UNIVERSIDADE FEDERAL DO MARANHÃO

CIÊNCIA DA COMPUTAÇÃO – DEIN0114

Willy Kauã Diniz Teixeira

Glezier Montalvane de Farias Ferreira

Edmar Miqueias Carvalho Vieira

Luis Eduardo do Rosario Fonseca

Projeto Final: Simulação de Técnicas de Alocação de Arquivos

São Luís – MA

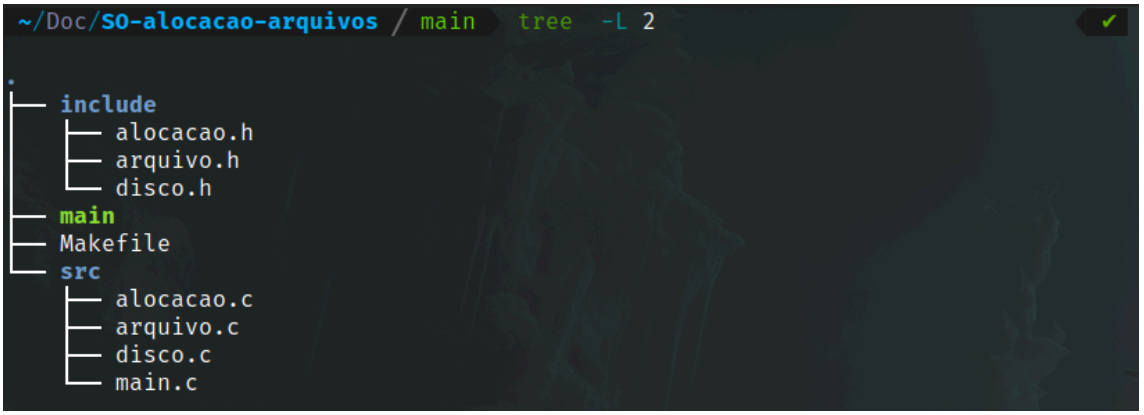
06 de Janeiro, 2026

INTRODUÇÃO

O presente relatório documenta o desenvolvimento do projeto final da disciplina de Sistemas Operacionais I, cujo objetivo central é a implementação e análise de uma simulação de técnicas de alocação de arquivos. Optou-se por usar a linguagem de programação C. O programa simula um disco, como um array de “Arquivos”, e mostra seu estado após inserção e remoção de arquivos. Para esse trabalho foram abordados os tipos de alocação contíguo, encadeado e indexado.

IMPLEMENTAÇÃO

A estrutura do projeto adota uma abordagem modular. O código é organizado em arquivos fonte e seus respectivos cabeçalhos (headers), que definem as estruturas de dados (structs) e as funções de manipulação do sistema. A interação com o usuário é centralizada na main, que implementa uma interface de terminal para inserir, remover e visualizar arquivos no disco simulado. Segue a estrutura:



O disco foi modelado como um vetor (array) de estruturas do tipo Blocos. Cada elemento deste vetor representa uma unidade de alocação e é composto por três campos fundamentais: uma instância da estrutura Arquivo, que armazena os dados ou metadados pertinentes; uma variável binária (flag) para indicar o status de ocupação do bloco; e uma referência para o bloco sucessor, utilizada especificamente para manter a integridade lógica na técnica de alocação encadeada.

Obs: Não serão detalhadas as implementações desses arquivos; dar-se-á ênfase à abordagem dos três tipos de alocação.

ALOCAÇÃO CONTÍGUA

A lógica é direta: percorre o disco buscando um espaço vazio sequencial, se achar, aloca todo o arquivo lado a lado, caso contrário, não aloca.

```
for (int i = 0; i < TAM_DISCO; i++) { // percorre todo o disco
  if (!disco[i].ocupado) { // se o bloco estiver livre, incrementa o contador
    if (livres == 0) { // se é o primeiro bloco livre encontrado, salva o índice
      inicio = i;
    }
    livres++;
```

No momento em que a variável livres se torna exatamente igual ao tamanho do arquivo, ele aloca:

```
    if (livres == arquivo->tamanho) { // se encontrou espaço suficiente, aloca
      for (int j = inicio; j < arquivo->tamanho + inicio; j++) {
        disco[j].ocupado = 1;
        disco[j].arquivo = arquivo;
      }
      arquivo->blocoInicial = inicio; // basta o índice do primeiro bloco
      arquivo->tipo = CONTIGUA;
      return TRUE;
```

