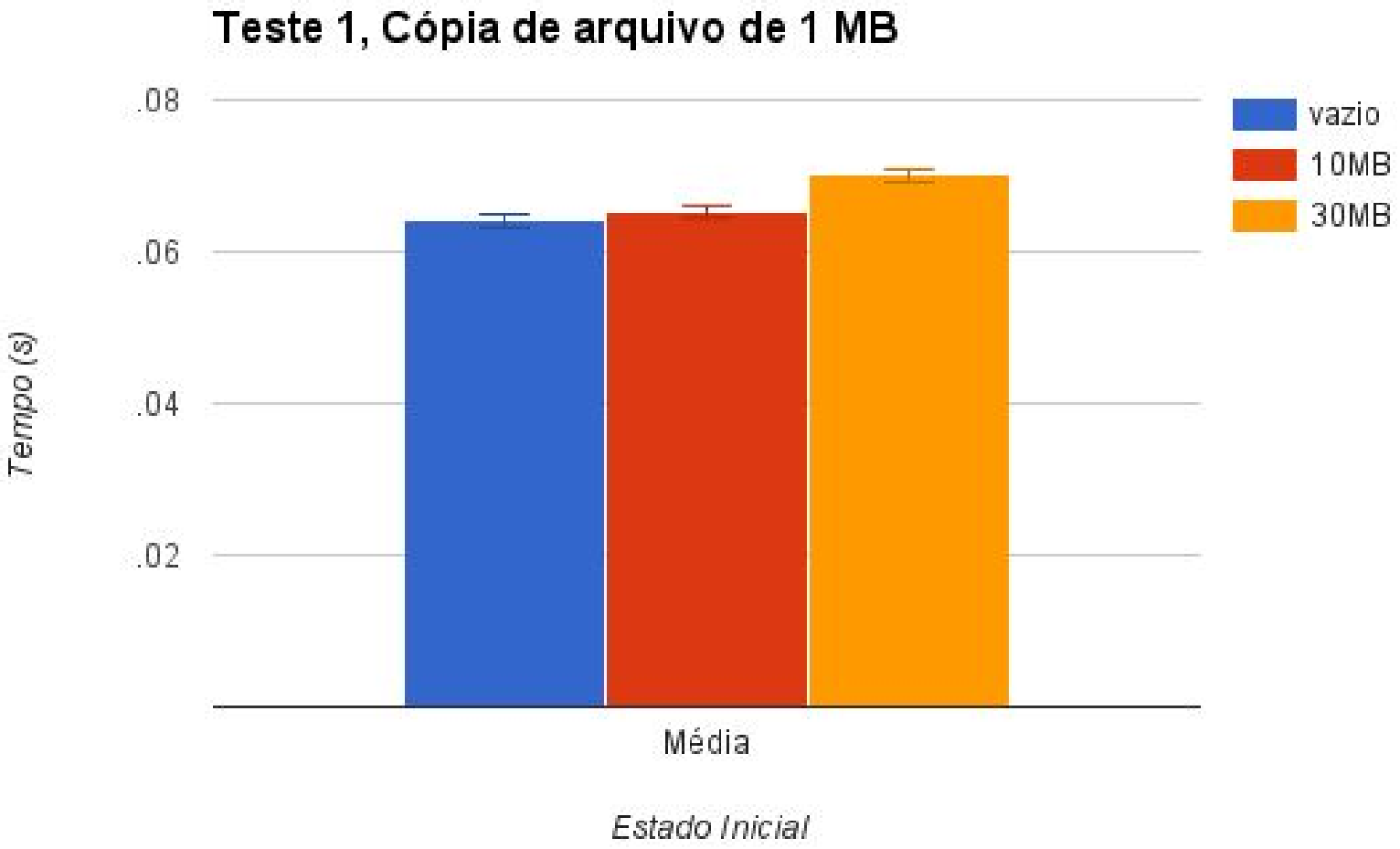
$time 1 imprime na saida de erro o tempo decorrido e atualiza o tempo atual;

