UNIVERSIDADE FEDERAL RURAL DO SEMI-ÁRIDO
PRÓ-REITORIA DE GRADUAÇÃO
CENTRO DE CIÊNCIAS EXATAS E NATURAIS
DEPARTAMENTO DE COMPUTAÇÃO
CURSO DE CIÊNCIA DA COMPUTAÇÃO
TRABALHO DE CONCLUSÃO DE CURSO (2024)

DESENVOLVIMENTO DE UM ARQUIVADOR *MULTI-THREADED* PARA SISTEMAS *UNIX-LIKE*

Ríad Oliveira de Morais[1], Paulo Henrique Lopes Silva[2]

RESUMO

Este trabalho apresenta o desenvolvimento de um arquivador *multi-threaded* para sistemas *Unix-like*, com o objetivo de otimizar o desempenho de operações de arquivamento e desarquivamento por meio da paralelização de tarefas. O *software* foi implementado em *C*, utilizando a biblioteca POSIX *threads* para o gerenciamento de *threads*. A estratégia de paralelização adotou o padrão de projeto *Producer-Consumer*, permitindo o compartilhamento eficiente de dados entre *threads*. Foram efetuados testes comparativos de desempenho com arquivadores comuns dos sistemas *Unix-like*, como o *tar* e o *cpio*. Os resultados obtidos apresentaram ganhos de performance significativos do arquivador *multi-threaded* em relação aos demais. Como trabalhos futuros, sugere-se a adição de novas funcionalidades e a integração com compressores paralelos, visando aprimorar o desempenho e ampliar a abrangência do arquivador.

Palavras-chave: arquivador; *unix-like*; computação paralela; *multithreading*; otimização.

1 INTRODUÇÃO

O gerenciamento eficiente de arquivos é crucial em diversos sistemas computacionais. Nesse âmbito, arquivadores são *softwares* capazes de combinar múltiplos arquivos e diretórios em um único arquivo de saída, permitindo a recuperação dos conteúdos posteriormente. Devido a esse funcionamento, é comum utilizá-los em conjunto com compressores, otimizando o tamanho do arquivo resultante. Dessa forma, torna-se nítida a relação intrínseca entre ferramentas de arquivamento e compressão, sendo frequentemente integrados em um único programa para simplificar a utilização pelos usuários.

Tendo em vista a finalidade desses utilitários, percebe-se a essencialidade deles na execução eficiente de diversas tarefas computacionais, principalmente aquelas relacionadas à transferência e ao armazenamento de grandes volumes de dados. Diante disso, é válido presumir que esses *softwares* aplicam múltiplas técnicas de otimização, como a paralelização, para maximizar o desempenho e atender às demandas dos usuários de forma satisfatória.

Contudo, em sistemas *Unix-like*, alguns dos arquivadores mais disseminados são o *tar* e o *cpio*, os quais, de acordo com suas documentações e códigos-fonte, ainda realizam o arquivamento de maneira sequencial (GNU, 2024). Essa abordagem, embora eficaz nesse contexto, não aproveita as capacidades dos processadores modernos. Por outro lado, existem múltiplos compressores que paralelizam o processamento de suas computações, entre eles, podem ser destacados o *pigz* (ADLER, 2024) e o *zstd* (FACEBOOK, 2024).

---

[1] Graduando do Bacharelado em Ciência da Computação da UFERSA
[2] Professor Associado do Departamento de Computação da UFERSA/Mossoró

Diante desse contraste, surge o questionamento sobre por que os arquivadores ainda não empregam algoritmos paralelos. Para entender isso, é necessário revisitar o conceito de paralelismo, o qual, segundo Bellairs (2019), pode ser definido como o processo de usar um conjunto de recursos para resolver determinado problema em menos tempo, por meio da divisão do trabalho computacional. Contudo, nem todas as operações se beneficiam igualmente da paralelização, Butenhof (1997) explica que as tarefas paralelas ideais são aquelas classificadas como *CPU-bound*, ou seja, dependem fortemente dos recursos do processador para serem executadas, requerendo sincronização e bloqueio mínimos. Com isso, dividir esse tipo de atividade em múltiplos fluxos de processamento frequentemente é uma escolha vantajosa, resultando em maior performance do programa.

Em contrapartida, o funcionamento dos arquivadores consiste essencialmente em ler os conteúdos fornecidos como entrada e escrever esses dados em um arquivo de saída. Portanto, a execução desses *softwares* caracteriza-se como *I/O-bound*; em outras palavras, sua velocidade e eficiência estão significativamente sujeitas à taxa de transferência de dados que os dispositivos de entrada e saída do sistema são capazes de atingir.

Devido à limitação atrelada ao problema, o desenvolvimento de uma solução efetiva pode ser um desafio. Todavia, outros autores exploraram maneiras de contornar esse tipo de dificuldade previamente, como May (2000) que estudou a possibilidade de criar múltiplos caminhos entre a memória e os dispositivos de armazenamento para otimizar operações de entrada e saída, percebendo a necessidade de considerar os pontos de gargalo do *software* nas análises. Além disso, Matney e Shipman (2011) desenvolveram alternativas paralelas de utilitários *I/O-bound* do *Unix*, como o *tar* e o *cp*, efetuando *benchmarks* e demonstrando a obtenção de performances substancialmente maiores em relação às versões originais. Com base nesses trabalhos, torna-se evidente que existem estratégias capazes de paralelizar eficientemente as atividades exercidas pelos arquivadores, mostrando-se essencial localizar as seções críticas de desempenho no algoritmo, para aplicá-las de maneira eficaz.

Dessa forma, é crucial determinar qual método é o mais adequado para paralelizar as operações de arquivamento, sendo *multithreading* e *multiprocessing* as principais abordagens. Para explorar cada um desses conceitos, é importante ressaltar a definição de *threads*, as quais são fluxos de execução independentes em processos.

Dito isso, conforme Wong (2023), conceitualmente *multithreading* refere-se à capacidade do computador processar múltiplas *threads* de maneira concorrente, enquanto *multiprocessing* seria a execução de vários processos paralelamente, em processadores distintos. Ademais, ela afirma que utilizar *multithreading* para operações *CPU-bound* pode reduzir a performance da aplicação, tendo em vista a competitividade pelos recursos compartilhados. Por outro lado, aplicar *multiprocessing* em tarefas *I/O-bound* pode gerar custos elevados em relação ao gerenciamento e à comunicação entre processos. Tendo isso em mente, evidencia-se a estratégia de *multithreading* como a escolha ideal para o trabalho.

A técnica de *multithreading* é difundida por ser eficaz e vantajosa, como é ratificado no estudo de Shanthi e Irudhayaraj (2009), o qual explora as potencialidades e os benefícios dessa estratégia no incremento do desempenho de aplicações inseridas em contextos diversos, evidenciando os ganhos de performance em relação a algoritmos *single-threaded*. Além disso, Butenhof (1997) enfatiza a relevância das *threads* para garantir a utilização dos recursos computacionais de forma eficiente, possibilitando a criação de aplicações otimizadas.

No entanto, conceitualmente as *threads* funcionam com base em concorrência, Bellairs (2019) explica que *multithreading*, em *single-core*, proporciona a ilusão de paralelismo, mas o processador está trocando rapidamente entre os contextos das *threads*. Outra desvantagem foi apontada por Shanthi e Irudhayaraj (2009), sendo essa a necessidade de gerir e coordenar cada uma das *threads* criadas, processo o qual pode escalar exponencialmente em complexidade. Dessa forma, para aproveitar os benefícios dessa abordagem e aplicá-la adequadamente, é

necessário lidar com seus custos e, almejando o funcionamento verdadeiramente paralelo, utilizá-la especialmente em ambientes *multi-core*.

A essencialidade dos arquivadores provou-se indubitável, sendo imprescindíveis tanto para o funcionamento de áreas críticas da computação, quanto para além da esfera de desenvolvimento, como em atividades cotidianas de usuários comuns, as quais frequentemente envolvem transferência e armazenamento de dados. Tendo em vista a relevância dessas ferramentas, torna-se evidente que, independentemente do âmbito, os usuários almejam executar suas tarefas da maneira mais otimizada possível. Diante dessa realidade, justifica-se a busca por soluções capazes de aprimorar o desempenho do arquivamento e, consequentemente, a experiência de utilização dos arquivadores.

