

Instituto de Ciências Matemáticas e de Computação  
Universidade de São Paulo

# Trabalho de Sistemas Operacionais

Código da Disciplina: SSC5723  
Professor: Dr. Julio César Estrella

Estevam Fernandes Arantes – 9763105  
Murilo Cunha dos Santos – 11925875

São Carlos  
2020

# Resumo

Para o desenvolvimento dos módulos presentes na disciplina de sistemas operacionais foram utilizados diversas estruturas de dados e vários conteúdos foram abordados. As linguagens utilizadas durante o desenvolvimento foram C, C++ e Python e ao total foram desenvolvidos quatro módulos durante o curso. Assim, a primeira parte foi responsável por introduzir os alunos a conceitos relacionados a chamadas de sistemas de um sistema operacional. As chamadas de sistema deveriam ser relacionadas a Arquivo, Memória ou Processos. Então, para separar estes assuntos foram desenvolvidos três programas distintos e em cada programa contém três primitivas para cada tipo de chamada. E também foi realizado nos algoritmos que permitissem exibir o comportamento de cada execução informando se aquele processo era CPU ou I/O Bound. Para este módulo apenas a linguagem C foi utilizada devido a fornecer estruturas para realizar chamadas de sistemas. O segundo módulo foi um pouco mais elaborado devido à necessidade de utilização de semáforos e uma estrutura de dados que fornecia uma fila de endereços para inserção de dados em um buffer. Este módulo teve como objetivo resolver o problema de produtor e consumidor que se resume em um processo produtor inserir dados em um buffer e o processo consumidor retirar os dados do buffer, porém alguns problemas como o processo produtor não poder inserir dados em um buffer cheio e nem o processo consumidor retirar dados em um buffer vazio. A maior dificuldade deste módulo encontrada foi em relação a que devido a um erro de programação acabava gerando um deadlock e o processo não executava. Então, para solucionar o problema foi a utilização de três semáforos para controle sendo um para verificar se o buffer está cheio, um para verificar se o buffer está vazio e um para garantir que os processos produtor e consumidor não acessem o buffer ao mesmo tempo, ou seja, permite a exclusão mútua. Para o terceiro módulo envolvia o conceito de gerenciamento de memória virtual com paginação. Portanto, o desenvolvimento consistia em desenvolver um simulador para verificar as trocas de páginas entre as memórias real e virtual acessadas a partir de um arquivo que contém as informações de cada processo, sendo a sua criação, sua instrução e o tipo do processo sendo CPU ou I/O Bound. Este módulo necessitou a troca de linguagem para a C++ devido à necessidade de utilização de muitas estruturas de dados o que ocasionaria em muito tempo de desenvolvimento, e a linguagem C++ pouparia tempo. Para realizar o gerenciamento da memória foram desenvolvidos algoritmos para a configuração da paginação, que envolve por exemplo a quantidade de página, o tamanho de cada página, os tamanhos das memórias reais e virtuais e o tipo de algoritmo utilizado para realizar as trocas de páginas da memória virtual para real, e também foi desenvolvido um algoritmo que fosse responsável para a manipulação das memórias. Assim, na manipulação das memórias algumas dificuldades encontradas foram na simulação da estrutura de uma tabela de memória, em conseguir acessar devidamente a região desejada. O algoritmo também contém estruturas para acessar as memórias, remover as páginas e verificar se o processo já foi mapeado na memória. Por fim, foi a elaboração do módulo quatro que consistia na comparação de alguns tipos de sistemas de arquivos avaliando seu desempenho em relação ao tempo e foram analisadas algumas operações de arquivos. Então, para cada tipo de operação foi desenvolvido um arquivo diferente com um tamanho de arquivo diferente. Este módulo utilizou a linguagem C e também Python para elaborar os gráficos de comparação para cada tamanho de arquivo e também uma comparação entre os sistemas de arquivos. O que mais gerou preocupação foi em relação a função que calcula os tempos para as comparações conseguiria se manter eficiente, então como medida para garantir foi utilizado o tempo com base no clock do processador para assim, permitir uma assertividade maior.

# Relatório Módulo 1

Para este módulo foram desenvolvidos três programas distintos, cada qual com o seu foco entre chamadas de sistema relacionadas a Arquivo, Memória ou Processos. Serão escolhidas ao menos 3 primitivas para cada uma das categorias e estas serão analisadas com relação ao tempo de execução em modo de Kernel e de usuário, à quantidade de trocas de contexto e à quantidade de vezes que ela é chamada em cada um dos programas.

## Instruções Para Execução dos Códigos

Foi criado um script `run.sh`, que se encontra na pasta `Módulo 1` deste repositório. Ao rodar este programa ele irá compilar os diferentes códigos utilizando o compilador `gcc` e irá executar os comandos `strace` e `time`. É necessário que os 3 estejam instalados, o que é o padrão em uma distribuição como o Ubuntu 18.04 LTS.

É importante clonar todo o repositório, dado que para o programa relativo aos Arquivos, por exemplo, é necessário o uso de 2 arquivos de texto adicionais, já presentes no Github.