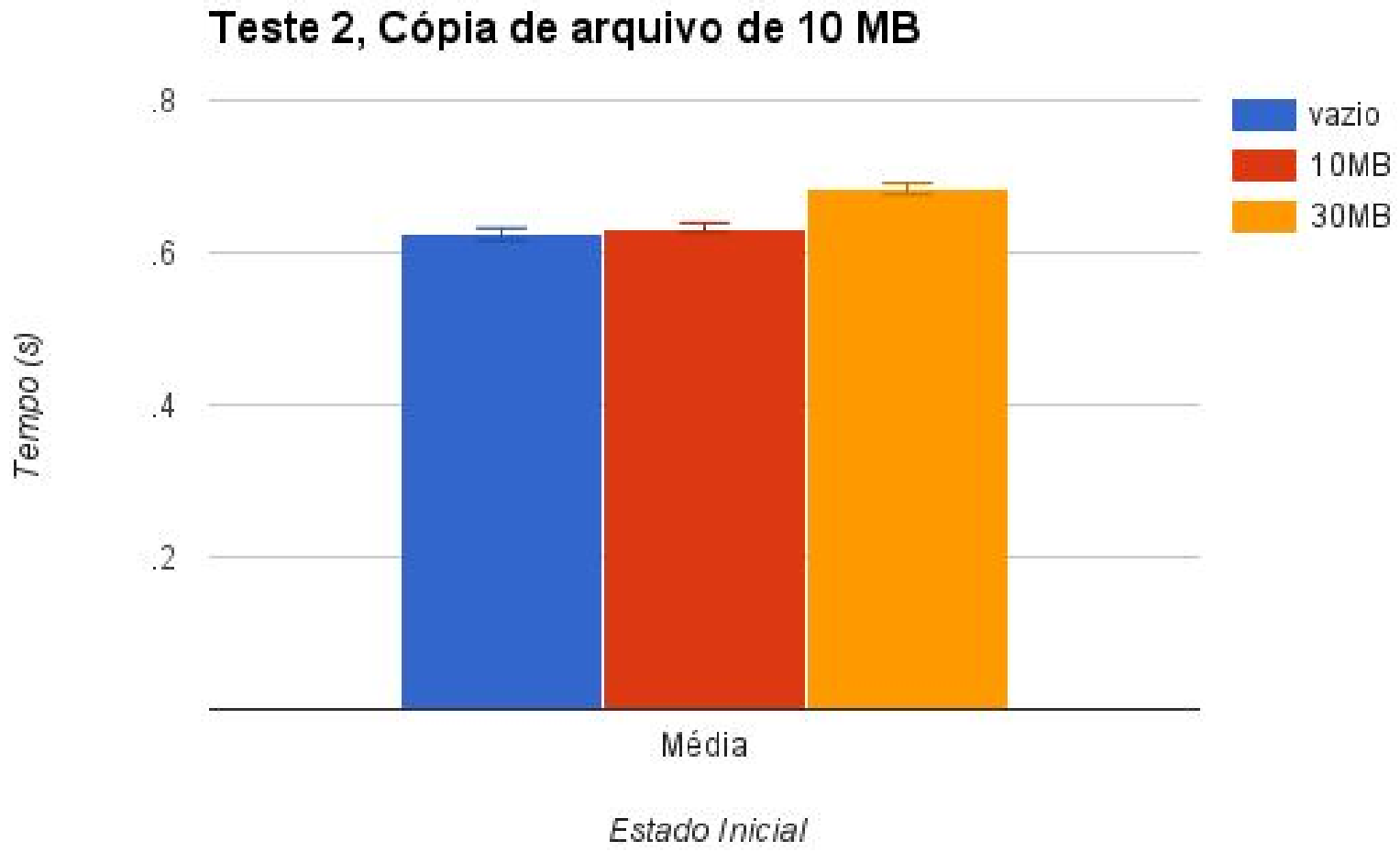
A saída de erro foi utilizada para facilitar separar das outras partes impressas pelo EP.