Sendo assim, o trabalho busca estudar a viabilidade e os desafios de desenvolver um arquivador que utilize *multithreading* para paralelizar suas tarefas em processadores *multi-core*. Com essa abordagem, o objetivo é alcançar maior desempenho na execução das operações de arquivamento, superando os utilitários padrão dos sistemas *Unix-like*.

Por fim, para atingir o propósito deste trabalho, é necessário utilizar uma linguagem de programação de baixo nível, visando reduzir os custos de abstração e intensificar a performance, por meio do gerenciamento preciso dos recursos computacionais. Desse modo, optou-se pela linguagem *C*, a qual, além de apresentar as características desejadas, possui a biblioteca POSIX *threads* (*pthreads*) que permite a criação e manipulação de *threads*, através do fornecimento de funções básicas, as quais seguem os padrões POSIX *(Portable Operating System Interface)* em suas interfaces, ou seja, é um *software* facilmente portável em sistemas *Unix-like*, sendo comumente instalado por padrão (THE OPEN GROUP, 2004).

2 DESENVOLVIMENTO

Esta seção aborda todas as informações pertinentes e etapas necessárias para efetuar o planejamento e desenvolvimento do projeto abordado.

2.1 Método

Para evidenciar os detalhes do desenvolvimento, separou-se as tarefas necessárias para o cumprimento dos objetivos definidos em subseções, que serão aprofundadas posteriormente, mas estão listadas e brevemente comentadas a seguir:

1. **Definição das tecnologias**: descrição das ferramentas selecionadas para o desenvolvimento do projeto, assim como a justificativa para a escolha delas;
2. **Abordagem de implementação**: discussão acerca das possíveis abordagens para o desenvolvimento do *software*, com análise dos benefícios de cada uma, dos públicos-alvo que visam atender, e a justificativa para a escolha final;
3. **Arquitetura geral do projeto**: apresentação da estrutura completa do código, destacando a organização e o propósito de cada pasta e subpasta;
4. **Estratégia de paralelização**: definição do padrão de paralelização adotado, explorando os conceitos relacionados e a aplicação desse modelo no algoritmo do arquivador;
5. **Processo de arquivamento**: detalhamento do algoritmo empregado no arquivamento de conteúdos, com foco nas responsabilidades de cada *thread* e no tratamento fornecido para os tipos de conteúdos;
6. **Processo de desarquivamento**: descrição do processo de restauração dos conteúdos arquivados, enfatizando as operações relativas a cada *thread* e as peculiaridades do algoritmo para recriar fielmente as informações originais.

Dessa forma, as etapas descritas foram organizadas de forma prática e coerente, assegurando que cada uma contribua efetivamente para os propósitos do trabalho.

2.2 Definição das tecnologias

Como mencionado anteriormente, a linguagem *C* foi escolhida para desenvolver o projeto e, convenientemente, sistemas *Unix-like* comumente possuem o GCC (GNU, 2024) instalado por padrão, sendo este uma coleção de compiladores da GNU que inclui um compilador eficaz para *C*. Por conta desta praticidade e eficiência, ele foi selecionado para a criação do *software*. Visando estes mesmos benefícios, optou-se por utilizar o *Visual Studio Code* (MICROSOFT, 2024) como editor de código, o qual é uma ferramenta flexível, leve e configurável.

Para garantir a organização e legibilidade do código, é essencial que ele seja estruturado de maneira modular. Nesse contexto, empregou-se o *Make* (GNU, 2024) e o *CMake* (KITWARE, 2024) para automatizar a compilação do projeto, assegurando eficiência e organização na construção do *software*. O *Make* é uma ferramenta de gerenciamento que recompila apenas os arquivos modificados desde a última compilação. Por sua vez, o *CMake* é uma ferramenta multiplataforma capaz de gerar scripts de compilação de forma automatizada, permitindo a construção do projeto em diferentes ambientes com facilidade.

2.3 Abordagem de implementação

Tendo em vista o objetivo do trabalho, que concentra-se na criação de um *software* utilitário, duas abordagens principais podem ser consideradas para seu desenvolvimento: a criação de uma CLI (*Command Line Interface*) e de uma *library* (biblioteca).

As CLIs são mecanismos de *software* capazes de proporcionar uma interface simples para a interação com sistemas, por meio de linhas de comando. Essa comunicação consiste basicamente no fornecimento da operação desejada e dos dados a serem utilizados. Após isso, a CLI trata essas informações e, se estiverem em conformidade com suas descrições, faz o redirecionamento destas para o algoritmo principal. Ao fim da execução, os resultados obtidos são retornados ao usuário. Por conta desse funcionamento direto, percebe-se que essas ferramentas garantem eficiência e velocidade tanto na implementação quanto na utilização.

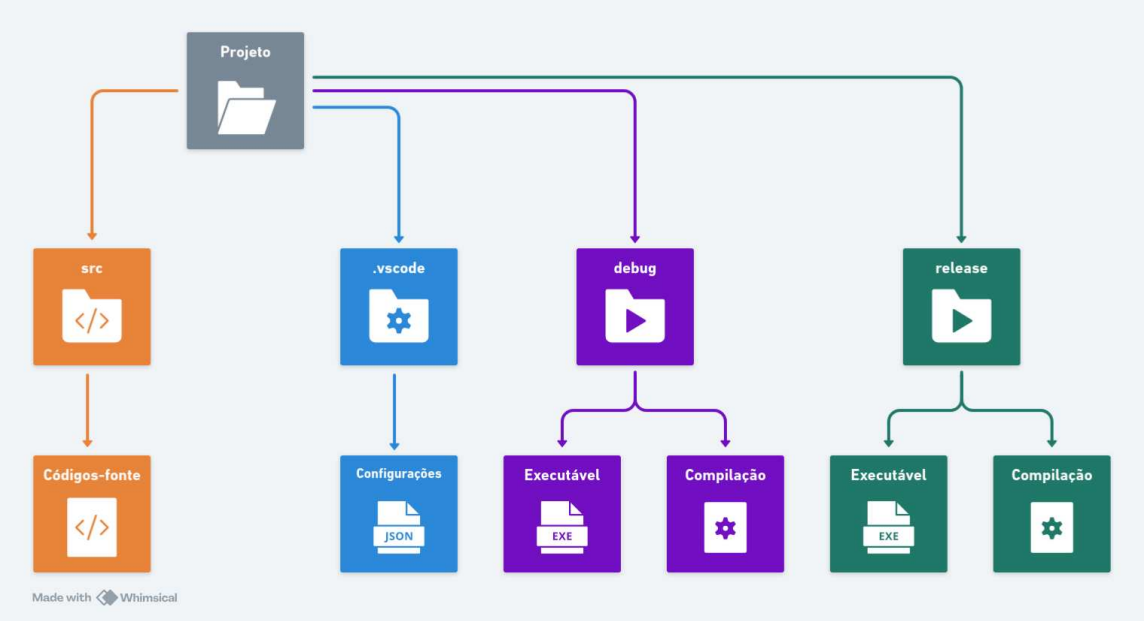
Por outro lado, as *libraries* são componentes independentes de *software*, desenvolvidos para solucionar problemas específicos. Como essas soluções podem ser empregadas em diferentes ambientes, é comum disponibilizá-las para outros projetos, permitindo que suas funcionalidades sejam utilizadas por meio das interfaces definidas no código. Dessa forma, uma *library* oferece aos programadores a possibilidade de reutilizar soluções previamente criadas, otimizando o processo de desenvolvimento.

Como demonstrado, as abordagens apresentadas oferecem benefícios distintos, sendo a CLI mais adequada para usuários finais, enquanto a *library* garante maior flexibilidade para programadores. Por conta disso, e por serem complementares, decidiu-se implementar as duas no trabalho, garantindo todas as vantagens especificadas para o arquivador.