## Identificação e Descrição das Chamadas de Sistema

Por meio do comando `strace` são identificadas as chamadas de sistemas nas tabelas abaixo, referentes às primitivas de memória, processos e arquivos, respectivamente. As suas breves descrições estão nestas mesmas tabelas.

### Chamadas de Memória

syscall	descrição
brk	desloca o fim do segmento de dados para um valor especificado, desde que o tamanho do processo não exceda o máximo permitido
mmap	cria um novo mapeamento no espaço de endereçamento virtual do processo chamado
munmap	deleta o mapeamento feito em um intervalo de endereços dado. É feito automaticamente quando o processo termina
mprotect	Define uma proteção para uma dada região de memória, como read only, write, etc.
time	Retorna o tempo em segundos passados desde 01/01/1970. Caso não seja nulo, o valor de retorno é guardado no valor da memória passado pelo parâmetro.

### Chamadas de Processos

syscall	descrição
execve	Executa o programa apontado no parâmetro, o qual deve ser um executável binário ou um script.
kill	Envia um sinal para o processo ou grupo especificado. No caso utilizado é enviado o sinal SIGKILL, que o mata.
clone	Cria um novo processo. É utilizado pelo comando fork do C. Ao contrário da syscall fork ele permite que o processo divida partes da sua execução com o seu pai.
getpid	Retorna o identificador (ID) do processo.
arch prctl	Define o processo ou thread específico de determinada arquitetura, selecionando uma subfunção e passando como argumento para a chamada de sistema.

## Chamadas de E/S e Arquivos

syscall	descrição
access	Checa se o processo tem acesso requisitado ao arquivo dado. É feita utilizando o UID e GID real do processo, ao invés do ID virtual, assim determinando com facilidade a autoridade do usuário.
read	Lê um número de bytes, passados por parâmetro, de um arquivo e armazena em um buffer.
write	Funciona de forma semelhante ao read, mas escrevendo até o número de bytes passados pelo parâmetro com base nos dados do buffer.
close	Fecha um dado descritor de arquivo aberto anteriormente, assim fazendo com que ele possa ser reutilizado e qualquer ligação que o arquivo com, ou possuída pelo processo, são removidas.
fstat	Retorna o status de um dado arquivo, sendo esse status informações sobre o arquivo, como permissão, id, inode, tamanho do bloco, grupo, etc.
lseek	Reposiciona o offset (em bytes) do arquivo para leitura/escrita, podendo ser a partir de um certo local especificado, o tamanho do arquivo ou a referência zero.
openat	Abre e, se necessário, cria um arquivo em um endereço dado. Podendo ser utilizado, portanto, em chamadas ao sistema subsequentes.

## Tempo e Frequência de Chamadas ao Sistema

Por meio do comando ``strace -c`` é possível obter as seguintes saídas, que mostram o tempo de cada chamada ao sistema e a sua frequência em cada um dos programas.

Resultado do ``strace`` para o programa relativo à Memória

% time	seconds	usecs/call	calls	errors	syscall
100.00	0.191071	38	5005		brk
0.00	0.000000	0	1		read
0.00	0.000000	0	2		close
0.00	0.000000	0	2		fstat
0.00	0.000000	0	7		mmap
0.00	0.000000	0	3		mprotect
0.00	0.000000	0	1		munmap
0.00	0.000000	0	6		pread64
0.00	0.000000	0	1	1	access
0.00	0.000000	0	1		execve
0.00	0.000000	0	2	1	arch_prctl
0.00	0.000000	0	2		openat
100.00	0.191071		5033	2	total

Resultado do ``strace`` para o programa relativo aos Arquivos

% time	seconds	usecs/call	calls	errors	syscall
37.68	0.028704	142	202		openat
24.58	0.018729	92	202		close
14.74	0.011230	55	202		fstat
10.16	0.007744	76	101		read
7.56	0.005763	57	100		write
4.21	0.003204	32	100		lseek
0.32	0.000241	34	7		mmap
0.28	0.000211	35	6		pread64
0.15	0.000113	37	3		brk
0.14	0.000105	35	3		mprotect
0.09	0.000065	32	2	1	arch_prctl
0.05	0.000040	40	1		munmap
0.05	0.000037	37	1	1	access
0.00	0.000000	0	1		execve
100.00	0.076186		931	2	total

Resultado do `strace` para o programa relativo aos Processos

% time	seconds	usecs/call	calls	errors	syscall
71.57	0.117020	117	1000		clone
28.43	0.046474	46	1000		kill
0.00	0.000000	0	1		read
0.00	0.000000	0	2		close
0.00	0.000000	0	2		fstat
0.00	0.000000	0	7		mmap
0.00	0.000000	0	3		mprotect
0.00	0.000000	0	1		munmap
0.00	0.000000	0	1		brk
0.00	0.000000	0	6		pread64
0.00	0.000000	0	1	1	access
0.00	0.000000	0	1		getpid
0.00	0.000000	0	1		execve
0.00	0.000000	0	2	1	arch_prctl
0.00	0.000000	0	2		openat
100.00	0.163494		2030	2	total

Como é possível ver pela saída dos comandos de fato tem-se as primitivas analisadas anteriormente.