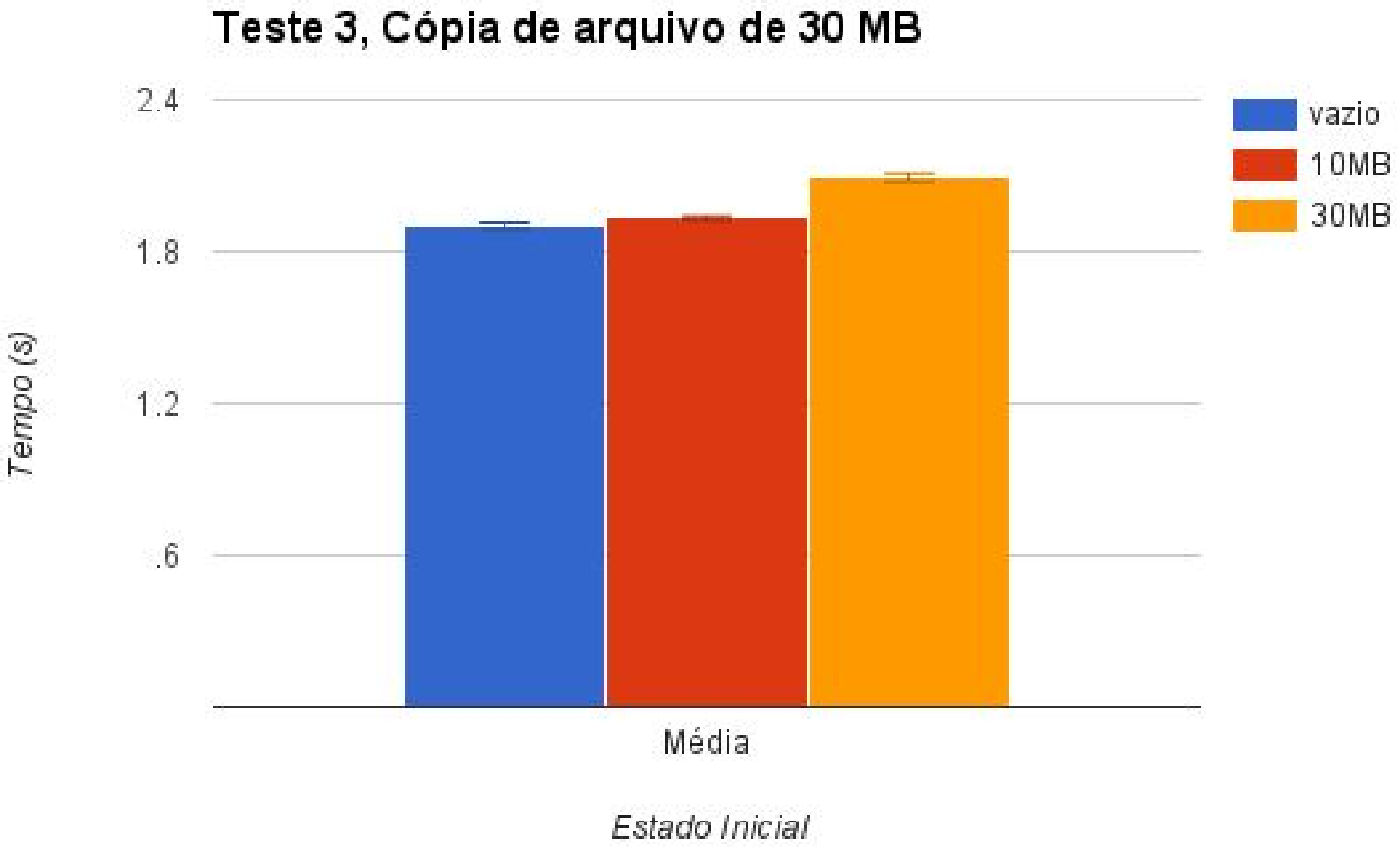
Testes e Resultados

Máquina Utilizada

- SO: Xubuntu Linux 14.04 x86_64
- Processador: Intel Core (TM)2 Duo T8300
- Processadores: 2
- Clock: 2.4GHz
- Cache L1: 64kB
- Cache L2: 3072kB
- RAM: 2x2GB DDR2


Teste 1, Cópia de arquivo de 1 MB
.08
.06
.04
.02
Tempo (s)
vazio
10MB
30MB
Média
Estado Inicial