2.4 Arquitetura geral do projeto

O trabalho foi estruturado de maneira eficaz e organizada, atendendo satisfatoriamente aos objetivos propostos. Como ilustrado na Figura 1, foram estabelecidas poucas pastas com nomes claros e distintos.

Figura 1 – Estrutura de diretórios do projeto



Fonte: Autoria própria (2024)

Os diretórios mostrados na Figura 1 foram fundamentais para o desenvolvimento do projeto, e as finalidades de cada um deles são descritas detalhadamente a seguir:

- ***.vscode***: contém configurações de tarefas relacionadas à compilação, além de informações de depuração, como os parâmetros a serem utilizados;
- ***debug***: o executável e os arquivos de compilação da versão de *debug* são direcionados para este diretório;
- ***release***: possui o mesmo propósito da pasta *debug*, porém voltada para as versões de *release*;
- ***src***: armazena os códigos-fonte do projeto, incluindo arquivos de cabeçalho (.h) e de implementação (.c). Dessa forma, essa pasta contém os componentes principais do *software*, reunindo todos os seus algoritmos e lógicas de funcionamento.

Tendo em vista a relevância da pasta ***src***, ela foi subdividida em outros diretórios correspondentes aos módulos da aplicação, visando assegurar a organização e a modularidade do projeto. A seguir, são apresentados os propósitos de cada um deles:

- ***lib***: módulo da *library*, responsável por fornecer aos programadores e à CLI uma interface de acesso às funcionalidades do arquivador. Inclui as funções e os algoritmos responsáveis pelo funcionamento do arquivador, sendo o módulo principal do projeto;
- ***cli***: engloba os códigos relativos à CLI, como suas descrições, tipos e lógicas para o tratamento e a validação dos argumentos recebidos. Ao final de suas operações internas, este módulo chama a função correspondente para processar os dados recebidos, sendo comumente alguma das declaradas na *library*;
- ***utils***: contém funções e tipos com propósitos gerais, sendo utilizados por ambos os outros módulos.

2.5 Estratégia de paralelização

Existem múltiplas estratégias de paralelização que podem ser aplicadas no projeto proposto. Dentre as opções possíveis, escolheu-se o padrão de projeto *Producer-Consumer*, o qual é empregado quando existem tarefas produtoras e consumidoras de dados capazes de atuar simultaneamente. Conforme Ranasinghe (2023), esse método garante o compartilhamento de dados eficiente entre os trabalhadores e ajuda a prevenir condições de corrida, que ocorrem quando duas tarefas tentam modificar dados compartilhados ao mesmo tempo, podendo resultar em erros e comportamentos inesperados.

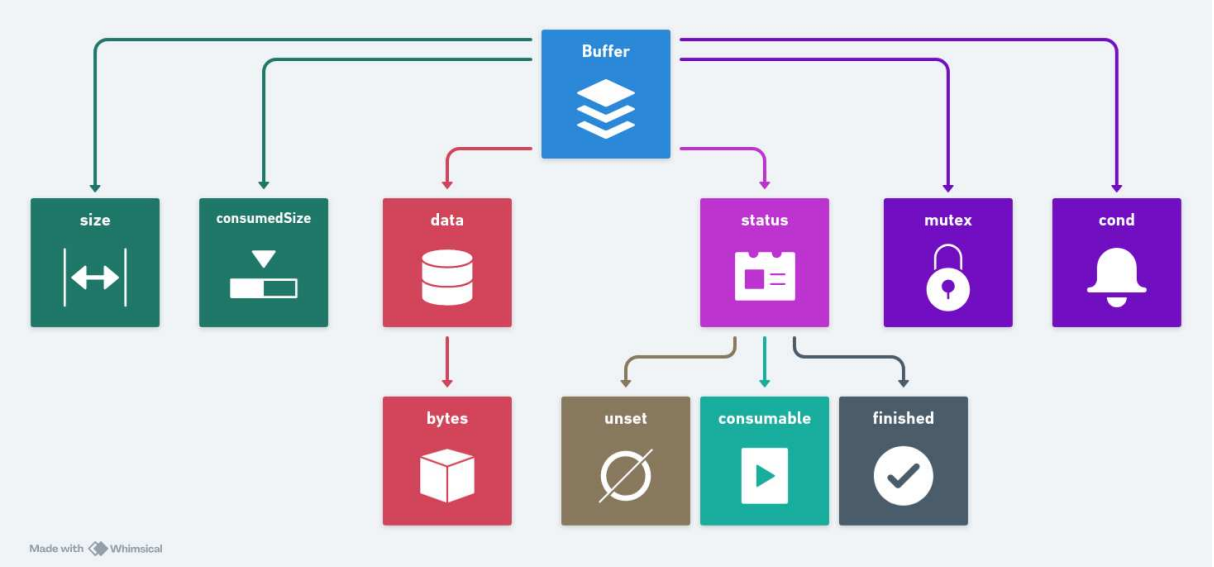
Diante do padrão evidenciado, percebe-se que ele está fortemente associado ao conceito de dados compartilhados, os quais costumam ser retratados como *buffers* no código. Segundo Bhargav (2024), esses mecanismos são definidos como formas de armazenamento temporárias e intermediárias, sendo utilizados quando um componente do sistema fornece dados para outro e, especialmente, se estes possuírem velocidades de trabalho distintas. Em vista disso, é possível notar como a definição geral de *buffer* é abstrata, classificando-os como recursos flexíveis, podendo ser implementados por meio de diferentes formatos e abordagens.

Em vista disso, decidiu-se adotar a seguinte estratégia para o algoritmo: criaram-se duas *threads* de execução, uma representando o *Producer*, responsável por ler os arquivos de entrada e inserir seus dados em *buffers* compartilhados, e outra retratando o *Consumer*, que acessa os conteúdos dos *buffers* e executa as operações de escrita na saída a partir deles. Em meio a esse processo, vale ressaltar a lentidão de requisições de *I/O* ao sistema. Por conta disso, é imprescindível que as leituras e escritas sejam efetuadas em blocos; e não *byte* a *byte*, a fim de minimizar o número de solicitações de *I/O*, e otimizar a execução do arquivador.

2.5.1 Estrutura dos *buffers*

Diante da lógica de paralelização evidenciada, percebe-se como os *buffers* são imprescindíveis nesse processo. Considerando que eles podem ser construídos com diferentes estruturas e propriedades, é importante definir as características a serem utilizadas pelo arquivador. Esses atributos estão retratados na Figura 2.

Figura 2 – Atributos do *buffer*



Fonte: Autoria própria (2024)

Entre as propriedades listadas, a principal é a ***data***, a qual se refere ao espaço de armazenamento do *buffer*, sendo composto por um conjunto de *bytes*. É importante ressaltar que o tamanho suportado deve ser suficientemente grande para otimizar as operações de *I/O*.

Portanto, optou-se por um tamanho de 1.048.576 *bytes*, equivalente a 1 *Megabyte* (MB), que resultou em desempenhos satisfatórios nos testes realizados.

As demais variáveis são empregadas para garantir e facilitar a manipulação dos dados. Sendo assim, o atributo ***size*** indica a quantidade total de *bytes* armazenados, enquanto ***consumedSize*** informa a quantia já consumida. A variável ***status*** especifica o estado do *buffer*, sendo essencial para as *threads* decidirem qual operação executar, podendo assumir os valores: ***unset*** (não inicializado), ***consumable*** (pronto para consumo) ou ***finished*** (finalizado). Por fim, os atributos ***mutex*** e ***cond*** são responsáveis pela sincronização das *threads* no gerenciamento dos *buffers*. O funcionamento dessas propriedades será detalhado mais adiante.

