RELATÓRIO EXPERIMENTO 4 - MC504 (SISTEMAS OPERACIONAIS)

André Santos Rocha - 235887
Pedro da Rosa Pinheiro - 231081

1. Introdução

Buscamos com este relatório discorrer sobre o desenvolvimento do experimento 4 realizado para a disciplina de Sistemas Operacionais. O projeto consiste em otimizar um sistema xv6 para uma carga pré-determinada de dois processos independentes diferentes: um *IO-bound* e o outro *CPU-bound*.

Primeiramente realizamos a implementação de rotinas separadas que executam ambos processos, para, posteriormente, juntarmos ambas em um só programa, aqui chamado de *run_experiment*, responsável por executar 30 rodadas de uma quantidade pseudoaleatória dos dois processos citados anteriormente.

Em seguida, implementamos um conjunto de métricas responsáveis por avaliar diferentes áreas do experimento, como justiça entre processos e eficiência do sistema de arquivos, de tal modo a serem devolvidas ao usuário ao final de cada rodada. Desse modo, temos uma análise normalizada do funcionamento e eficiência do nosso programa sob diferentes atributos e características.

Como última etapa do experimento, iniciamos a otimização do escalonador do sistema operacional utilizado. O xv6 utilizado foi implementado por um conjunto de professores do MIT objetivando auxiliar no processo de ensino e aprendizagem das matérias relacionadas à Sistemas Operacionais. Portanto, embora possua um algoritmo de escalonamento pronto, foi necessário explorarmos opções e diferentes abordagens, de modo a encontrarmos a melhor estratégia para a nossa rotina.

2. Processos e Rotinas

A nossa rotina principal *run_experiment* é responsável por tanto rodar os diferentes processos implementados quanto por calcular as métricas associadas a cada rodada.

Objetivando gerar uma carga de processamento a CPU, fizemos uma tarefa *CPU-bound*, incumbida de *(1)* gerar 1000 grafos direcionados aleatórios, cada um com um número de vértices entre 100 e 200 e número de arestas entre 50 e 400, e,

juntamente, *(2)* calcular o tamanho do caminho mínimo entre um vértice de origem e todos os outros do grafo. Assim, implementamos duas funções: uma que gera um grafo direcionado aleatório conforme os limites pré-estabelecidos e outra que implementa o algoritmo de Dijkstra em sua versão baseada em um vetor de vértices de valor mínimo. Válido ressaltar aqui que tanto nesta etapa dos grafos aleatórios quanto em qualquer outro momento do projeto, utilizamos nossa própria função de geração números pseudoaleatórios.

Como segunda parte da geração de carga para o experimento, fizemos também uma tarefa *IO-bound* que *(1)* gera strings de 100 caracteres aleatórios, *(2)* escreve 100 dessas strings aleatórias em um arquivo, *(3)* realiza a permutação de 50 dessas strings aleatórias dentro do arquivo e, por fim, *(4)* deleta o arquivo utilizado. Todas as funções associadas à essas etapas fazem uso das *syscalls write()* e *read()* pré-implementadas no xv6. Dessa forma, a implementação dessa etapa foi direta: temos uma função que gera uma string aleatória de 100 caracteres, a qual chamamos 100 vezes intercalando com as escritas para depois, ao fim dessa etapa, gerarmos 50 vezes dois índices aleatórios representando duas linhas a serem trocadas.

Por fim, temos a rotina responsável por organizar ambas as descritas acima: *run_experiment*. Nela, executamos 30 rodadas, onde cada uma é composta por um número X de execuções da tarefa *CPU-bound* e um número Y de execuções da tarefa *IO-bound* tal que $X \in [6, 14]$ e $Y = 20 - X$. Além disso, para cada uma das rodadas, calculamos cada uma das métricas necessárias discorridas abaixo e devolvemo-nas para a saída padrão.

3. Métricas

Como falado anteriormente, as métricas implementadas são responsáveis por avaliarem o desempenho do nosso sistema na execução das tarefas e rotinas implementadas. Cada uma considera um atributo diferente do sistema e, ao final, são combinadas em uma métrica final associada ao desempenho geral do sistema.

