



PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS

ICEI – Instituto de Ciências Exatas e Informática

Departamento de Ciência da Computação – *campus* Poços de Caldas

João Henrique dos Santos Ferreira

Marcelle Andrade Pereira

TRABALHO DE PIPES MAIS THREADS: TRÁFEGO AÉREO

Poços de Caldas

2024

João Henrique dos Santos Ferreira
Marcelle Andrade Pereira

TRABALHO DE PIPES MAIS THREADS: TRÁFEGO AÉREO

Este documento representa o trabalho de **Pipes mais Threads**, pertinente ao curso de Ciência da Computação, *campus* Poços de Caldas da Pontifícia Universidade Católica de Minas Gerais - PUC, como requisito parcial para aprovação na matéria de Sistemas Operacionais, código 5324.1.00.

Professor Dr.: João Carlos de M. Morcelli Jr.

João Henrique dos Santos Ferreira
Marcelle Andrade Pereira

TRABALHO DE PIPES MAIS THREADS: TRÁFEGO AÉREO

Este documento representa o trabalho de **Pipes mais Threads**, pertinente ao curso de Ciência da Computação, *campus* Poços de Caldas da Pontifícia Universidade Católica de Minas Gerais, como requisito parcial para aprovação na matéria de Sistemas Operacionais, código 5324.1.00.

Prof. Dr. João Carlos de Moraes Morcelli Junior (Orientador/Avaliador)

Poços de Caldas, 19 de novembro de 2024

RESUMO

Este trabalho apresenta o desenvolvimento de uma aplicação em linguagem C pura, focada em demonstrar conceitos de Sistemas Operacionais. A aplicação utiliza IPC (*Inter-Process Communication*) Pipes para a troca de mensagens entre subprocessos e threads para contabilização, abordando de forma prática a sincronização e a gestão de processos paralelos. O tema escolhido, "Tráfego Aéreo", modela a torre de controle e aviões como subprocessos, enquanto threads coordenam a contagem de aeronaves. Todo o código foi implementado para compilação no ambiente Linux, utilizando o GCC, atendendo aos requisitos da disciplina de Sistemas Operacionais do curso de Ciência da Computação da PUC Minas.

O trabalho está estruturado em quatro capítulos que detalham os objetivos, a lógica do algoritmo, a implementação do código e os resultados obtidos, ilustrados por capturas de tela. Além disso, o arquivo-fonte do código está disponível em repositório público no GitHub. A metodologia aplicada incluiu o uso do Visual Studio Code, GitHub e terminal Linux, assegurando organização, eficiência e funcionalidade.

Palavras-chaves: sistemas operacionais, IPC *pipes*, *threads*, comunicação entre processos, linguagem C, tráfego aéreo, desenvolvimento de software, GCC Linux.

ABSTRACT

This work presents the development of an application in pure C, focused on demonstrating Operating Systems concepts. The application utilizes IPC (Inter-Process Communication) Pipes for message exchange between subprocesses and threads for counting, practically addressing synchronization and management of parallel processes. The chosen theme, "Air Traffic," models the control tower and airplanes as subprocesses, while threads coordinate the counting of aircraft. The entire code was implemented for compilation in a Linux environment using GCC, fulfilling the requirements of the Operating Systems course in the Computer Science program at PUC Minas.

The work is structured into four chapters that detail the objectives, algorithm logic, code implementation, and the results obtained, illustrated with screenshots. Additionally, the source code file is available in a public repository on GitHub. The applied methodology included the use of Visual Studio Code, GitHub, and the Linux terminal, ensuring organization, efficiency, and functionality.

Keywords: operating systems, IPC pipes, threads, inter-process communication, C language, air traffic, software development, GCC Linux.