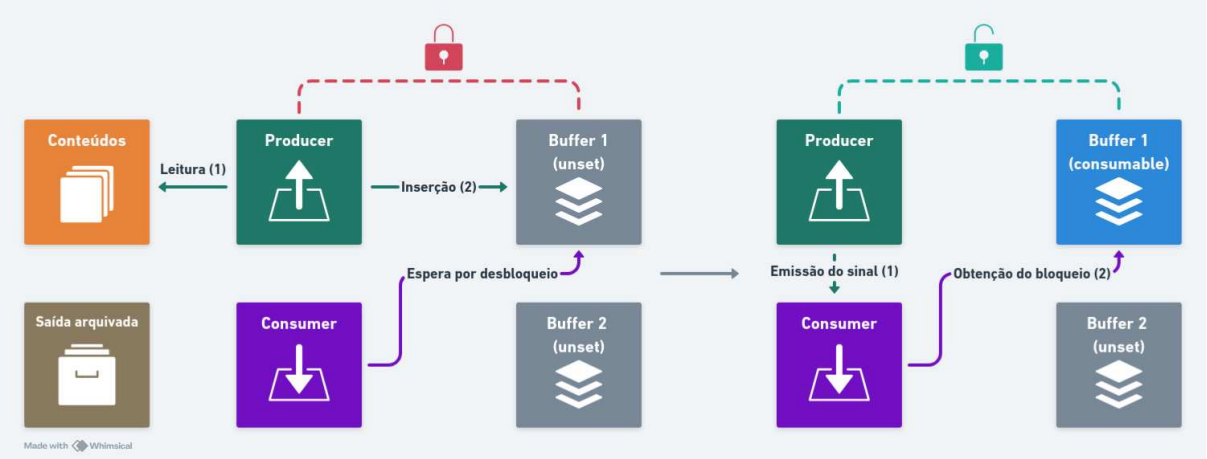
2.5.2 Gerenciamento de *buffers*

Dada a estratégia de paralelização apresentada, é essencial garantir a sincronização entre as *threads*, de modo que elas não manipulem o mesmo *buffer* simultaneamente, evitando erros inesperados. Na aplicação, essa coordenação foi implementada por meio de determinados mecanismos da biblioteca *pthreads*, conhecidos como ***mutex*** e ***cond***, os quais, como mencionado anteriormente, foram incorporados à estrutura do *buffer*. Resumidamente, o ***mutex*** bloqueia o acesso a recursos específicos, permitindo que apenas a *thread* detentora do bloqueio possa utilizar o *buffer* naquele momento. Após finalizar suas operações, a *thread* libera o *buffer* enviando um sinal, por meio do ***cond***, indicando a disponibilidade dos dados para serem usados pela outra *thread*, que estava aguardando.

Em meio ao processo de sincronização, o atributo ***status*** do *buffer* desempenha um papel crucial no controle dos bloqueios e sinais gerenciados pelo ***mutex*** e ***cond***. As *threads* utilizam a informação de estado para determinar se devem aguardar o sinal da outra *thread*, realizar a operação designada no *buffer* ou finalizar sua execução.

Nesse contexto, evidencia-se a diferença de execução entre as *threads*, as quais apresentam cargas de trabalho e velocidades distintas. Dessa forma, é possível que *threads* mais lentas, ou com mais tarefas, bloqueiem os recursos computacionais por períodos maiores que o desejado, impactando negativamente a eficiência da aplicação. Para minimizar esse problema, é ideal balancear a carga de trabalho entre as *threads* e implementar múltiplos *buffers*. Por essa razão, além do balanceamento, optou-se pela utilização de dois *buffers*, cuja lógica de gerenciamento e sincronização é ilustrada a seguir:

Figura 3 – Etapas iniciais da iteração de gerenciamento dos *buffers*

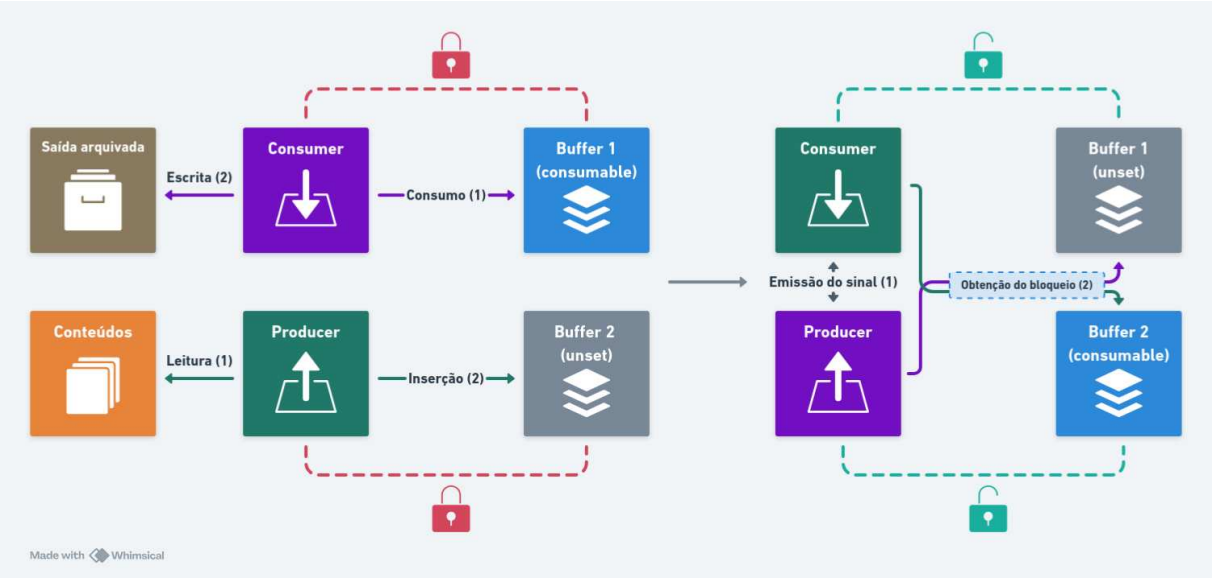


Fonte: Autoria própria (2024)

A Figura 3 demonstra a etapa inicial de comunicação entre as *threads* da aplicação. Nesse estágio, o *Consumer* ainda não obteve o bloqueio de nenhum *buffer* e permanece

aguardando o preenchimento e a liberação do primeiro *buffer* pelo *Producer*. Após a conclusão desse processo, o *buffer* é liberado, tendo seu ***status*** atualizado para ***consumable***, e um sinal é enviado ao *Consumer*, que então requisita o bloqueio para sua utilização. As fases seguintes do gerenciamento são mostradas abaixo.

Figura 4 – Etapas finais da iteração de gerenciamento dos *buffers*



Fonte: Autoria própria (2024)

As etapas retratadas na Figura 4 possuem continuidade direta com as da Figura 3. Desse modo, o *Consumer* obteve o acesso ao primeiro *buffer*, enquanto o *Producer* requisitou e bloqueou o segundo. Nesse instante, as *threads* estão operando paralelamente. Em determinado momento, uma das *threads* finaliza seu trabalho, envia o sinal de liberação para a outra e solicita o acesso ao *buffer* que estava sendo utilizado por esta. Como esse processo pode ocorrer em qualquer ordem, ilustrou-se ambas as *threads* efetuando o procedimento de finalização descrito. Após isso, as etapas demonstradas nas Figuras 3 e 4 voltam a se repetir até que o *Consumer* tenha utilizado todos os dados gerados pelo *Producer*.