A **vazão** *(throughput)* calcula o número de processos do tipo *CPU-bound* e *IO-bound* executados pelo sistema ao longo de um segundo. Para isso, utilizamos a função pré-implementada no *xv6 uptime* que retorna o número de ticks desde que o sistema foi iniciado. Mantemos sempre uma variável da forma *old* que guarda o

valor retornado na última chamada da função e, a cada 100 ticks, analisamos o número de rotinas que foram executadas, atualizando, a cada checagem, o valor máximo de mínimo de vazão encontrados, ou seja, a quantidade de processos executados nesse intervalo. Válido ressaltar que utilizamos o valor de 100 ticks, já que, como descrito na documentação do próprio *xv6*, um tick é realizado a cada 10ms. Assim, ao final de uma rodada, temos o valor máximo e mínimo de vazão encontrados juntamente ao número de ticks e de processos executados totais, permitindo-nos calcular a vazão normalizada através da fórmula

$$T_{put}^{norm} = 1 - \frac{E[T_{put}] - T_{put_{min}}}{T_{put_{max}} - T_{put_{min}}}$$

A **justiça** entre processos representa a equidade do algoritmo de escalonamento de processos utilizado pelo escalonador. Toda vez que o programa finaliza um processo, seja esse *CPU-bound* ou *IO-bound*, contabilizamos o tempo passado e, em outra variável, o quadrado de tal valor. Ao final, é possível calcular a justiça através da fórmula

$$J_{cpu} = \frac{(\sum_{i=1}^{N} x_i)^2}{N * \sum_{i=1}^{N} x_i^2}$$

tomando *N* como o número de processos realizados na rodada e *xi* como o tempo que o processo *i* demorou para ser finalizado.

A **eficiência** do sistema de arquivos é definida como o tempo médio de leitura, escrita e remoção de arquivos. Para cada uma das chamadas de sistema realizadas em *write(), read()* ou *unlink()*, contabilizamos o tempo levado para tal. Assim, ao longo das funções de escrita de strings aleatórias e de permutação de linhas, agregamos valores de número de *syscalls* pelo total de ticks que levaram para as funções *write(), read()* e *unlink()*. Com isso, é possível calcular a eficiência com

$$E_{fs} = \frac{1}{T_{write} + T_{read} + T_{del}}$$

O ***overhead*** do gerenciamento de memória nos dá um valor associado ao tempo necessário para realizar operações na memória principal, como alocação de memória e liberação da mesma. Semelhantemente ao cálculo da eficiência, toda vez que realizamos uma chamada de sistema associada ao “manuseio” de memória, como *malloc()* e *free()*, precisamos guardar o tempo que foi levado para a conclusão. Medimos, portanto, ao final de cada rodada, os valores da quantidade dessas *syscalls* sobre o número de ticks que levaram para ser concluídas, nos permitindo obter o *overhead* com

$$M_{over} = \frac{1}{M_{access} + M_{alloc} + M_{free}}$$

Por fim, baseando-nos nas quatro métricas descritas acima, podemos calcular, para cada rodada, um valor para avaliar o **desempenho geral do sistema**. Assim como falado, essa métrica nos permite ter uma ideia do funcionamento e da eficiência do sistema de uma forma balanceada entre diferentes atributos avaliados. Para tal, temos a fórmula

$$S_{perform} = 0.25 * T_{put}^{norm} + 0.25 * J_{cpu} + \\ + 0.25 * E_{fs} + 0.25 * M_{over}$$

4. Escalonador

Para a otimização do escalonador, optamos por substituir o Round Robin utilizado pelo xv6 pelo Multi Level Feedback Queue.

A estratégia de escalonamento Round Robin no sistema operacional xv6 é um método simples e eficiente de gerenciamento de processos que garante a equidade na utilização do processador. Nesse esquema, cada processo na fila de prontos recebe um intervalo de tempo, chamado de *time slice* ou *quantum*, durante o qual ele pode ser executado. Quando esse tempo se esgota, o processo é colocado no final da fila e o próximo processo é selecionado para execução. Isso continua ciclicamente, garantindo que todos os processos tenham acesso ao processador de forma periódica.

Por outro lado, a estratégia de escalonamento *Multi-Level Feedback Queue* (MLFQ) é um método sofisticado de gerenciamento de processos que permite a adaptação dinâmica das prioridades de execução com base no comportamento dos processos. Nesse esquema, existem várias filas, cada uma com uma prioridade diferente, sendo que processos que necessitam de menos tempo de CPU (curtos ou interativos) são promovidos para filas de maior prioridade, enquanto processos que requerem mais tempo de execução são rebaixados para filas de menor prioridade. A principal vantagem dessa abordagem é a sua flexibilidade em responder de forma eficiente tanto a processos interativos quanto a processos de longa duração, garantindo que processos com diferentes necessidades de CPU sejam tratados de forma justa e adequada. Essa adaptabilidade permite um balanceamento eficaz entre responsividade e utilização eficiente do processador, sendo especialmente útil em sistemas multitarefa.