## Análise de Tempo de Execução

Utilizando o comando `time`, juntamente com a flag `-f` para a especificação dos parâmetros adicionais solicitados, tem-se as saídas nas imagens abaixo

```
$ /usr/bin/time -f "Tempo total: %e  
Percentual de uso de CPU: %P  
Tempo em modo de Kernel: %S  
Tempo em modo de usuário: %U  
Trocas de contexto involuntárias: %c  
Trocas de contexto voluntárias: %w" ./a.out
```

Resultado do comando `time` para o programa relativo à memória

```
Tempo Memória  
Tempo total: 2.65  
Percentual de uso de CPU: 99%  
Tempo em modo de Kernel: 0.32  
Tempo em modo de usuário: 2.32  
Trocas de contexto involuntárias: 3  
Trocas de contexto voluntárias: 19
```

Resultado do comando `time` para o programa relativo aos arquivos

```
Tempo Arquivos  
Tempo total: 0.91  
Percentual de uso de CPU: 6%  
Tempo em modo de Kernel: 0.03  
Tempo em modo de usuário: 0.03  
Trocas de contexto involuntárias: 0  
Trocas de contexto voluntárias: 1716
```

Resultado do comando `time` para o programa relativo aos processos

```
Tempo Processos  
Tempo total: 0.12  
Percentual de uso de CPU: 97%  
Tempo em modo de Kernel: 0.12  
Tempo em modo de usuário: 0.00  
Trocas de contexto involuntárias: 1  
Trocas de contexto voluntárias: 19
```

## Análise CPU/I-O Bound

Como mostrado pelas saídas do comando `time` da sessão anterior, os programas possuem características que os distinguem facilmente.

Um exemplo claro é a comparação entre os programas referente às primitivas de Memória e o programa referente às primitivas de Arquivos. Enquanto o primeiro possui alta porcentagem de uso de CPU (99%), o segundo possui este mesmo percentual valendo 6%.

Por meio desta comparação, já seria possível ter fortes indícios que o processo referente aos Arquivos é I/O Bound enquanto o de Memória é CPU-bound.

Outra estatística que corrobora para esta afirmação é o número de trocas de contexto voluntárias do processo de arquivos, que é alto, enquanto no processo de memória este número é baixo.

Esperaria-se que no caso de memória, por serem operações demoradas, haveria um maior número de trocas involuntárias, porém isto não ocorreu por haver poucos processos competindo pela CPU no momento em que este foi executado.

## Relatório Módulo 2

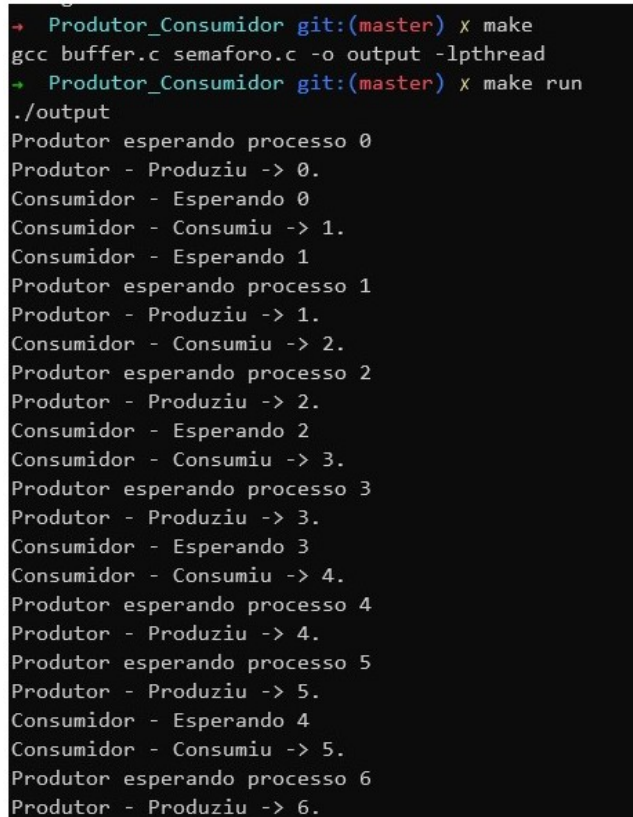
Para este módulo foi desenvolvido um programa principal que exemplifica o problema do Produtor x Consumidor, que, por meio do uso de semáforos, controla o acesso a um buffer compartilhado, onde itens serão adicionados e consumidos por diferentes threads da biblioteca pthreads, conforme a especificação.

### Instruções Para Execução do Código

Para compilar o código basta utilizar o comando `make` dentro da pasta `Produtor\_Consumidor`. Isto gerará um arquivo `output` que pode ser executado com `./output` ou `make run`.