ALOCAÇÃO ENCADEADA

A abordagem aqui é um pouco diferente, à medida que blocos desocupados são encontrados, a alocação é realizada imediatamente, mesmo que não sejam sequenciais.

```
  // Percorre todo o disco ou até preencher o arquivo
  for (int i = 0; i < TAM_DISCO && indicesOcupados < arquivo->tamanho; i++) {
    if (!disco[i].ocupado) { // se o bloco estiver livre, aloca
      disco[i].ocupado = 1;
      disco[i].arquivo = arquivo;
      indicesOcupados++;
```

Cada bloco precisa ter uma referência para o próximo bloco onde continua o arquivo, guardamos isso da seguinte forma:

```
// se é o primeiro bloco alocado, salva o índice e como o anterior para o próximo
if (!achouOprimeiro) {
  arquivo->blocoInicial = i;
  achouOprimeiro = TRUE;
  anterior = i;
} else { // se não for o primeiro bloco,
  // liga o bloco anterior ao atual
  disco[anterior].proximo = i;
  anterior = i;
```

No fim, se não foi possível alocar todo o arquivo, removemos a alocação do que já foi alocado:

```
while (atual != -1) { // percorre a lista encadeada desalocando
    int prox = disco[atual].proximo;
    disco[atual].ocupado = 0;
    disco[atual].arquivo = NULL;
    disco[atual].proximo = -1;
    atual = prox;
}
```

ALOCAÇÃO INDEXADA

A lógica dessa alocação é bem similar à encadeada, a diferença é que não possui referência para o próximo bloco. Primeiro, busca-se um espaço desocupado no bloco. Ao encontrar, já aloca-se espaço para o arquivo, que corresponde ao bloco de índice.

```
// Encontra um bloco livre para o índice
for (int i = 0; i < TAM_DISCO; i++) {
    if (!disco[i].ocupado) {
        blocoIndice = i;
        disco[i].ocupado = 1;
        disco[i].arquivo = arquivo;
        break; // para após encontrar o primeiro bloco livre
    }
}
```

Depois, se há ao menos um espaço livre no bloco, busca-se mais espaços livres para continuar a alocação do arquivo por completo. É importante destacar que, quando uma posição é ocupada, o seu endereço é salvo no bloco de índices, permitindo assim acessá-la para remoção.

```
// percorre todo o disco ou até preencher o arquivo
for (int i = 0; i < TAM_DISCO && indicesOcupados < arquivo->tamanho; i++) {
    if (!disco[i].ocupado) { // se o bloco estiver livre, aloca
        disco[i].ocupado = 1;
        disco[i].arquivo = arquivo;
        arquivo->blocos[indicesOcupados] = i; // salva o índice do bloco alocado no bloco de dados
        indicesOcupados++;
    }
}
```

Caso exista espaço suficiente no bloco para todo o arquivo, a alocação é realizada com sucesso e retorna TRUE. Caso contrário, o que já foi alocado é desalocado e retorna FALSE.

```
// Se não conseguiu alocar todo o arquivo, desfaz a alocação
if (indicesOcupados < arquivo->tamanho) {
    disco[blocoIndice].ocupado = 0; // desaloca o bloco de índice
    disco[blocoIndice].arquivo = NULL;

    // desaloca os blocos de dados já alocados
    for (int j = 0; j < indicesOcupados; j++) {
        int aux = arquivo->blocos[j];
        disco[aux].ocupado = 0;
        disco[aux].arquivo = NULL;
    }
    return FALSE;
}

arquivo->blocoInicial = blocoIndice; // índice
arquivo->tipo = INDEXADA;
return TRUE;
```

REMOVER ARQUIVO

A função de remoção de arquivos opera de acordo com o tipo de alocação realizada para o arquivo alvo de remoção.