5. Conclusão

Após a implementação de todas as rotinas, métricas e otimização acima, realizamos um experimento completo do sistema pré e pós otimização. Assim, chegamos nas seguintes comparações

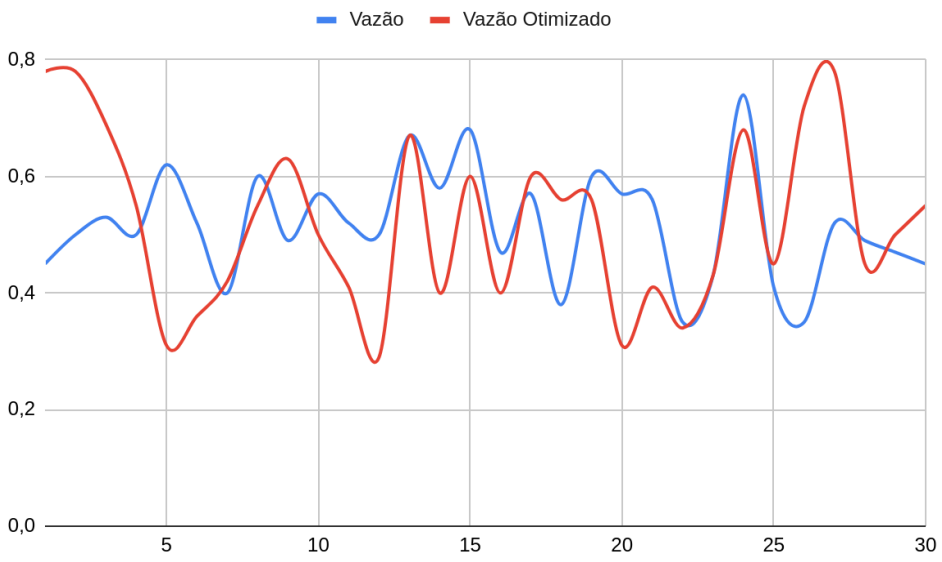

Figura 1: Gráfico da Vazão e Vazão Otimizado X Rodadas

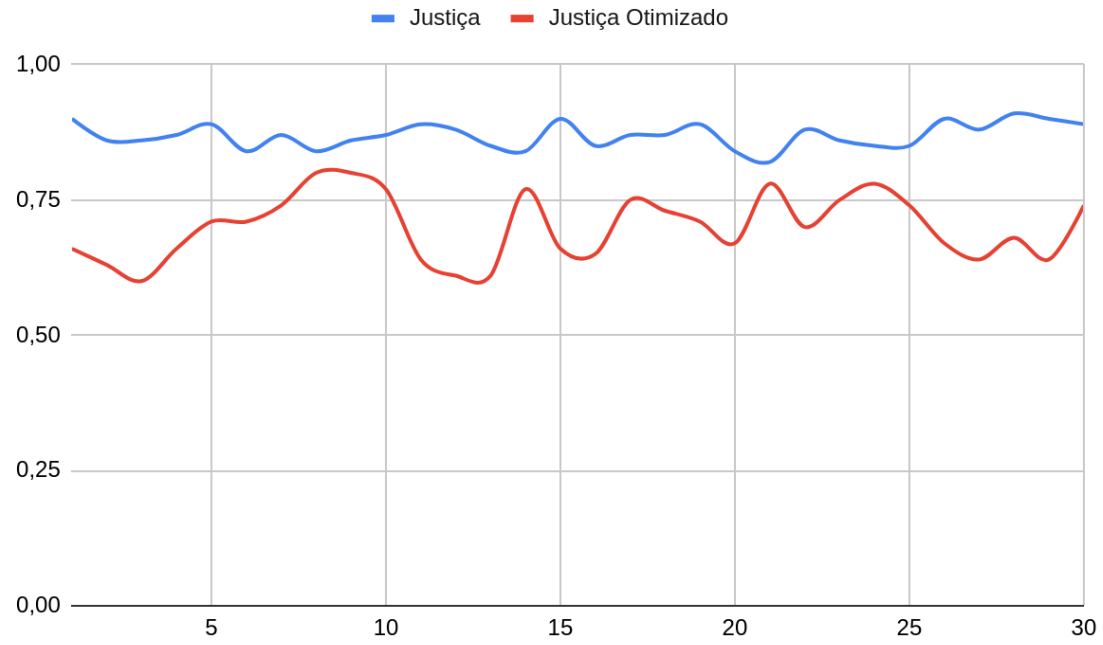

Figura 2: Gráfico da Justiça e Justiça Otimizado X Rodadas