SUMÁRIO

1.	INTRODUÇÃO	7
2.	OBJETIVOS	8
2.1.	Objetivo Geral	8
2.2.	Objetivos Específicos	8
3.	DESCRIÇÃO DA APLICAÇÃO	9
3.1.	Visão Geral	9
3.2.	Estrutura Geral	9
3.3.	Fluxo Geral	11
3.4.	Organização do código	11
3.5.	Etapas de Desenvolvimento	12
4.	DESCRIÇÃO DO CÓDIGO	14
4.1.	Explicação Geral	14
4.1.1.	<i>Bibliotecas</i>	<i>14</i>
4.1.2.	<i>Definição de Máximos</i>	<i>14</i>
4.1.3.	<i>Strucs – Estrutura de Dados</i>	<i>15</i>
4.1.4.	<i>Assinaturas das funções</i>	<i>17</i>
4.1.5.	<i>Variáveis Globais e inicialização</i>	<i>18</i>
4.1.6.	<i>Função int main()</i>	<i>20</i>
4.1.7.	<i>Função Avião</i>	<i>25</i>
4.1.8.	<i>Função Requisição</i>	<i>30</i>
4.1.9.	<i>Função thread_function</i>	<i>36</i>
4.1.10.	<i>Função handle_sigint</i>	<i>38</i>
4.1.11.	<i>Função de inicialização do contador</i>	<i>39</i>
4.2.	Funcionamento da aplicação – Resumo	40
4.2.1.	<i>Inicialização do Sistema</i>	<i>40</i>
4.2.2.	<i>Estrutura da Torre de Controle</i>	<i>41</i>
4.2.3.	<i>Funcionamento dos Aviões</i>	<i>41</i>
4.2.4.	<i>Processo de Pouso:</i>	<i>41</i>
4.2.5.	<i>Processo de Decolagem</i>	<i>41</i>
4.2.6.	<i>Monitoramento e Estatísticas</i>	<i>42</i>
4.2.7.	<i>Registro de Operações</i>	<i>42</i>
4.2.8.	<i>Tratamento de Interrupções</i>	<i>42</i>
4.2.9.	<i>Diagrama esquemático</i>	<i>42</i>
5.	TELAS (PRINTS) DA EXECUÇÃO	44
6.	REFERENCIAS	47

1. INTRODUÇÃO

O presente trabalho aborda o desenvolvimento de uma aplicação em linguagem C pura, que utiliza comunicação entre processos e threads para demonstrar conceitos fundamentais de Sistemas Operacionais.

A aplicação foi projetada para explorar as funcionalidades de IPC (*Inter-Process Communication*) Pipes na troca de mensagens entre subprocessos, aliada ao uso de threads para gerenciar a contagem de aeronaves. Todo o código foi implementado seguindo os padrões exigidos para compilação no ambiente Linux, utilizando o GCC, em conformidade com os requisitos da disciplina de Sistemas Operacionais do curso de Ciência da Computação da Pontifícia Universidade Católica de Minas Gerais (PUC Minas).

Estes documentos detalhar a construção e o funcionamento da aplicação, apresentando a lógica subjacente, as etapas de desenvolvimento e a organização do código. Além da documentação descritiva, o trabalho conta com o arquivo-fonte do código disponibilizado em um repositório público no GitHub, disponível através do link: <https://github.com/Marcelleap/TrafegoAereo>.

Este documento é dividido em quatro capítulos principais. O primeiro capítulo apresenta os objetivos e o tema do trabalho, enquanto o segundo capítulo oferece uma visão geral da proposta. No terceiro capítulo, são detalhados os aspectos do código, com ênfase na lógica implementada. Finalmente, o quarto capítulo reúne capturas de tela que demonstram o comportamento do sistema durante a execução.

A metodologia utilizada incluiu ferramentas como Visual Studio Code para a edição do código, GitHub para o controle de versão e o terminal Linux para a realização de testes e compilação.

2. OBJETIVOS

2.1. Objetivo Geral

O objetivo deste trabalho pauta-se em desenvolver uma aplicação em linguagem C pura que implemente os conceitos de threads e subprocessos, utilizando comunicação por IPC Pipes, com o intuito de aplicar e consolidar os conhecimentos teóricos sobre sincronização e comunicação entre processos estudados na disciplina de Sistemas Operacionais, em um ambiente Linux.