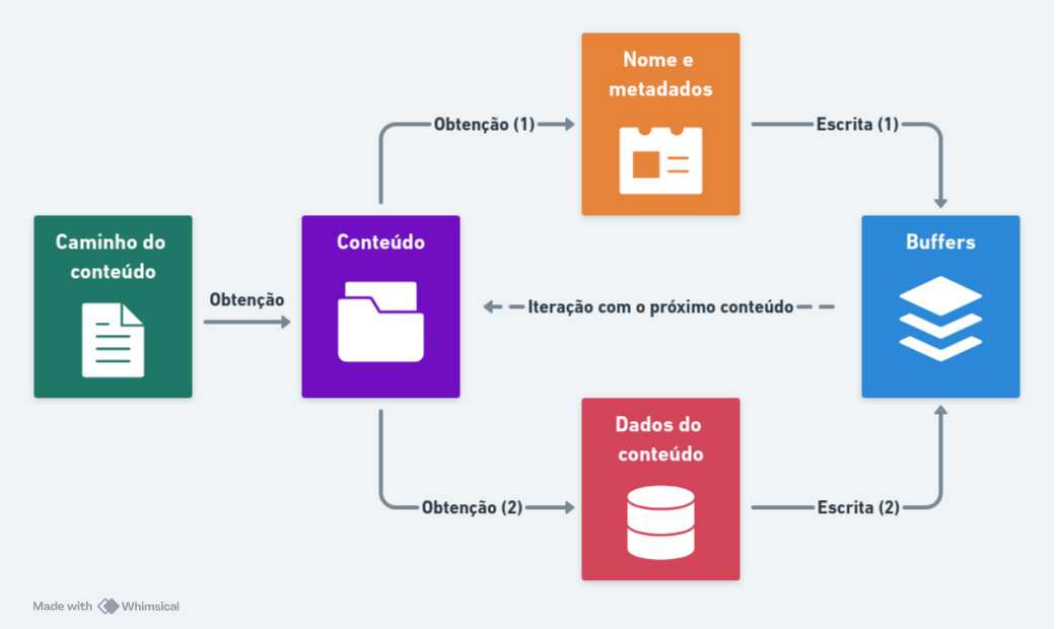
2.6 Processo de arquivamento

Este tópico explica detalhadamente o algoritmo empregado no arquivamento de conteúdos, destacando e distinguindo as operações efetuadas pelo *Producer* e *Consumer*, além de suas interações. No início do código, os *buffers* são inicializados com valores padrão e repassados para as funções das *threads*, juntamente com seus outros argumentos. O *Producer* é responsável por ler os conteúdos de entrada, recebendo os *buffers* criados, os caminhos para os conteúdos e a quantidade deles. Por outro lado, o *Consumer* requisita os *buffers* e o caminho onde o documento arquivado será escrito. Após essa inicialização, cada *thread* segue seu fluxo de execução, os quais são explorados em seus respectivos subtópicos a seguir.

2.6.1 Arquivamento do *Producer*

No contexto do arquivamento, esta *thread* realiza a maioria das operações, manipulando e transformando os conteúdos recebidos para gerar os dados do resultado final. O funcionamento do *Producer* baseia-se em um *loop* principal, ilustrado na Figura 5.

Figura 5 – Algoritmo de arquivamento do *Producer*



Fonte: Autoria própria (2024)

Conforme apresentado, para identificar os conteúdos, seus nomes e metadados são inseridos antes dos dados propriamente ditos. Com isso, a partir dos metadados, é possível identificar se as informações referem-se a um arquivo ou diretório, permitindo a execução de procedimentos distintos de acordo com o tipo. Vale ressaltar que o gerenciamento dos buffers é aplicado nesse processo, mas foi simplificado na figura para facilitar o entendimento.

A Figura 5 oferece uma visão geral do processo de arquivamento, mas as operações variam ligeiramente de acordo com o tipo de conteúdo. Dessa forma, o tratamento de arquivos é o mais simples, seguindo exatamente as etapas demonstradas. Em contrapartida, os diretórios apresentam peculiaridades, pois podem conter outros conteúdos internamente.

Ao processar pastas, primeiramente, o nome e os metadados são colocados nos *buffers*, mas o metadado de tamanho é desconsiderado na inserção, pois, além de não ser necessário para a reconstrução dos conteúdos, sua exclusão otimiza o armazenamento das informações. Em seguida, cada conteúdo interno é iterado, gerando seu caminho completo a partir da concatenação com o caminho da pasta pai. O caminho obtido é então passado para a função de tratamento, que pode ser referente a arquivos ou outras pastas, seguindo os mesmos algoritmos descritos. Ao concluir a inserção de todos os conteúdos internos, o caractere especial de barra inclinada (*slash*) é inserido para indicar o fim dos conteúdos do diretório.

Em resumo, a *thread Producer* segue as etapas apresentadas, inserindo as informações de cada conteúdo recebido nos *buffers* compartilhados. Quando o *buffer* é preenchido completamente, o ***status*** é atualizado de ***unset*** para ***consumable***, e os dados são liberados para o *Consumer*. Ao concluir o processamento do último conteúdo, o ***status*** do *buffer* é alterado para ***finished***, indicando ao *Consumer* que aquele é o último *buffer* da execução.

2.6.2 Arquivamento do *Consumer*

Em contraste ao *Producer*, o *Consumer* tem um único objetivo: carregar os dados dos *buffers* compartilhados para o documento de saída. Para isso, a *thread* abre o arquivo de saída no caminho especificado pelos argumentos, e permanece em *loop* enquanto o ***status*** do *buffer* for diferente de ***finished***. Durante a execução, suas operações consistem em aguardar a liberação do *buffer* e, em seguida, escrever seus dados no arquivo de saída, atualizando o ***status*** para ***unset*** após a escrita, sinalizando que o *Producer* pode utilizá-lo novamente.

2.7 Processo de desarquivamento

O desarquivamento segue uma lógica semelhante à do arquivamento. As operações das *threads* também são claramente definidas: o *Producer* é responsável por ler os dados do documento arquivado, recebendo como parâmetros o caminho desse documento e os buffers; em contrapartida, o *Consumer* tem a tarefa de restaurar os conteúdos arquivados nos diretórios apropriados, utilizando os mesmos *buffers* e o caminho onde os conteúdos serão desarquivados. Após essa configuração inicial, cada *thread* segue suas respectivas rotinas, que serão detalhadas nos subtópicos a seguir.

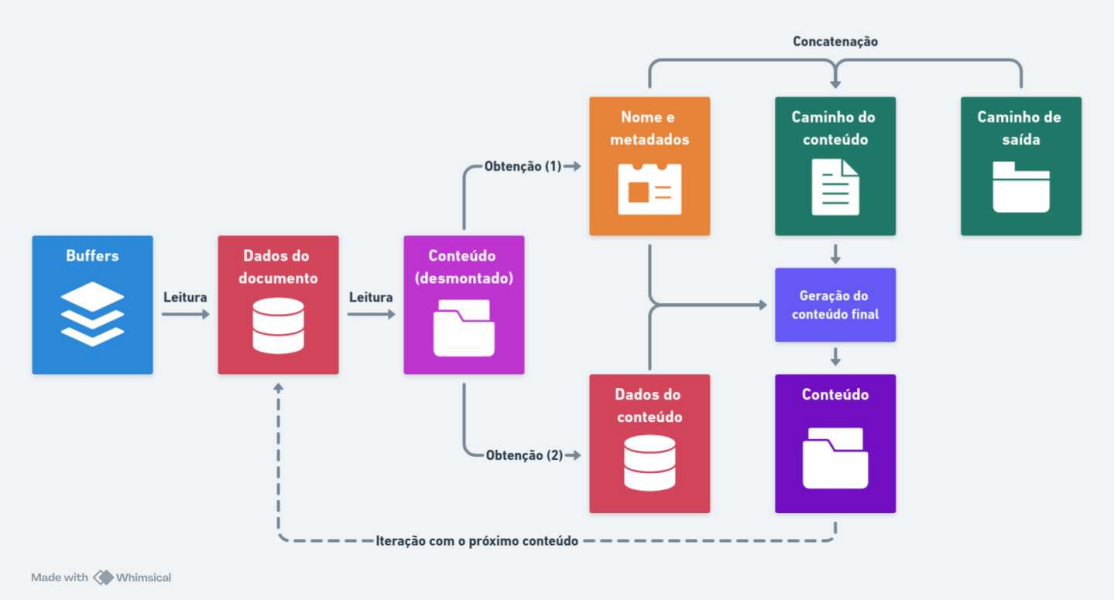
2.7.1 Desarquivamento do *Producer*

No desarquivamento, o papel do *Producer* é mais simples em comparação ao processo de arquivamento, pois, nesse contexto, esta *thread* não é responsável pela preparação dos dados. Sua principal função é ler o documento arquivado e inserir os dados nos *buffers* compartilhados, alterando o ***status*** de ***unset*** para ***consumable*** à medida que os dados são liberados para o *Consumer*. Ao final da leitura completa, o ***status*** do último *buffer* é atualizado para ***finished***, indicando o término da execução.