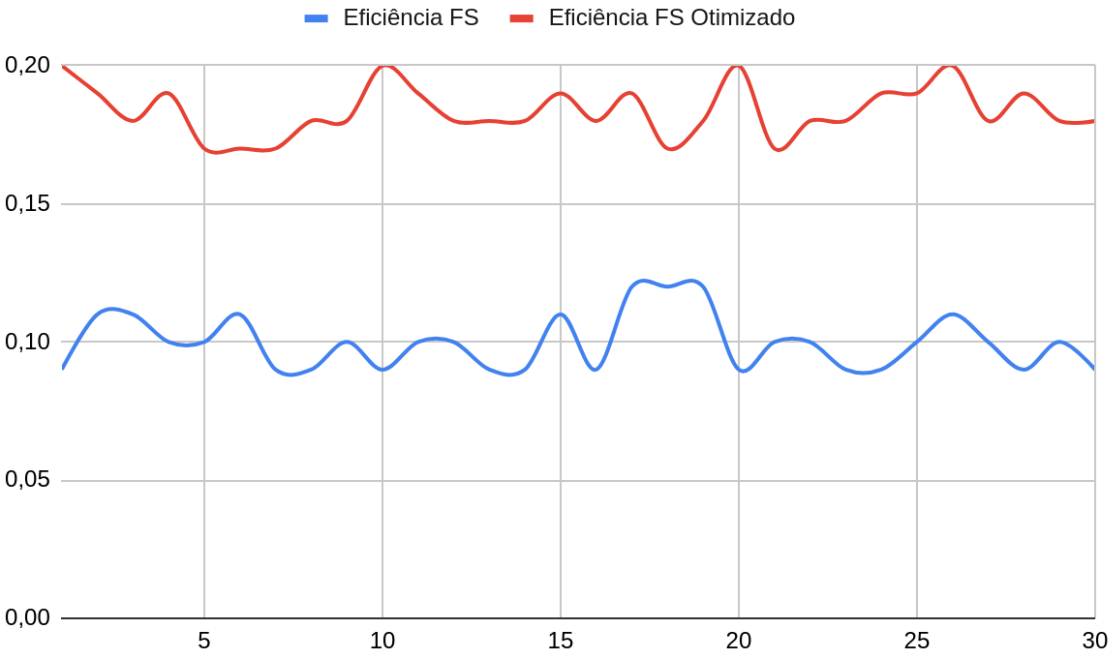

Figura 3: Gráfico da Eficiência e Eficiência Otimizado X Rodadas

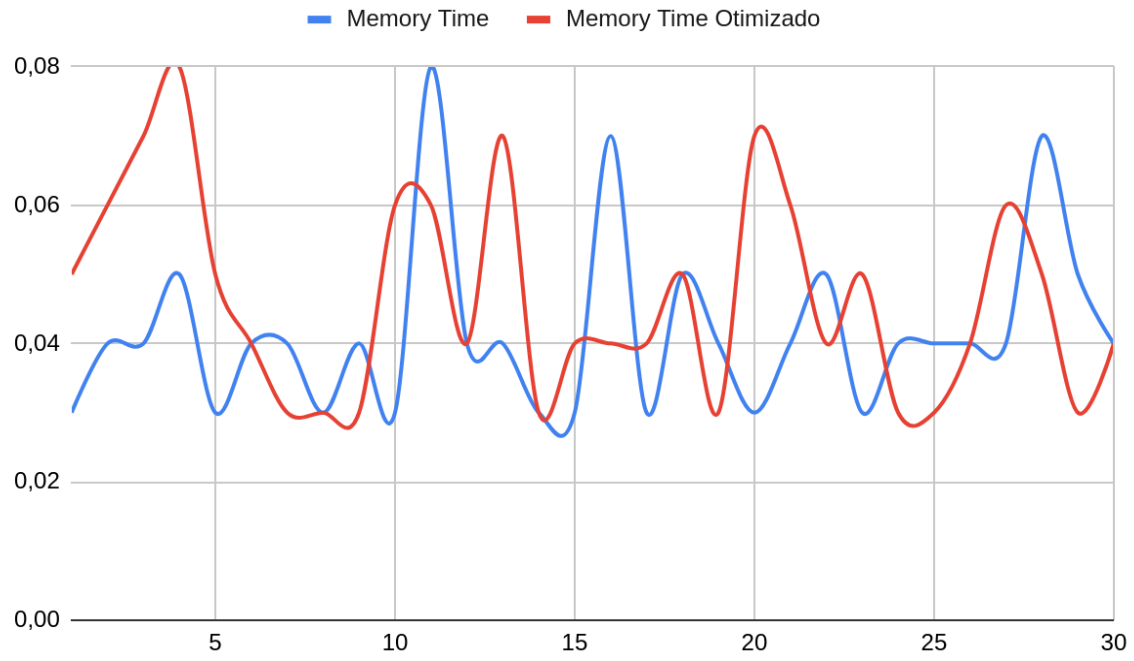

Figura 4: Gráfico do Overhead de Memória e Overhead de Memória Otimizado X Rodadas

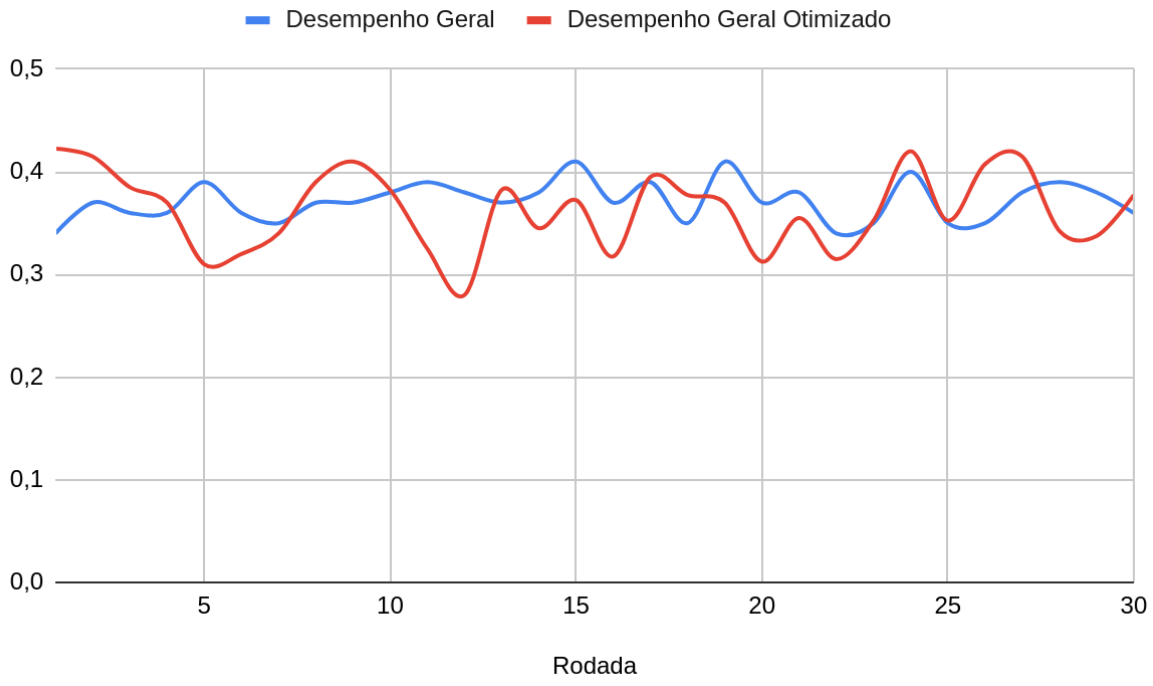

Figura 5: Desempenho Geral e Desempenho Geral Otimizado X Rodadas

Vemos logo de cara que, com as otimizações, obtivemos um aumento significativo na eficiência do sistema de arquivos e no overhead do gerenciamento de memória, assim como pode ser visto na Figura 3 e 4, respectivamente. No entanto, vê-se que, simultaneamente, tivemos uma queda considerável na justiça entre os processos. Conclui-se, portanto, que apenas com uma otimização no algoritmo de escalonamento utilizado, obtivemos uma melhora pequena, mas ainda sim importante, no desempenho geral do sistema.