### Exemplo de execução

Um exemplo de compilação e execução do código encontra-se na imagem abaixo.



```
➔ Produtor_Consumidor git:(master) x make
gcc buffer.c semaforo.c -o output -lpthread
➔ Produtor_Consumidor git:(master) x make run
./output
Produtor esperando processo 0
Produtor - Produziu -> 0.
Consumidor - Esperando 0
Consumidor - Consumiu -> 1.
Consumidor - Esperando 1
Produtor esperando processo 1
Produtor - Produziu -> 1.
Consumidor - Consumiu -> 2.
Produtor esperando processo 2
Produtor - Produziu -> 2.
Consumidor - Esperando 2
Consumidor - Consumiu -> 3.
Produtor esperando processo 3
Produtor - Produziu -> 3.
Consumidor - Esperando 3
Consumidor - Consumiu -> 4.
Produtor esperando processo 4
Produtor - Produziu -> 4.
Produtor esperando processo 5
Produtor - Produziu -> 5.
Consumidor - Esperando 4
Consumidor - Consumiu -> 5.
Produtor esperando processo 6
Produtor - Produziu -> 6.
```

## Especificação do código

O código foi dividido em 2 partes,

- A manutenção do buffer: Gerência de operações de inicialização, adição e remoção dos itens do buffer.
- O produtor-consumidor: Duas funções que, por meio de semáforos, controlam o acesso a recursos da região crítica acessada pelas duas threads em execução simultânea.

## Requisitos

- O programa deve ter um buffer limitado, acessível a qualquer processo decorrente do processo principal;
  - O Buffer `buf` foi criado e é acessado por meio de qualquer um dos dois processos por meio do parâmetro passado a eles.
- O programa deve ter uma fila apontando para o próximo endereço livre, a ser escrito;
  - A fila é representada pela variável `in`, que representa o próximo offset do endereço livre a ser escrito.
- O programa deve ter uma fila apontando para o próximo endereço ocupado, a ser lido e liberado;
  - A fila é representada pela variável `out`, que representa o próximo offset do endereço a ser lido e liberado.
- O programa deve controlar as seções críticas (Compartilhamento de memória), para que não haja acessos indevidos.
  - Ao utilizar semáforos o programa controla essas seções críticas para que não ocorra o acesso indevido.
- O programa deve ter a capacidade de colocar um processo em modo de espera;
  - Ao utilizar `sem_wait` o processo pode ser colocado em espera caso seja necessário.
- O programa deve ter a capacidade de controlar quando um processo está em espera, para poder "chamar" o mesmo;
  - Por meio dos semáforos este controle é feito.
- O programa deve controlar quantos endereços estão livres e quantos endereços estão ocupados.
  - Ao ter as variáveis `in` e `out` tem-se exatamente quantos endereços estão livres ou não.

## Relatório Módulo 3

Para este módulo foi desenvolvido um simulador de gerenciamento de memória virtual com paginação. Para isso foram tomadas como base instruções de escrita/leitura em uma posição de memória, criação de um processo de tamanho especificado e operações de CPU e I/O, que também foram mapeadas na memória.

### Instruções Para a Execução dos Códigos

Foi criado um arquivo `Makefile`, que se encontra na pasta [simulador](./simulador/). Para compilar o programa basta utilizar o comando `Make` nesta mesma pasta, o que gerará um arquivo binário também de nome `simulador`.

Para executar o programa, é possível utilizar um arquivo de entrada como argumento:



```
./simulador arquivo_teste.txt
```

Durante a execução são impressas na tela do usuário as manipulações feitas referentes tanto à memória virtual quanto à memória real simuladas.

## Arquitetura do Código

### A tradução de endereços

Como no arquivo de entrada são dados endereços, é necessário que estes sejam traduzidos para páginas de memória. Então isso é calculado com base no tamanho de página e então inserido na respectiva página da memória virtual.

Com relação à memória real, as páginas são criadas e inseridas sequencialmente no caso de que uma página de memória virtual não tivesse sido mapeada anteriormente. Caso contrário o acesso é direto.

### CPU e I/O Bound

Para operações do tipo CPU e I/O bound foram utilizado o bit mais significativo e o segundo bit mais significativo para indicar o mapeamento de páginas para esses tipos de operações na memória virtual. Por conta dessa decisão de design, cada uma dessas operações possui uma região específica para si na memória virtual, que não será utilizada por outras operações de memória.

### Estruturas de Dados para a Memória

Para que o projeto se assimilasse à realidade, foram utilizadas algumas estruturas de dados da biblioteca STL da linguagem C++.

As memórias virtual e real foram representadas com a estrutura ``vector`` do C++, devido ao seu comportamento sequencial. Essa estrutura foi escolhida também pois facilita manipulações do tipo inserção e remoção tanto no início/fim da memória como em posições intermediárias com complexidade amortizada em  $O(1)$ .

No caso da memória virtual, foi necessário utilizar um vetor de ``pair``, dado que esta é simulada a ser única para cada processo, então formando um par ``<pid, página de memória>``.

Foi utilizada uma tabela de memória virtual para a memória real, traduzindo as informações da memória virtual para o seu respectivo frame na memória real. Isso foi feito utilizando a estrutura de dados ``map``, que realiza acessos em  $O(1)$  amortizado e modificações em  $O(\log(n))$ .

Também, devido às limitações impostas de um número máximo de páginas de memória mapeadas simultaneamente de um processo e ao tamanho máximo de memória de um processo, foram criadas duas tabelas (utilizando `map`) para manter e controlar essas condições.