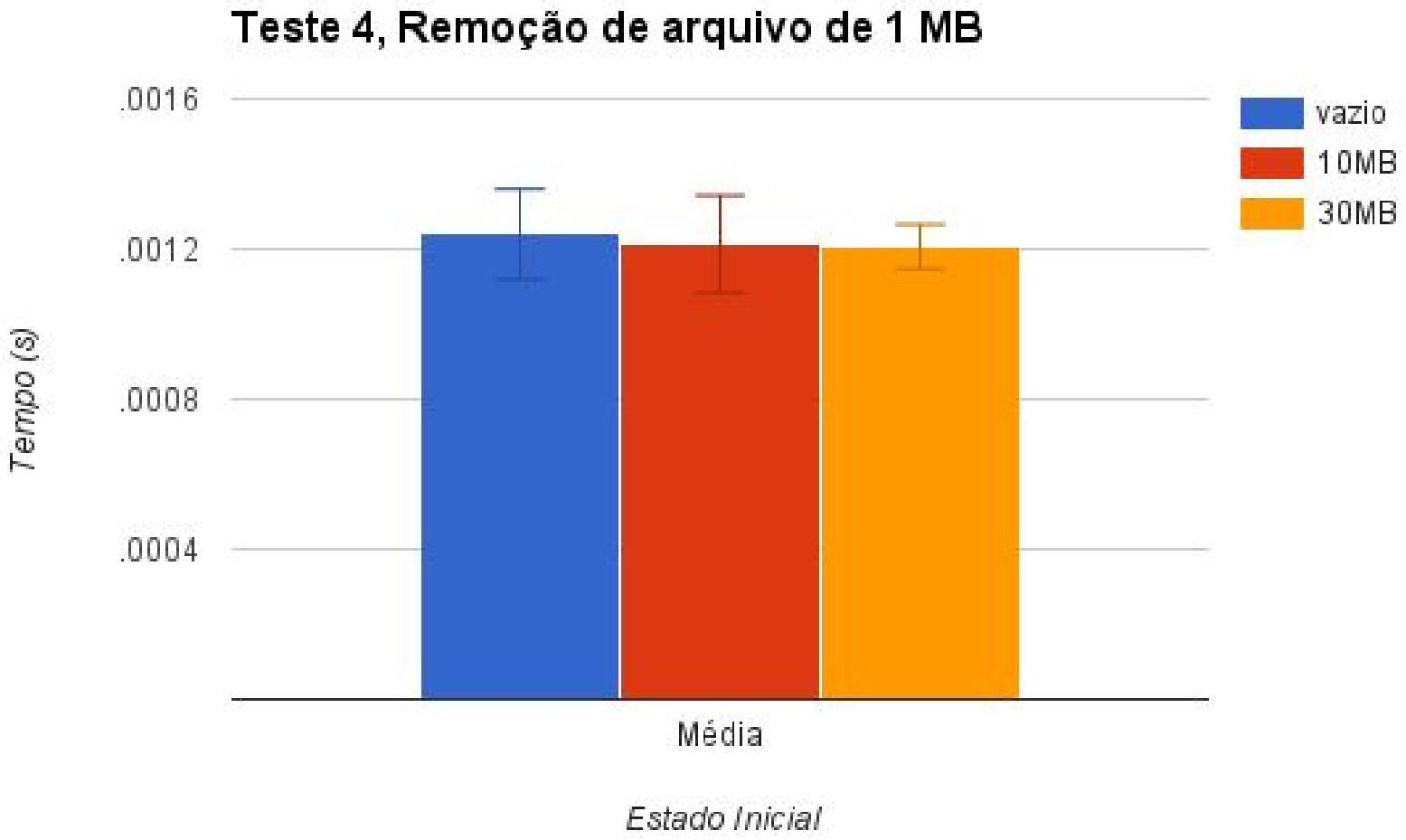

Teste 2, Cópia de arquivo de 10 MB
.8
.6
.4
.2
Tempo (s)
vazio
10MB
30MB
Média
Estado Inicial