2.2. Objetivos Específicos

- Projetar e implementar uma aplicação funcional que represente um cenário prático de tráfego aéreo, com subprocessos modelando torre de controle e aviões, e threads gerenciando a contagem de aeronaves;
- Utilizar IPC Pipes para realizar a troca de mensagens entre subprocessos, garantindo uma comunicação eficiente e sincronizada;
- Demonstrar o funcionamento do código em um ambiente Linux, utilizando o compilador GCC, conforme os padrões estabelecidos para a disciplina;
- Documentar detalhadamente a lógica do algoritmo, incluindo as etapas de desenvolvimento, estrutura do código e explicação das implementações;
- Disponibilizar o código-fonte em um repositório público no GitHub, acompanhado de link para direcionamento ao código.

3. DESCRIÇÃO DA APLICAÇÃO

3.1. Visão Geral

A aplicação desenvolvida simula um sistema de controle de tráfego aéreo, onde o processo principal atua como pai, a torre de controle atua como o processo filho e os aviões como subprocessos filhos. A função principal da torre é gerenciar o fluxo de pousos e decolagens, controlando quais aviões têm permissão para realizar essas ações com base na disponibilidade da pista.

O processo filho (torre) recebe solicitações dos subprocessos filhos (aviões), registrando informações como identificadores dos aviões, solicitações de pouso ou decolagem, e as decisões tomadas. Esses dados são armazenados em um arquivo CSV, que serve como um registro do tráfego aéreo.

Após esse entendimento, uma thread é utilizada para auxiliar a torre de controle no monitoramento do número de aeronaves que realizaram pousos e decolagens na pista. O funcionamento desse fluxo está ilustrado na imagem abaixo.

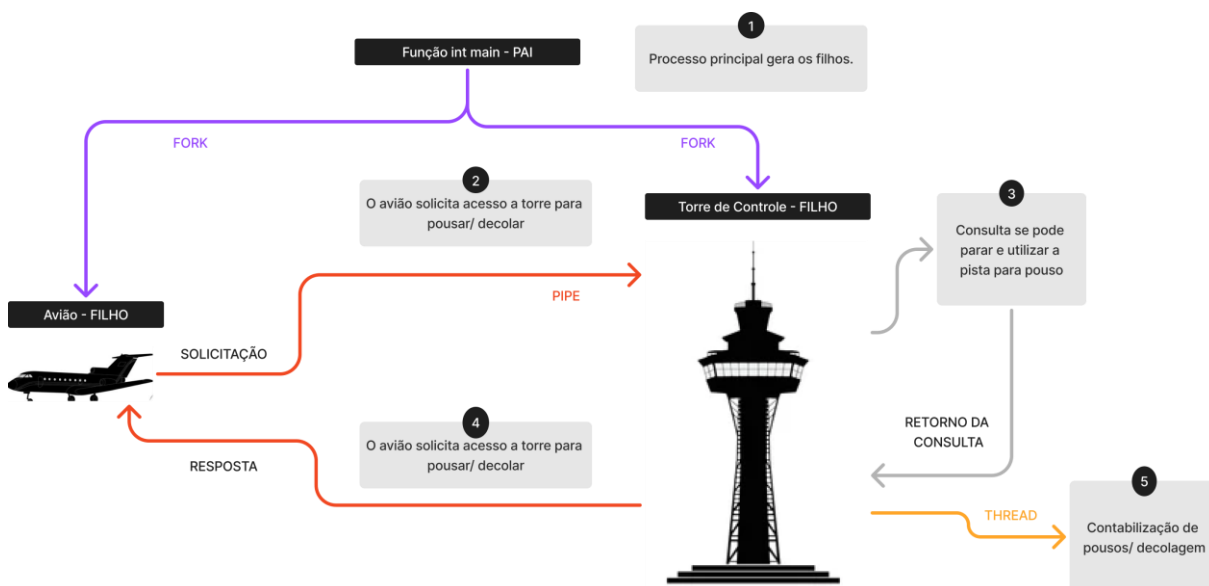


Figura 1: Diagrama de Lógica Inicial – Elaborado pelos autores

3.2. Estrutura Geral

A aplicação foi concebida para simular um sistema de tráfego aéreo. Abaixo estão detalhadas as principais entidades e funcionalidades que compõem a estrutura geral do sistema.