2.7.2 Desarquivamento do *Consumer*

Nesse processo, diferentemente do arquivamento, o *Consumer* realiza as manipulações efetivas dos dados lidos. Dessa forma, o algoritmo desta *thread* baseia-se em um *loop* principal, que identifica cada um dos conteúdos presentes no documento arquivado e reconstrói as informações originais. As etapas desse processo são demonstradas na Figura 6.

Figura 6 – Algoritmo de desarquivamento do *Consumer*



Fonte: Autoria própria (2024)

Conforme ilustrado, inicialmente são identificados o nome e os metadados do conteúdo sendo lido. Nessa identificação, os metadados não são obtidos de uma só vez; primeiro, verifica-se o tipo do conteúdo, o qual é o primeiro atributo a aparecer entre os dados. Com base nele, determina-se se existe um valor de tamanho a ser obtido, uma vez que

ele não está presente em diretórios. Portanto, a quantidade de metadados lidos nessa primeira fase varia de acordo com o tipo do conteúdo.

Após a identificação do conteúdo, gera-se o caminho completo de saída por meio da concatenação do caminho base recebido com o nome obtido na etapa anterior. Em seguida, procede-se à reconstrução do conteúdo original. No caso de arquivos, esse processo consiste na escrita dos dados dos *buffers* no respectivo arquivo de saída. Como esse tipo de conteúdo possui o atributo de tamanho em seus metadados, esse valor é usado para determinar quantos *bytes* devem ser lidos a seguir, para gerar o arquivo original.

No desarquivamento de diretórios, primeiramente realiza-se sua criação para então iterar-se sobre seus conteúdos internos. Para cada um desses, obtém-se seu caminho completo de saída, a partir do caminho da pasta pai. Além disso, o nome e os metadados do conteúdo são extraídos conforme descrito nas etapas anteriores. Essas informações são repassadas para a função de tratamento, que pode ser referente ao processamento de arquivos ou outras pastas, da mesma maneira evidenciada no algoritmo de arquivamento. Após a reconstrução de cada conteúdo, verifica-se se o próximo *byte* é equivalente à barra inclinada (*slash*), que indica a finalização dos conteúdos do diretório e, por sua vez, o encerramento de seu *loop*.

Ao fim do desarquivamento de cada arquivo e pasta, os metadados obtidos na etapa inicial são transferidos para a versão reconstruída, preservando as informações originais. O único metadado que não é restaurado é o tamanho, pois este é calculado automaticamente pelo sistema e não pode ser modificado pelo código.

Em suma, a *thread Consumer* executa as operações descritas, desarquivando cada um dos conteúdos presentes nos *buffers* do documento arquivado. Quando o *buffer* é completamente consumido, o ***status*** é atualizado de ***consumable*** para ***unset***, indicando que o *Producer* pode preenchê-lo com os próximos dados. Esse procedimento repete-se até o recebimento de um *buffer* com ***status finished***, sinalizando o término da execução.

3 RESULTADOS E DISCUSSÕES

Nesta seção são abordados os resultados alcançados a partir do processo de desenvolvimento explicado, explorando as características do arquivador criado. Além disso, são evidenciados detalhes importantes para a resolução do problema proposto, como as dificuldades técnicas enfrentadas.

3.1 Funcionamento e testes comparativos do arquivador

Este tópico busca analisar diversos aspectos do arquivador desenvolvido, apresentando sua forma de utilização, exemplos de aplicação e comparações de desempenho com as outras alternativas disponíveis do *Unix*.

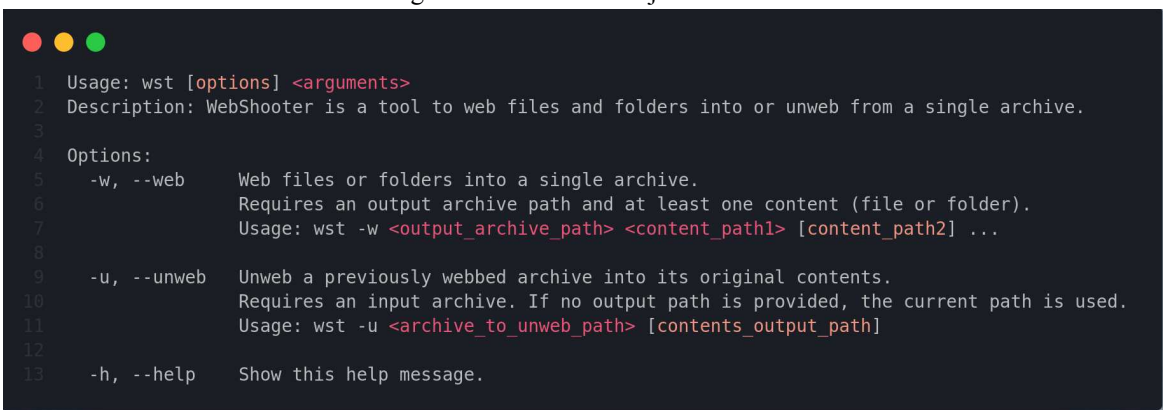
3.1.1 Utilização do arquivador

Como descrito anteriormente, optou-se pelas abordagens de *library* e CLI para a criação do *software*. Para que os programadores empreguem os métodos da *library* em suas aplicações, é necessário baixar os arquivos correspondentes, importar os cabeçalhos e chamar as funções requisitadas, passando os parâmetros indicados nas interfaces. Por outro lado, o uso da CLI exige a geração do arquivo executável, por meio da compilação do projeto, e a chamada deste com os argumentos da operação desejada.

A maioria das CLIs implementa uma funcionalidade de ajuda que exibe o modo de uso detalhado, indicando as opções disponíveis, assim como outras informações, como nome por extenso, descrição e exemplos de uso. Seguindo esse princípio, a CLI desenvolvida

oferece uma interface de auxílio completa e descritiva.

Figura 7 – Interface de ajuda da CLI

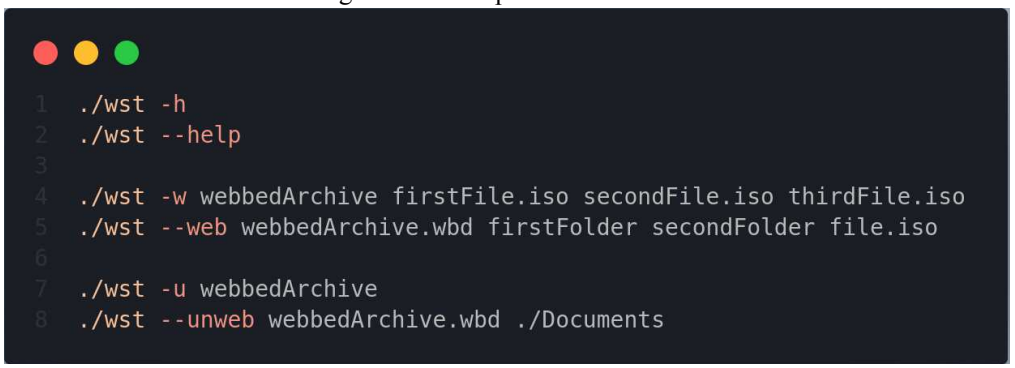


Fonte: Autoria própria (2024)

Como demonstrado na Figura 7, a CLI desenvolvida trabalha com três opções simples: arquivamento (***web***), desarquivamento (***unweb***) e ajuda (***help***). Vale notar que no arquivamento é necessário fornecer o caminho de pelo menos um arquivo ou diretório, enquanto no desarquivamento o caminho de saída dos conteúdos não precisa ser especificado, sendo utilizado o diretório atual do terminal como padrão. Adicionalmente, a funcionalidade de ajuda não requer nenhum argumento para exibir suas instruções.

Embora os exemplos de uso apresentados na interface de ajuda utilizem apenas a letra inicial das opções para referenciá-las, também é possível escrever seus nomes por extenso, precedidos por dois hífens. A seguir, são evidenciados vários exemplos de utilização, contemplando tanto o uso da opção abreviada quanto da forma escrita por completo.