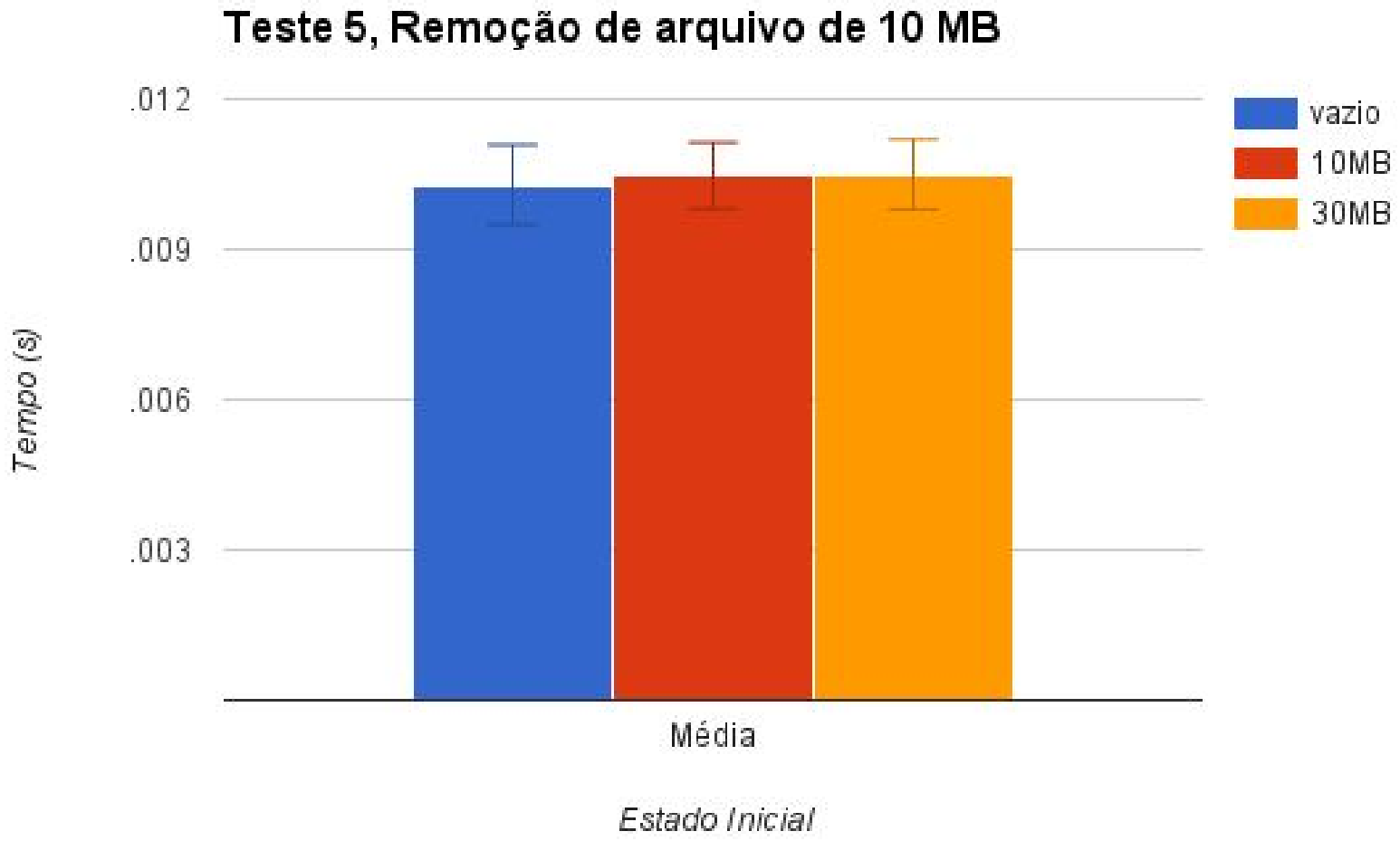

Teste 3, Cópia de arquivo de 30 MB
2.4
1.8
1.2
.6
Tempo (s)
vazio
10MB
30MB
Média
Estado Inicial