### Configuração

Conforme solicitado pela especificação, os parâmetros do simulador podem ser customizados na classe de configuração (Config.cpp), ou pode ser também criada uma nova classe de configuração com base no construtor especificado nesse mesmo arquivo.

## **Relatório Módulo 4**

Este modulo foi desenvolvido para comparar alguns dos sistemas de arquivos existentes, de forma que a comparação contém criação, leitura e exclusão de arquivos em cada tipo de sistema de arquivo. Uma breve explicação dos tipos de sistemas de arquivos é apresentado assim como a execução dos códigos implementados e os resultados obtidos.

### **Sistemas de arquivos**

É possível armazenar e recuperar uma grande quantidade de informação, desta forma permite ao sistema operacional poder ler e gravar dados nos arquivos ou poder criar arquivos. Possui como tarefas estruturar os arquivos, os nomes, os tipos de acessos, sua proteção e a sua implementação. Existem diversos tipos de sistemas de arquivos como o ext, ntfs e o fat32.

#### **Fat32**

O sistema de arquivo fat (Tabela de alocação de arquivos) foi desenvolvido para o MS-DOS e consiste em uma tabela na memória que armazena o ponteiro do bloco. Porém existia alguns limites para os nomes dos arquivos que eram curtos e não existia o conceito de diferentes usuários, de forma que todos os arquivos podiam ser acessados por todos os usuários. Assim, com o intuito de superar o limite do tamanho de volume dos arquivos, surge o Fat32. Consequentemente o tamanho dos blocos aumentaram possuindo assim 4KB, 8KB, 16KB e 32KB e também o tamanho da partição aumentou para 2TB.

#### **NTFS**

O ntfs (new technology filesSystem) é o sistema de arquivos padrão para o Windows NT e seus derivados. Possui algumas características do sistema de arquivo HPFS (high performance file system - sistema de arquivos do OS/2). O NTFS possui uma estrutura que armazena as localizações de todos os arquivos e diretórios, incluindo os arquivos referentes ao próprio sistema de arquivos denominado MFT (Master File Table). Possui como característica sua confiabilidade, que gera a capacidade de se recuperar de problemas sem perda de dados e possui melhor tolerância a falhas. Aceita volumes de até 256TB utilizando tamanho de cluster de 64KB;

#### **Ext4**

O ext4 é um sistema de arquivos de registro para Linux, desenvolvido para ser o sucessor do ext3. Assim, o ext4 perimiut a criação dos extents que é um conjunto de blocos físicos contíguos, o que, para arquivos grandes, faz aumentar o desempenho e reduzir a fragmentação. O ext4 assim como o ext3 possuem um registro de ações que é chamado de journing. O o journaling grava todas as mudas e usa um arquivo de registro de ações maior, só que este tipo pode ser mais lento mas possui maior capacidade de evitar perdas.

### **Instrução para execução**

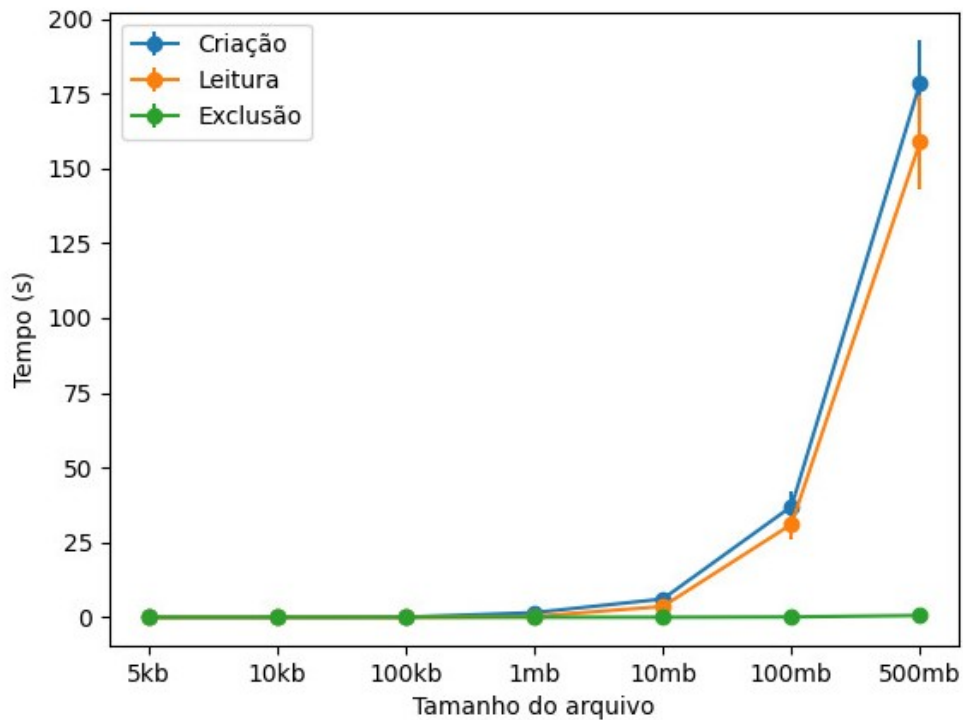
Para compilar o código é necessário executar o comando make para que gere um arquivo de execução. E assim, o script `test.py`, em python3, presente na pasta `Códigos` foi desenvolvido que executa todas as opções e mostra os resultados.