Figura 8 – Exemplos de uso da CLI



Fonte: Autoria própria (2024)

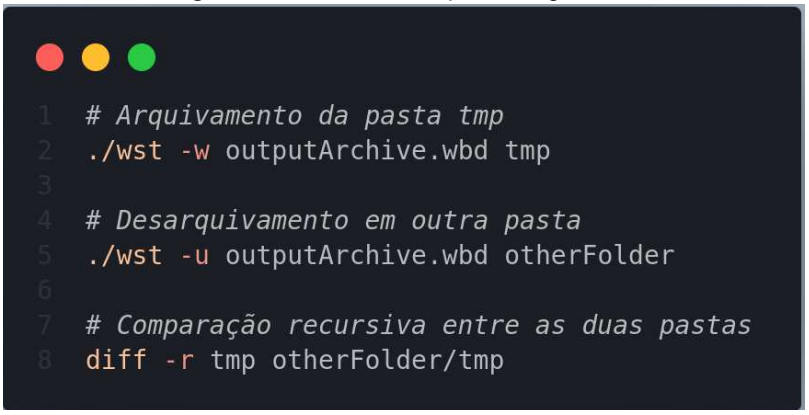
A Figura 8 exibe exemplos variados de uso da CLI, demonstrando operações com arquivos e diretórios distintos. Observa-se que é possível fornecer qualquer nome para os documentos arquivados, incluindo aqueles com extensões fictícias ou sem extensão alguma. Além disso, como mencionado e demonstrado, o caminho de desarquivamento é opcional.

3.1.2 Validação do funcionamento

Para realizar os testes de desempenho, é necessário, primeiramente, comprovar que o arquivador está funcionando adequadamente. Visando validar sua execução, os seguintes passos foram seguidos: arquivou-se uma pasta contendo múltiplos arquivos e subpastas, efetuou-se o desarquivamento do documento gerado em um diretório diferente e, por fim,

utilizou-se a ferramenta *diff* dos sistemas *Unix-like* para verificar se havia alguma diferença entre a pasta original e a desarquivada.

Figura 9 – Teste de validação do arquivador



Fonte: Autoria própria (2024)

O comando *diff* não retorna nenhum resultado no terminal caso os conteúdos comparados sejam idênticos. Ao seguir os passos mostrados na Figura 9, nenhuma mensagem foi exibida, indicando que ambas as operações do arquivador funcionaram corretamente.

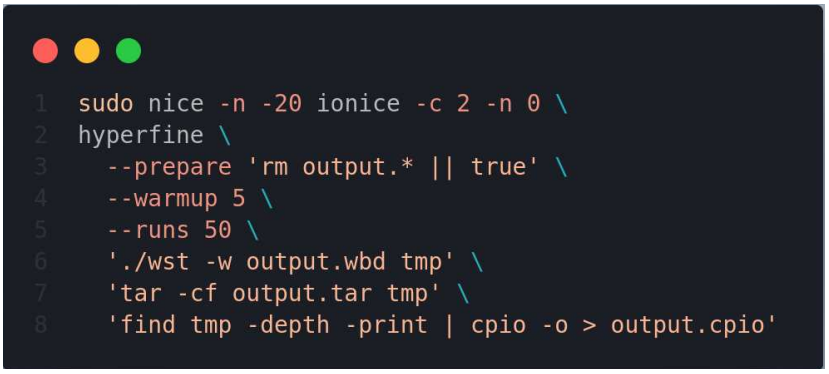
3.1.3 Comparações de desempenho

Para assegurar que o objetivo do trabalho foi alcançado, é necessário efetuar testes comparativos com outros arquivadores dos sistemas *Unix-like*. Para isso, utilizou-se a ferramenta *hyperfine*, a qual é uma CLI de código aberto que permite configurar múltiplos parâmetros para realizar comparações justas entre *softwares*, obtendo a média dos tempos de execução de cada programa e indicando qual deles apresentou a melhor performance, além de quantas vezes foi superior aos demais.

É importante enfatizar que o computador utilizado para os testes foi um **notebook gamer Dell G15**, modelo **G15-I1000-A20P**, equipado com um processador **Intel(R) Core(TM) i5-10500H**, cuja frequência base é de **2,50GHz**, podendo atingir até **4,50GHz**. Esse processador possui **6 núcleos físicos** e **12 *threads*** (**2 por núcleo**). O sistema também conta com **24 GB** de memória RAM **DDR4**, operando a uma frequência de **2933 MHz**. Além disso, o armazenamento é realizado em um **SSD PCIe NVMe M.2**, com velocidade de até **1500 MB/s**.

Para minimizar a interferência de outros processos no sistema e garantir a maior precisão possível dos resultados, aplicaram-se configurações específicas aos testes. Com isso, foram passadas opções dos sistemas *Unix-like* para aumentar a prioridade do processo, além de parâmetros do *hyperfine* para realizar execuções com um número elevado de repetições, assegurando que as médias dos tempos de finalização representassem fielmente a performance de cada *software*. O comando completo utilizado para as comparações de arquivamento é evidenciado na Figura 10.

Figura 10 – Comando de execução das comparações de arquivamento



```
sudo nice -n -20 ionice -c 2 -n 0 \
hyperfine \
  --prepare 'rm output.* || true' \
  --warmup 5 \
  --runs 50 \
  './wst -w output.wbd tmp' \
  'tar -cf output.tar tmp' \
  'find tmp -depth -print | cpio -o > output.cpio'
```

Fonte: Autoria própria (2024)

Como é possível perceber, o comando de comparação com o *hyperfine* é constituído por diversos componentes e configurações, cada um com um propósito específico, conforme descrito a seguir:

- ***sudo***: executa o comando com permissões de administrador, sendo necessário para modificar as prioridades do processo;
- ***nice***: ajusta a prioridade da CPU do processo para a mais alta possível;
- ***ionice***: define a prioridade das operações de *I/O* do processo para a mais alta possível;
- ***hyperfine***: executa a ferramenta responsável pela realização dos testes comparativos de desempenho. Os argumentos a seguir são específicos desse *software*:
    - ***--prepare***: especifica um comando a ser executado antes de cada iteração. Neste caso, utiliza-se a remoção do arquivo de saída para evitar interferências entre execuções;
    - ***--warmup***: define o número de execuções de aquecimento, que são efetuadas sem medições de tempo, com o objetivo de estabilizar o ambiente antes das execuções oficiais;
    - ***--runs***: determina a quantidade de execuções oficiais temporizadas que serão realizadas para cada comando comparado.

Após a definição dos parâmetros listados, são indicados os comandos a serem comparados. Nos experimentos deste trabalho, foram utilizados o arquivador *multithreaded* desenvolvido, o *tar* e o *cpio*, sendo estes últimos amplamente empregados em sistemas *Unix-like*, conforme mencionado anteriormente.

O comando apresentado na Figura 10 foi utilizado como base para as comparações, sendo aplicado também em diferentes cenários. Inicialmente, foram agrupados conjuntos de arquivos idênticos, cada um com aproximadamente **2 *Gigabytes* (GB)** de tamanho, totalizando **2 GB**, **4 GB**, **6 GB**, **8 GB** e **10 GB** de dados. A partir desses conjuntos, procedeu-se à execução dos testes comparativos de arquivamento para cada um deles. Em seguida, os documentos de saída gerados por cada arquivador foram desarquivados, utilizando a mesma lógica, mas com os respectivos comandos de desarquivamento.

Cada um desses testes retornou diversas informações, como os tempos mínimos, máximos e médios de execução, o desvio padrão do tempo médio, qual programa foi o mais rápido e em quantas vezes ele superou os demais. Para simplificar a apresentação desses resultados, eles foram resumidos em velocidades médias das operações.

Para calcular as velocidades médias de arquivamento e desarquivamento, foram extraídos os tempos médios t_i, em segundos, dos arquivadores em cada operação e conjunto de arquivos. Então, obteve-se a velocidade média v_i das tarefas, dividindo o tamanho dos conjuntos T, em *Megabytes* (MB), pelos respectivos tempos médios t_i, conforme a Equação 1.

$$v_i = \frac{T}{t_i} \quad (1)$$

Para encontrar as velocidades médias gerais dos arquivadores, representadas por v_m, somaram-se as velocidades v_i de cada uma de suas tarefas e dividiu-se o valor total pela quantidade de conjuntos de arquivos n, ou seja, cinco. Esta fórmula é expressa na Equação 2.

$$v_m = \frac{\sum_{i=1}^{n} v_i}{n} \quad (2)$$

Os resultados das comparações, obtidos por meio dos procedimentos e cálculos descritos, são apresentados a seguir.