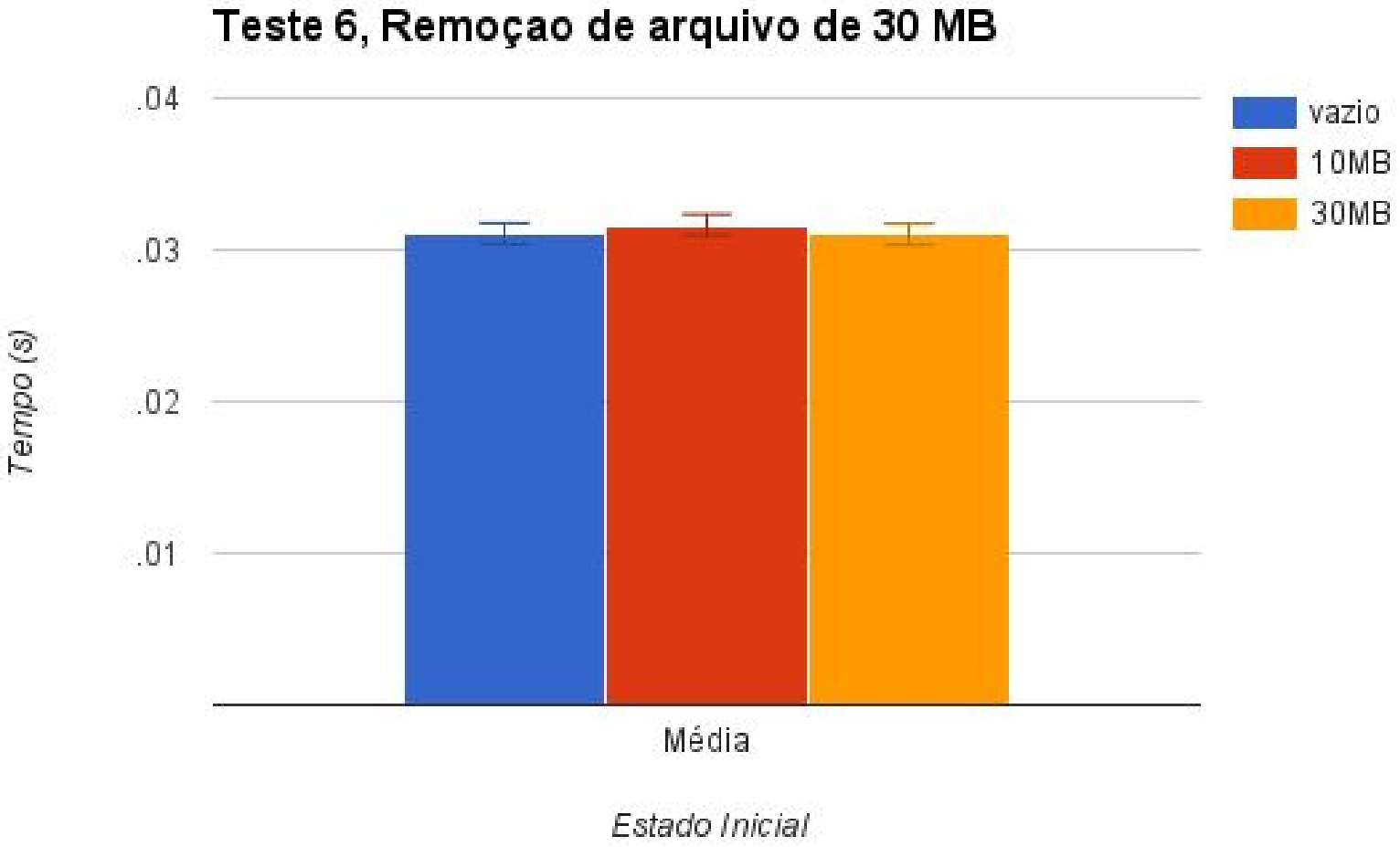
Testes 1 a 3

Os resultados desses 3 experimentos mostram que:

- Para gravar no disco quanto mais cheio mais ele demora;
- Quanto maior o arquivo maior o tempo para gravar e esse tempo aumenta quase linearmente para estes 3 valores.


Teste 4, Remoção de arquivo de 1 MB
.0016
.0012
.0008
.0004
Tempo (s)
vazio
10MB
30MB
Média
Estado Inicial