Para executá-lo basta utilizar o comando ``python test.py``.

Caso algum problema de biblioteca ocorra pode ser necessário instalar a biblioteca matplotlib, do python, o que varia com base na distribuição utilizada.

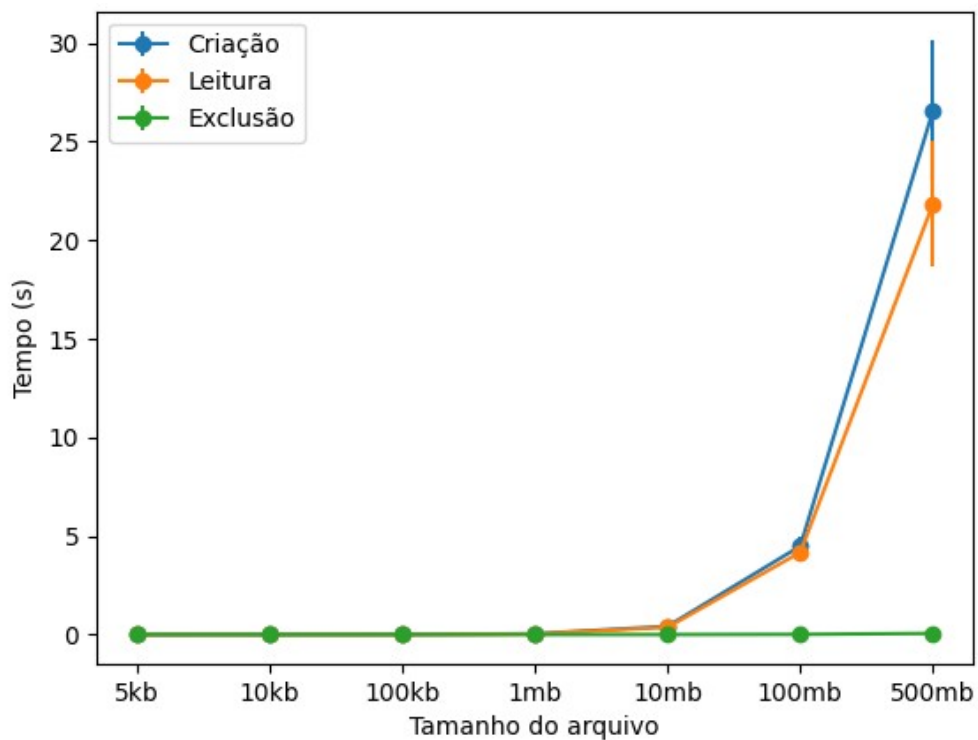
## Resultados

### FAT



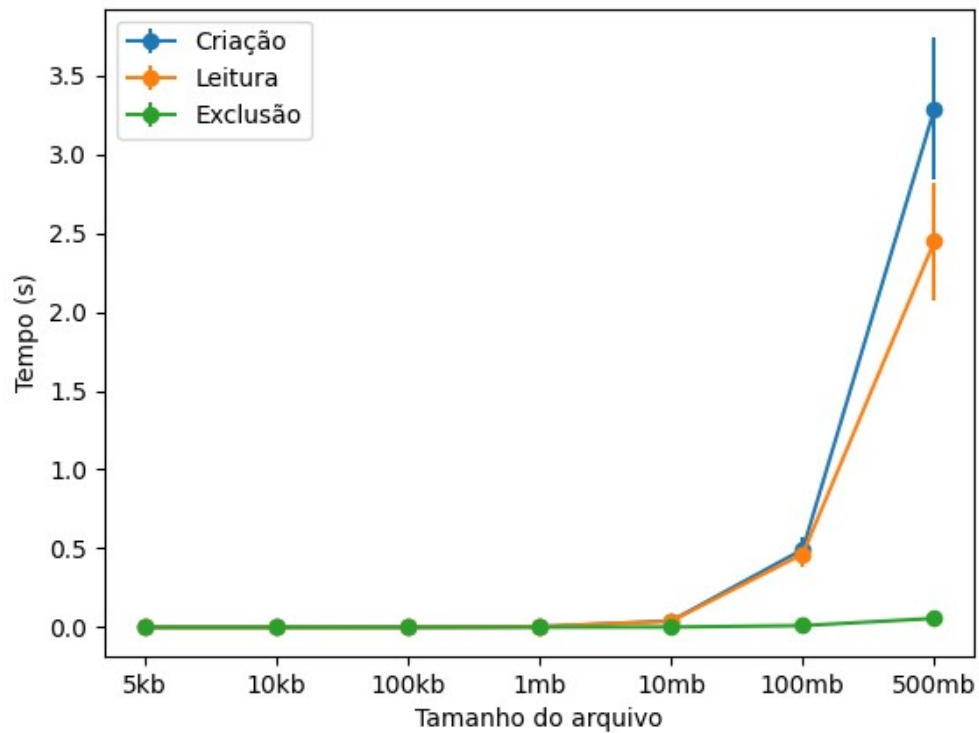
### NTFS

A execução no sistema de arquivos ntfs.