Tabela 1 – Resultados dos testes comparativos entre os arquivadores

Ferramenta	**Velocidade média de Arquivamento**	**Velocidade média de Desarquivamento**
Multi-threaded	1546,89 MB/s	1419,61 MB/s
GNU *tar*	1106,21 MB/s	918,55 MB/s
GNU *cpio*	448,61 MB/s	331,51 MB/s

Fonte: Autoria própria (2024)

Conforme evidenciado na Tabela 1, o arquivador *multi-threaded* alcançou velocidades médias consideravelmente superiores às demais ferramentas, tanto no arquivamento quanto no desarquivamento. Mesmo em comparação com o *tar*, que foi o arquivador mais rápido entre os padrões dos *Unix-like* analisados, o *software* desenvolvido superou sua performance em 40% no arquivamento e 55% no desarquivamento, os quais são resultados significativos.

Com base nos resultados de desempenho apresentados, é possível afirmar que os objetivos do trabalho foram atingidos de maneira satisfatória, oferecendo um arquivador que aproveita o paralelismo para realizar suas operações de forma eficiente e otimizada.

3.2 Dificuldades técnicas

O projeto exigiu o uso de linguagem e mecanismos de baixo nível, como a manipulação de dados em nível de *byte* e o gerenciamento de *threads* com informações compartilhadas. Esses aspectos demandam maior atenção e cuidado durante o desenvolvimento, pois aumentam a probabilidade de erros inesperados.

No contexto das *threads*, como mencionado anteriormente, foi utilizada a biblioteca *pthreads*. Embora básica, ela possui funções que podem ser complexas de entender à primeira vista, especialmente pela quantidade de informações requisitadas em suas chamadas. Tendo isso em mente, o uso dessa biblioteca exigiu estudos detalhados sobre suas interfaces, além de prática para aplicar suas funcionalidades de maneira adequada.

3.3 Propostas de trabalhos futuros

O trabalho desenvolvido é uma versão base, possuindo apenas as funcionalidades básicas de arquivamento e desarquivamento. No entanto, é comum que utilitários desse tipo possuam operações adicionais, como listagem dos conteúdos de documentos arquivados,

concatenação de documentos e um modo verboso, que indica qual arquivo ou pasta está sendo processado. Assim, seria possível expandir o *software* com a adição dessas novas opções.

Outra possibilidade de evolução do projeto seria a criação de uma interface gráfica, proporcionando maior acessibilidade a usuários que não possuem familiaridade com CLIs. Além disso, até mesmo alguns programadores preferem aplicações gráficas pela facilidade de uso e aprendizado. Portanto, a implementação dessa abordagem adicional promoveria benefícios a todos os públicos-alvo do arquivador.

Conforme exposto previamente, é comum que arquivadores sejam utilizados junto com compressores. Tendo isso em vista, seria altamente desejável promover essa integração para o arquivador desenvolvido, especialmente com ferramentas de compressão que também utilizem *multithreading*, visando maximizar o desempenho geral de uso.

4 CONSIDERAÇÕES FINAIS

O presente trabalho atingiu os objetivos estabelecidos de maneira satisfatória, desenvolvendo um arquivador *multi-threaded* para sistemas *Unix-like* com foco na paralelização das operações de entrada e saída de dados. A implementação criada promoveu otimizações significativas no desempenho das operações de arquivamento e desarquivamento, em comparação aos utilitários padrão dos sistemas *Unix-like*, destacando o potencial da abordagem *multithreading* em tarefas *I/O-bound*.

Durante o desenvolvimento, diversos desafios técnicos foram superados, como a sincronização de *threads* e o gerenciamento eficiente de *buffers* compartilhados. Nesse contexto, a escolha da linguagem *C*, demonstrou-se ideal para maximizar a performance e garantir a portabilidade entre sistemas *Unix-like*.

Adicionalmente, a estrutura modular do *software* possibilita sua evolução contínua, facilitando o desenvolvimento de novas funcionalidades. Tendo isso em mente, o arquivador pode expandir-se como uma ferramenta eficiente e versátil, atendendo a diferentes tipos de usuários e aplicações.

REFERÊNCIAS

BAELDUNG. **What's a Buffer**. Disponível em: <https://www.baeldung.com/cs/buffer>. Acesso em: 15 out. 2024.

BUILTIN. **Multithreading vs. Multiprocessing Explained**. Disponível em: <https://builtin.com/data-science/multithreading-multiprocessing>. Acesso em: 15 out. 2024.

BUTENHOF, D. R. **Programming with POSIX Threads**. Boston, MA, USA: Addison Wesley, 1997.

FACEBOOK. **Zstd - Fast Real-time Compression Algorithm**. Disponível em: <https://facebook.github.io/zstd/>. Acesso em: 15 out. 2024.

GAILLY, J. **Pigz - Parallel Implementation of Gzip**. Disponível em: <https://zlib.net/pigz/>. Acesso em: 15 out. 2024.

GNU. **Cpio - GNU Project - Free Software Foundation (FSF)**. Disponível em: <https://www.gnu.org/software/cpio/>. Acesso em: 15 out. 2024.

GNU. **GCC - GNU Compiler Collection**. Disponível em: <https://gcc.gnu.org/>. Acesso em: 15 out. 2024.

GNU. **Make - GNU Project**. Disponível em: <https://www.gnu.org/software/make/>. Acesso em: 15 out. 2024.

GNU. **Tar - GNU Project - Free Software Foundation (FSF)**. Disponível em: <https://www.gnu.org/software/tar/>. Acesso em: 15 out. 2024.

MATNEY, J.; SHIPMAN, G. **Parallelism in System Tools**. Disponível em: <https://cug.org/5-publications/proceedings_attendee_lists/CUG10CD/pages/1-program/final_program/CUG10_Proceedings/pages/authors/16-18Thursday/18A-Matney-paper.pdf>. Acesso em: 15 out. 2024.

MAY, J. M. **Parallel I/O for High Performance Computing**. Google Livros. Disponível em: <https://books.google.com.br/books?id=iLj516DOIKkC>. Acesso em: 15 out. 2024.

MICROSOFT. **Visual Studio Code**. Disponível em: <https://code.visualstudio.com/>. Acesso em: 15 out. 2024.

PERFORCE. **What is Parallel Programming and Multithreading**. Disponível em: <https://www.perforce.com/blog/qac/multithreading-parallel-programming-c-cpp>. Acesso em: 15 out. 2024.

RANASINGHE, N. **Design Patterns for Concurrent Programming: Producer-Consumer Pattern**. Disponível em: <https://medium.com/@nirajranasinghe/design-patterns-for-concurrent-programming-producer-consumer-pattern-39193cac195a>. Acesso em: 15 out. 2024.

SHANTHI, R.; IRUDHAYARAJ, M. **Multithreading - An Efficient Technique for Enhancing Application Performance**. Disponível em: <https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=d0fb20ae4e3e1ca92474691ce900d28164ca9886>. Acesso em: 15 out. 2024.

SHARKDP. **Hyperfine - A Benchmarking Tool**. Disponível em: <https://github.com/sharkdp/hyperfine>. Acesso em: 15 out. 2024.

SIM, R. **CPU-bound, I/O-bound**. Disponível em: <https://medium.com/@sim30217/cpu-bound-i-o-bound-ea3c92499e4c>. Acesso em: 15 out. 2024.

THE OPEN GROUP. **Open group threads**. Disponível em: <https://pubs.opengroup.org/onlinepubs/7908799/xsh/threads.html>. Acesso em: 15 out. 2024.

THAKUR, R. **High Performance Parallel I/O**. Google Livros. Disponível em: <https://books.google.com.br/books?id=HJLaBAAAQBAJ>. Acesso em: 15 out. 2024.