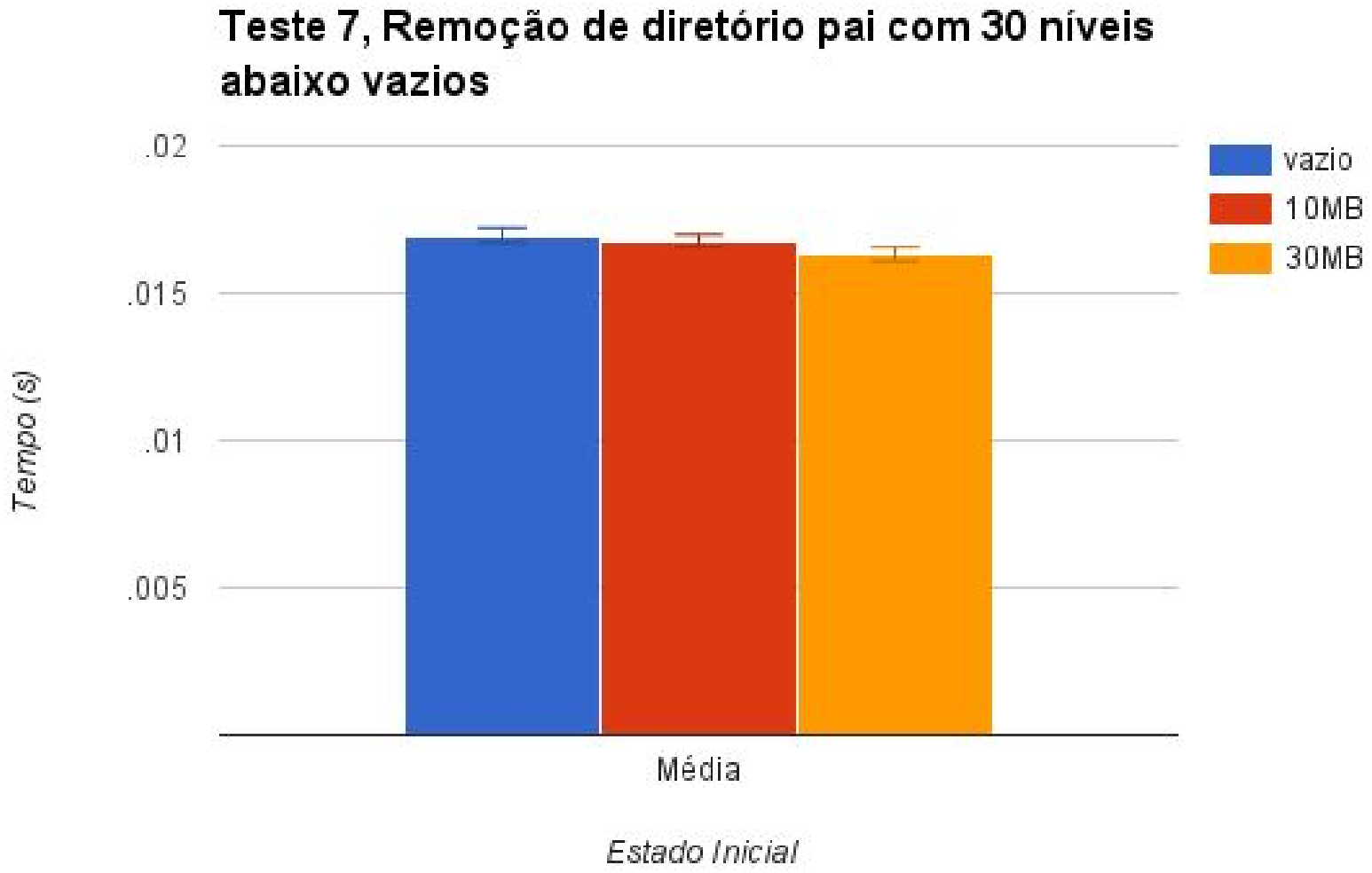

Teste 5, Remoção de arquivo de 10 MB
.012
.009
.006
.003
Tempo (s)
vazio
10MB
30MB
Média
Estado Inicial