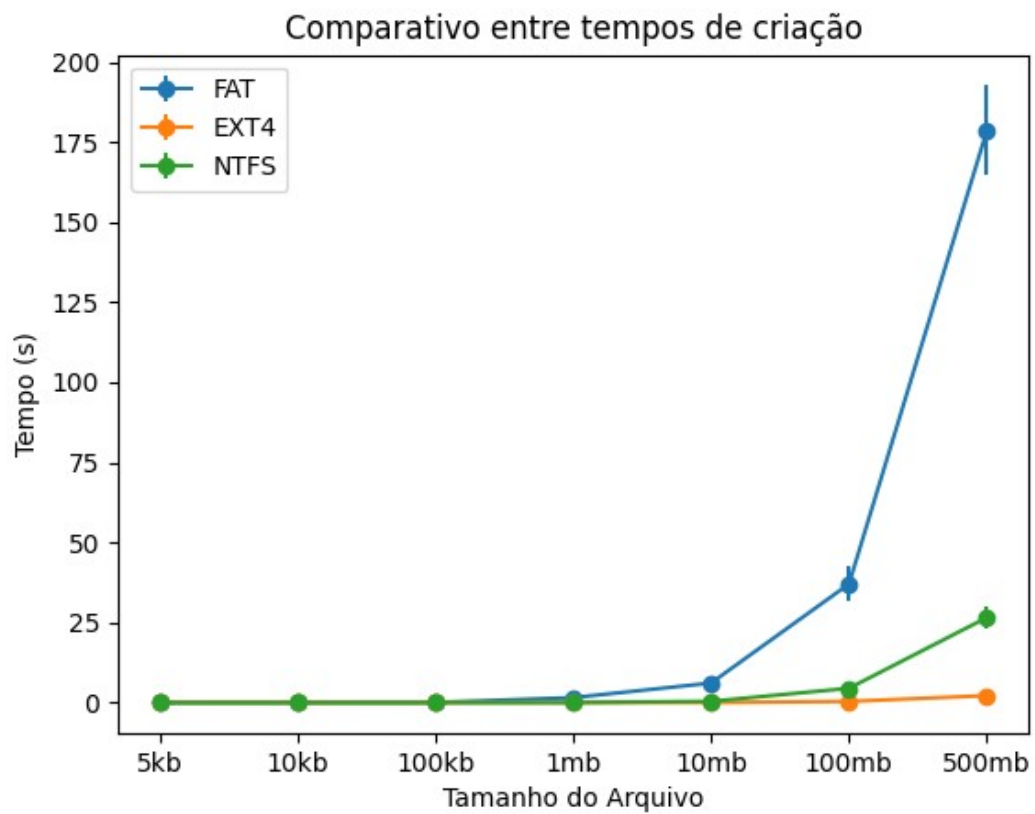
## EXT4

A execução no sistema de arquivos ext4.