Para um arquivo alocado de forma contígua, busca-se o primeiro bloco alocado para o arquivo e desaloca em sequência, a partir desse bloco inicial, até o tamanho total do arquivo.

```
case CONTIGUA: {
  for (int i = arquivo->blocoInicial; i < arquivo->blocoInicial + arquivo->tamanho; i++) {
    disco[i].ocupado = 0;
    disco[i].arquivo = NULL;
  }
  break;
}
```

Para alocação de forma encadeada, começa desalocando pelo espaço inicial do arquivo. Depois percorre o bloco sempre buscando desalocar o próximo espaço alocado do arquivo, seguindo a ordem dos ponteiros. Encerra quando encontra o fim do arquivo denotado por -1.

```
case ENCADEADA: {
  int atual = arquivo->blocoInicial;
  while (atual != -1) {
    int prox = disco[atual].proximo;
    disco[atual].ocupado = 0;
    disco[atual].arquivo = NULL;
    disco[atual].proximo = -1;
    atual = prox;
  }
  break;
}
```

Por fim, em alocação de forma indexada, percorre o bloco de espaços e desaloca seguindo a tabela de índices, parando ao encontrar o tamanho do arquivo alocado.

```c
case INDEXADA: {
  for (int i = 0; i < arquivo->tamanho; i++) {
    int bloco = arquivo->blocos[i];
    if (bloco != -1) {
      disco[bloco].ocupado = 0;
      disco[bloco].arquivo = NULL;
    }
  }

  int blocoIndice = arquivo->blocoInicial;
  disco[blocoIndice].ocupado = 0;
  disco[blocoIndice].arquivo = NULL;
  break;
}
```

Ao final da função para remoção, existe um código para limpar os metadados do arquivo após sua remoção, garantindo que ele não mantenha referências a blocos já desalocados.

```c
// Limpa metadados do arquivo
arquivo->blocoInicial = -1;
arquivo->tipo = -1;
for (int i = 0; i < arquivo->tamanho; i++) {
  arquivo->blocos[i] = -1;
}
```

COMO EXECUTAR

O projeto foi desenvolvido em ambiente Linux, portanto, o mais adequado seria utilizar o mesmo sistema operacional para rodar o código. Há funções nativas de Linux como system(“clear”).

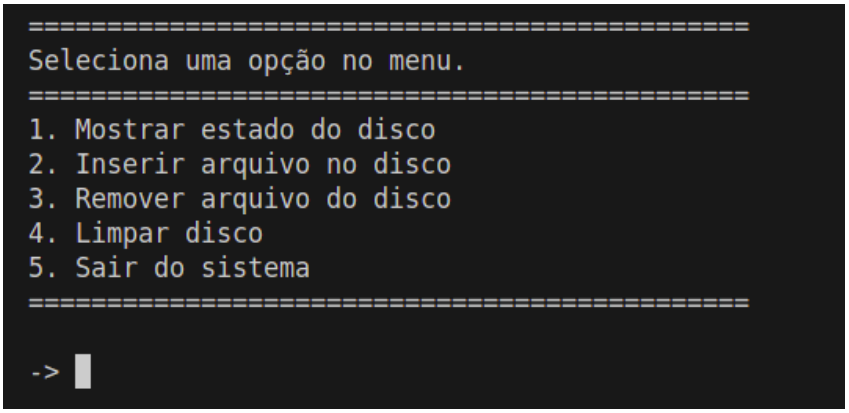
Foi utilizada uma função de espera (sleep). No Windows, precisa importar a biblioteca time.h e, no Linux, precisa-se importar a biblioteca unistd.h

O projeto também tem como pré-requisitos um compilador para linguagem C. Como são utilizados vários arquivos, foi adotado um makefile para automatizar a compilação de todos os arquivos. Para compilar, é necessário apenas o comando “make” e, para rodar o projeto, usa-se o comando “make run”.

Após compilado e rodando, há uma interface para confirmação de início de alocação.

```
==============================================
Bem-vindo ao sistema de alocação de arquivos!
==============================================
Deseja iniciar o sistema? (1 - Sim, 0 - Não):
```

Após confirmação, navega-se através da interface de escolhas:

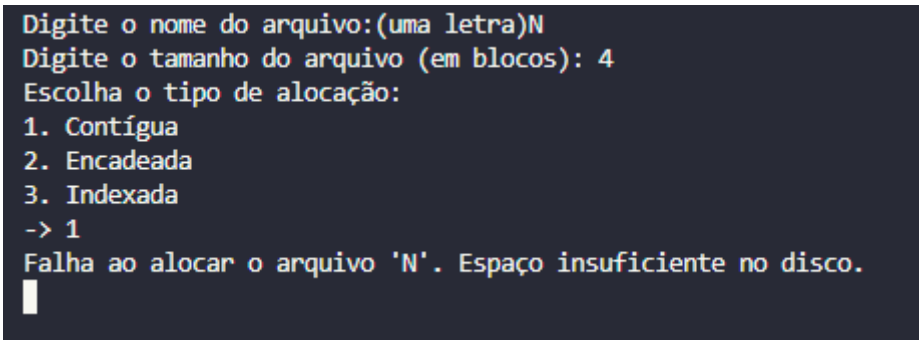


TESTE E ANÁLISE

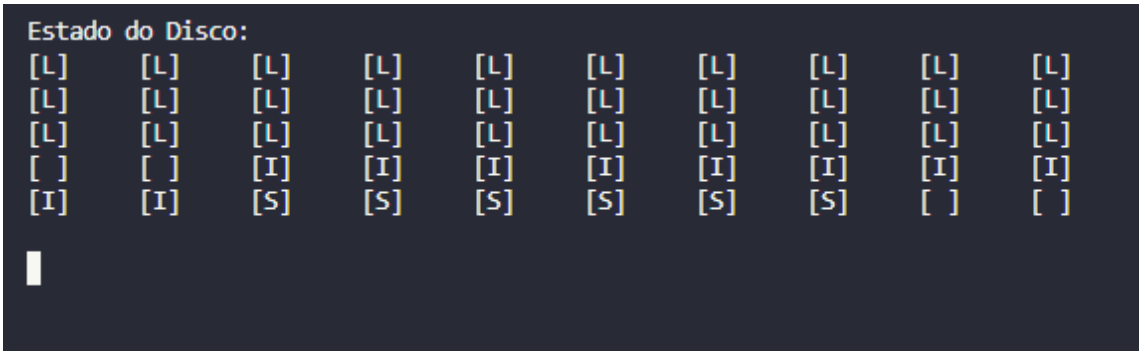
Foram utilizados alguns exemplos que demonstram na prática os três tipos de alocação, enfatizando seus pontos positivos e negativos.

Alocação Contígua

Utilizando um exemplo em que temos 4 blocos livres na memória e tentamos alocar um arquivo de tamanho 4 usando alocação contígua, a situação é a seguinte: apesar de existirem os 4 blocos livres, eles não estão dispostos de forma sequencial. Ou seja, há espaço disponível no disco, porém não de forma contínua.



Resultado:



Nesse cenário, o algoritmo de alocação contígua não consegue realizar a alocação e retorna uma mensagem de erro. Esse comportamento representa o problema da fragmentação externa, onde o espaço livre existe, mas não pode ser aproveitado devido à sua distribuição na memória.

Em contraste, utilizando a alocação encadeada, esse mesmo espaço livre poderia ser facilmente utilizado, já que os blocos não precisam ser contíguos, resultando em uma alocação bem-sucedida.

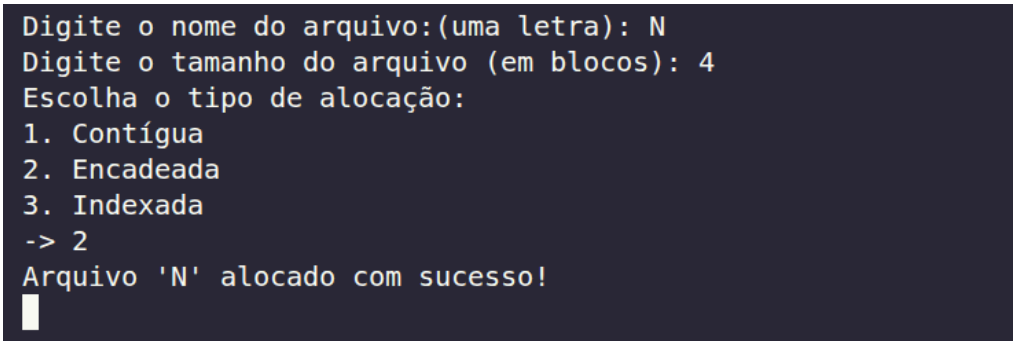
Dessa forma, os pontos positivos da alocação contígua são o acesso simples e rápido, pois os blocos são sequenciais, além de possuir uma estrutura simples e de fácil implementação.