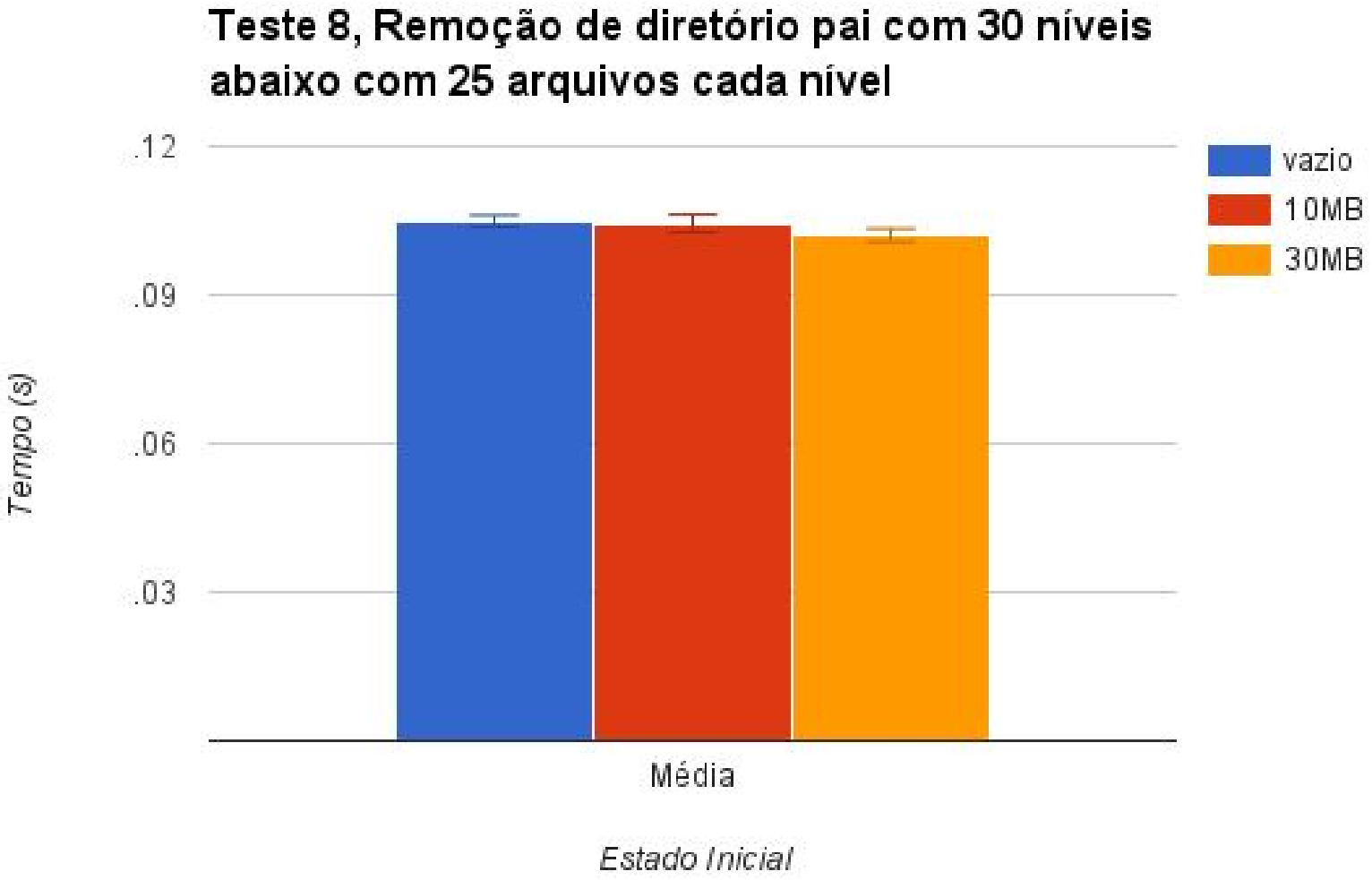

Teste 6, Remoçao de arquivo de 30 MB
.04
.03
.02
.01
Tempo (s)
vazio
10MB
30MB
Média
Estado Inicial

Testes 4 a 6

Os resultados desses 3 experimentos mostram que:

- Para apagar do disco o quão cheio está não interfere de maneira substancial;
- Quanto maior o arquivo maior o tempo para apagar e esse tempo aumenta quase linearmente para estes 3 valores.


Teste 7, Remoção de diretório pai com 30 níveis abaixo vazios
.02
.015
.01
.005
Tempo (s)
vazio
10MB
30MB
Média
Estado Inicial


Teste 8, Remoção de diretório pai com 30 níveis abaixo com 25 arquivos cada nível
.12
.09
.06
.03
Tempo (s)
vazio
10MB
30MB
Média
Estado Inicial

Testes 7 e 8

Os resultados desses 2 experimentos mostram que:

- Surpreendentemente, para apagar do disco o quão cheio está interfere de maneira inversamente proporcional quando se trata de apagar em cadeia.

Problemas nos testes

Testar com centenas de arquivos foi especificamente trabalhoso.

Criamos um *script* que faz várias execuções do programa, rodando vários comandos consecutivamente, mas tínhamos erros em chamadas a *fseek* que poderiam ocorrer em qualquer ponto do código.

Uma pesquisa um pouco mais aprofundada nos revelou que não é um problema do nosso código em si, mas sim porque um processo estava chamando a função *fopen* vezes demais e, por segurança, em determinado momento o sistema operacional não deixava mais ele abrir um arquivo, carregando um ponteiro nulo para FILE para dentro da função *fseek*, resultando em SIGSIEV naquele trecho de código.

Problemas nos testes

A solução encontrada para testar foi criar baterias de teste menores e rodar novas instancias do EP separadamente, carregando o binário do ponto em que tinha sido deixado na última execução, assim cada nova *run* do EP3 seria um novo processo e o total de *fopen*'s que poderíamos dar seria zerado, dado que é um novo processo.

Para descobrir que este era o problema, apenas usamos o auxílio da biblioteca *<errno.h>**, que retornava toda hora o *erro 24*. Abaixo a linha do *header* da *errno.h* que é retornada no EP caso a execução extrapole no limite de chamadas a *fopen*.

"#define EMFILE 24 /* Too many open files */"

Problemas nos testes

* Obs: a versão que está sendo entregue não tem essa modificação do código por não haver necessidade, uma vez que já estamos escrevendo aqui o que deve ser feito para evitar o *crash*, caso o usuário decida rodar o EP3 com um *script* que vai fazer milhares de chamadas a *fopen* num mesmo processo)

Obs2: outro possível *workaround* (não testado por nós) para este problema é mexer nas configurações de segurança do sistema operacional (aqui, estamos apenas falando de UNIX), no arquivo *sysctl* e mudar o número limite de arquivos que um processo iniciado pelo seu usuário pode abrir. Por não ter sido testado por nós, qualquer um que tentar essa aproximação está fazendo por conta própria e nós não nos responsabilizamos pelo resultado.