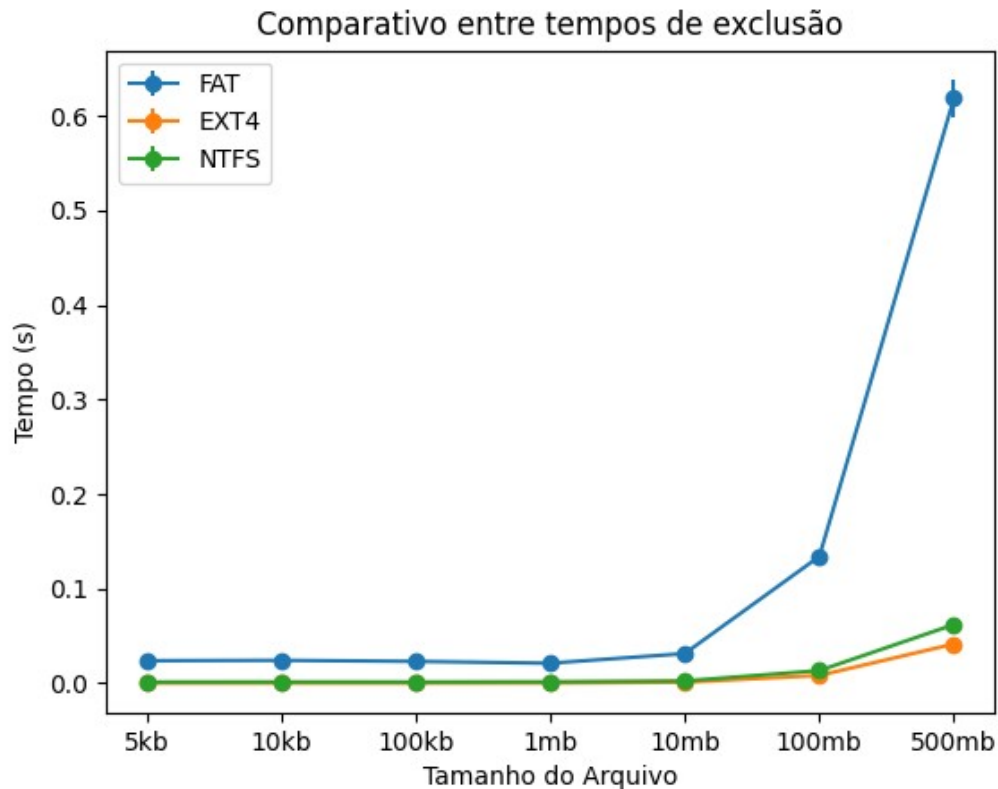


## Comparativos

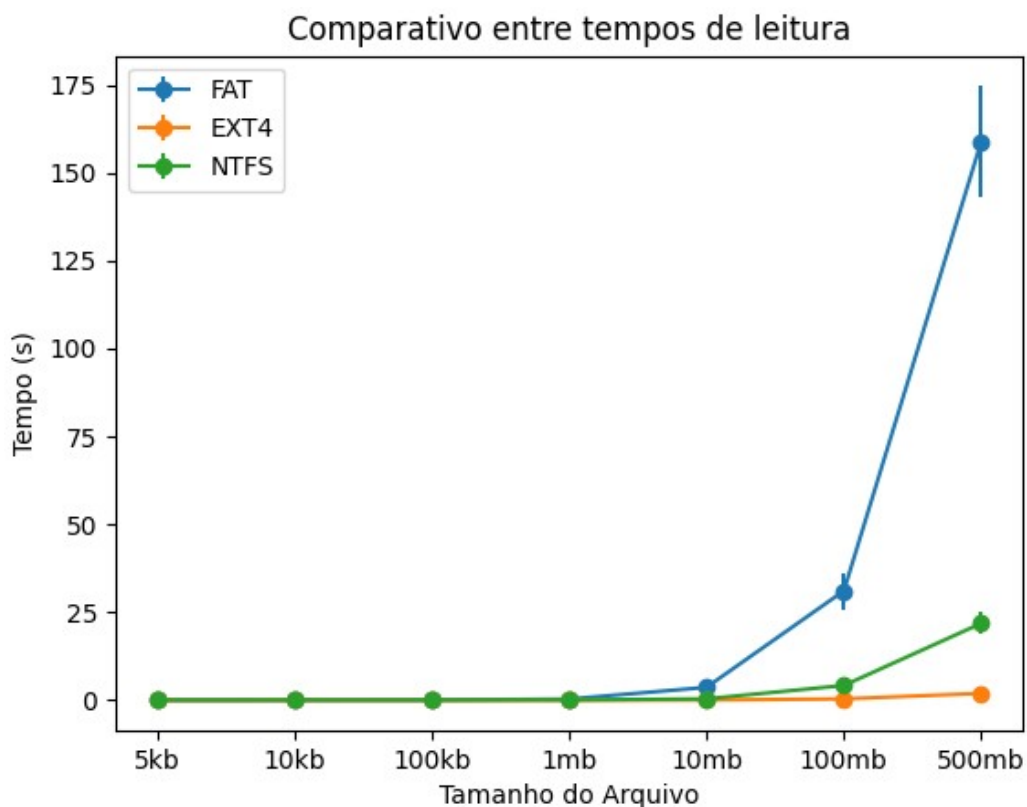
### Criação de Arquivos



## Exclusão de Arquivos



## Leitura de Arquivos



## Resultados das execuções

Os resultados obtidos demonstraram que a operação de criação de um arquivo é a que demanda mais tempo para sua execução e a operação de exclusão é a que demora menos tempo. Em relação ao tempo para cada tamanho de arquivo é possível observar que conforme o tamanho do arquivo

aumenta, maior é o tempo de uma operação. Em relação em comparação dos três tipos escolhidos de sistemas de arquivos foi constatado que o sistema de arquivo fat32 foi que obteve os maiores tempos de execução das operações demonstrando que não é tão eficiente em comparação aos outros tipos de sistemas de arquivos para arquivos grandes. O sistema de arquivo ntfs obteve melhores tempos em relação ao fat32, porém comparado com o ext4 não. Dado que os resultados do ext4 os tempos ficaram abaixo de 4 segundos para qualquer tipo de operação. Sendo assim, o ext4 se sobressaiu sobre todos os sistemas de arquivos verificados.

Os resultados numéricos para as 3 execuções se encontram no arquivo ``comparativo.py``, com um exemplo de geração de imagem de comparação.

## **Especificação do código**

O código foi dividido em três arquivos sendo cada arquivo para um tipo de operação existente, como um para criação, leitura e exclusão de arquivos. E para cada tipo de operação, foi realizado uma verificação com o tamanho variado dos arquivos, sendo os tamanhos 5Kb, 10Kb, 100Kb, 1Mb, 10Mb, 100Mb e 500Mb, assim medindo o tempo de execução para cada tamanho.

Para a medição do tempo foi utilizada a função ``gettimeofday``, que dá maior precisão do que utilizar a biblioteca ``time`` por si.

Além disso, como garantia, foram medidos quantos ciclos de cpu cada uma das execuções leva, porém não foi necessário utilizar tais medidas para a criação dos gráficos dado que elas geravam apenas redundância.