Entretanto, seus pontos negativos incluem justamente a fragmentação externa, em que espaços livres entre blocos não podem ser utilizados, e a necessidade de espaço contíguo, que pode impossibilitar a alocação mesmo quando há espaço livre suficiente no disco.

Alocação Encadeada

Os principais pontos positivos da alocação encadeada estão relacionados à sua flexibilidade e eficiência no uso do espaço. Como os blocos não precisam ser contíguos, o algoritmo consegue alocar arquivos sempre que houver espaço livre disponível, eliminando o problema da fragmentação externa. Isso facilita o crescimento dos arquivos e permite uma melhor utilização da memória. Além disso, o processo de alocação é simples e rápido, já que qualquer bloco livre pode ser utilizado, independentemente de sua posição no disco.

Vamos replicar o mesmo cenário do caso anterior, mas dessa vez usando alocação encadeada:



Resultado:

```
Estado do Disco:
[L]  [L]  [L]  [L]  [L]  [L]  [L]  [L]  [L]  [L]
[L]  [L]  [L]  [L]  [L]  [L]  [L]  [L]  [L]  [L]
[L]  [L]  [L]  [L]  [L]  [L]  [L]  [L]  [L]  [L]
[N]  [N]  [I]  [I]  [I]  [I]  [I]  [I]  [I]  [I]
[I]  [I]  [S]  [S]  [S]  [S]  [S]  [S]  [N]  [N]
```

Por outro lado, os pontos negativos estão associados ao tempo de acesso. Para acessar o bloco N de um arquivo, é necessário acessar primeiro o bloco N−1, o que pode tornar a operação lenta em arquivos grandes. Outro aspecto negativo é o sobrecarga de memória, pois cada bloco precisa armazenar um ponteiro para o próximo, aumentando o consumo de espaço.

Alocação Indexada

Entre os pontos positivos da alocação indexada, destaca-se o fato de que o algoritmo não sofre com fragmentação externa: havendo espaço livre no disco, a alocação é sempre possível. Outro ponto importante é o acesso direto aos blocos, já que não é necessário percorrer os blocos anteriores; basta consultar a tabela de índices para acessar qualquer bloco do arquivo.

Como pontos negativos, o algoritmo exige a utilização de uma estrutura adicional, a tabela de índices. Nos testes realizados, ao alocar um arquivo de tamanho 4, ele passa a ocupar 5 blocos no disco, sendo um deles dedicado exclusivamente ao índice. Além disso, arquivos pequenos podem gerar desperdício de espaço, pois apenas alguns ponteiros do bloco de índice são utilizados, enquanto o restante permanece inutilizado. Nesse contexto, podemos aproveitar para demonstrar uma situação de falha onde há espaço suficiente para o arquivo, mas não para o bloco de índice:

```
Digite o nome do arquivo:(uma letra): N
Digite o tamanho do arquivo (em blocos): 4
Escolha o tipo de alocação:
1. Contígua
2. Encadeada
3. Indexada
-> 3
Falha ao alocar o arquivo 'N'. Espaço insuficiente no disco.
```

Analisando os três algoritmos, é possível observar que cada um apresenta vantagens e desvantagens específicas. A alocação contígua se destaca pela simplicidade e rapidez de acesso, porém sofre com fragmentação externa. A alocação encadeada resolve esse

problema e utiliza melhor o espaço, mas apresenta desempenho inferior no acesso aos dados. Já a alocação indexada oferece um bom equilíbrio entre flexibilidade e desempenho, ao permitir acesso direto e eliminar a fragmentação externa, embora introduza sobrecarga adicional.

Dessa forma, considerando o uso no dia a dia de sistemas operacionais, a alocação indexada tende a ser a mais eficiente, pois combina bom desempenho de acesso com melhor aproveitamento do espaço, características fundamentais para sistemas modernos que lidam com grande volume e variedade de arquivos.

CONCLUSÃO

Este projeto permitiu compreender, de forma prática, o funcionamento dos três tipos de alocação de arquivos: contígua, encadeada e indexada. A implementação possibilitou observar as principais vantagens e limitações de cada método, evidenciando como a escolha da estratégia de alocação impacta o uso do espaço em disco e o acesso aos dados. Assim, o desenvolvimento do sistema contribuiu para consolidar os conceitos teóricos estudados em Sistemas Operacionais por meio de uma abordagem prática e comparativa.