- a) Cabeçalhos e Inclusões: O programa utiliza bibliotecas padrão como `stdio.h`, `stdlib.h`, `string.h`, `pthread.h`, além de outras relacionadas a sinais, pipes e threads, garantindo a funcionalidade essencial do sistema.
- b) Definições de Constantes: Incluem o tamanho máximo de buffers, o número de aeronaves simuladas e os diferentes estados possíveis das aeronaves, como pousando, decolando ou aguardando permissão.
- c) Estruturas de Dados: As estruturas foram definidas para organizar e maximizar a quantidade de aeronaves simuladas, incluindo seus atributos e status.
- d) Variáveis Globais: Declaradas para serem compartilhadas entre processos e threads. Estas variáveis são protegidas por mutexes para evitar problemas de concorrência.
- e) Funções Definidas:
 - Função principal (`main`) para coordenar a execução do programa;
 - Função de requisição para gerenciar as solicitações das aeronaves;
 - Função de controle para implementar a lógica central da torre;
 - Função para inicializar a thread responsável pelo monitoramento;
 - Função de interrupção para lidar com entradas de teclado e encerrar o programa de forma controlada;
 - Função para iniciar e atualizar o contador de pousos e decolagens.
- f) Comunicação entre Subprocessos: Pipes são empregados para enviar e receber informações entre a torre de controle e os processos das aeronaves. O código inclui funções específicas para configurar e gerenciar essa comunicação.
- g) Thread de Monitoramento: Uma thread dedicada monitora e exibe, periodicamente, estatísticas sobre o número de pousos e decolagens realizados, funcionando de maneira paralela ao restante do sistema.
- h) Finalização e Limpeza: No encerramento do programa, todos os recursos são devidamente liberados:
 - Pipes são fechados;
 - Mutexes e threads são encerrados;

- O arquivo de registros é salvo e fechado para garantir a persistência dos dados coletados.

3.3. Fluxo Geral

A inicialização da aplicação ocorre com a criação do processo filho, representado pela torre de controle, que configura os IPC Pipes para permitir a comunicação entre os processos. Em seguida, subprocessos filhos, que representam os aviões, são gerados e inicializados, estabelecendo a comunicação com a torre de controle.

Durante a comunicação, os aviões enviam solicitações para a torre por meio dos IPC Pipes, solicitando permissões para pouso ou decolagem. A torre processa essas solicitações, verificando a disponibilidade da pista antes de responder aos aviões. Caso algum erro ocorra no fluxo de comunicação ou no processo, ele é registrado e exibido no terminal para identificação e correção.

No gerenciamento da pista, a torre de controle controla o uso sequencial da pista, garantindo que apenas um avião possa operar na pista de cada vez. Após a realização de cada operação, seja pouso ou decolagem, a pista é liberada para o próximo avião na fila, permitindo a execução do próximo processo de forma ordenada.

Por fim, todas as atividades realizadas, como permissões concedidas, pousos, decolagens e o status dos aviões, são registradas em um arquivo CSV. Esse arquivo funciona como um histórico, permitindo auditoria e análise posterior das operações realizadas pela torre de controle e pelos aviões no sistema.

3.4. Organização do código

A organização do código é observada a seguir:

1. Bibliotecas;
2. Definições de máximos;
3. Structs necessárias;
4. Assinatura das funções;
5. Variáveis globais;
6. Função principal;

7. Funções adjuntas;

3.5. Etapas de Desenvolvimento

As etapas de desenvolvimento do projeto foram estruturadas de maneira incremental, com o objetivo de construir a aplicação de forma gradual, validando cada funcionalidade conforme o progresso. Dessa forma, o escopo fora definido em 5 etapas:

- I. **Etapas:** Envolveu a análise de requisitos e o levantamento das funcionalidades essenciais para a aplicação. O foco inicial foi criar uma estrutura simples, composta por um processo filho (torre de controle) e um processo filho (avião), com o objetivo de testar a comunicação entre eles, utilizando os IPC Pipes.
- II. **Etapas:** A comunicação entre o processo filho e outro foi expandida para integrar a funcionalidade da pista. A torre de controle passou a gerenciar os eventos relacionados ao uso da pista, verificando a disponibilidade para pouso e decolagem dos aviões. A comunicação entre os processos foi ajustada para garantir que a torre poderia controlar adequadamente os pousos e decolagens, mantendo a ordem e a eficiência na utilização da pista.
- III. **Etapas:** Consistiu na introdução de múltiplos aviões para testar a aplicação com mais de um processo filho. Isso permitiu validar o funcionamento da aplicação em cenários com mais aviões, garantindo que a torre de controle pudesse gerenciar simultaneamente diversas solicitações de pouso e decolagem.
- IV. **Etapas:** Foram implementadas estampas de tempo em postos-chaves do código. A adição de tempos simulou um ambiente mais próximo da realidade e facilitou o acompanhamento das ações de pouso e decolagem.
- V. **Etapas:** Implementação da Thread para contabilização de pouso e decolagem.

Como referência desta seção, o repositório documentou e acompanhou todo o processo de desenvolvimento. Dessa forma, a imagem apresentada abaixo foi extraída diretamente do repositório, representando uma etapa específica do projeto e ilustrando a implementação realizada:



Figura 2: Demonstração Gráfica de Acompanhamento de todo o desenvolvimento do projeto.

4. DESCRIÇÃO DO CÓDIGO

4.1. Explicação Geral

4.1.1. Bibliotecas

O código foi desenvolvido com base na inclusão de bibliotecas fundamentais. A biblioteca `stdio.h` desempenhou um papel na exibição de mensagens no console e na manipulação de arquivos, enquanto a `string.h` foi para operações envolvendo textos, como a comparação de estados com base em strings. Para operações relacionadas à alocação dinâmica de memória e controle de processos, foi utilizada a biblioteca `stdlib.h`.

A comunicação entre subprocessos foi habilitada por meio da biblioteca `unistd.h`. Para implementar o suporte a threads, a inclusão da biblioteca `pthread.h` foi indispensável, permitindo a criação e sincronização de threads que operam em paralelo.

Além disso, a biblioteca `signal.h` foi utilizada para tratar sinais de interrupção, como o gerado pelo comando CTRL+C. Por fim, a biblioteca `sys/wait.h` foi incorporada para possibilitar que o programa aguardasse a finalização de subprocessos, garantindo que os recursos fossem liberados adequadamente antes do término da execução.

```
1. // BIBLIOTECAS
2.
3. #include <stdio.h>      // Biblioteca para entrada e saída de dados
4. #include <string.h>     // Biblioteca para manipulação de strings
5. #include <stdlib.h>     // Biblioteca para funções de alocação de memória
6. #include <unistd.h>     // Biblioteca para funções de sistema
7. #include <sys/wait.h>   // Biblioteca para funções de espera de processos
8. #include <time.h>       // Biblioteca para funções de tempo
9. #include <signal.h>     // Biblioteca para manipulação de sinais
10. #include <pthread.h>   // Biblioteca para manipulação de thread
```

Bloco de código 1: Inclusão de Bibliotecas

4.1.2. Definição de Máximos

Após a inclusão das bibliotecas essenciais, foi necessário definir as quantidades máximas para os diversos setores do aeroporto, levando em consideração a limitação de vagas e recursos disponíveis. No caso da pista de pouso e decolagem, foi estabelecido que a capacidade máxima seria de apenas 3 vagas, representada pela constante `MAX_AVIOES_AEROPORTO`, definida por meio da diretiva `#define`.

De maneira similar, foi necessário definir parâmetros temporais que simulam as operações de pouso e decolagem. Para isso, foram criadas duas constantes: `TEMPO_POUSO`, com valor de 5 minutos, e `TEMPO_DECOLAGEM`, com valor de 3 minutos. O trecho do código que implementa essas definições pode ser visualizado a seguir.

```
11. // DEFINIÇÃO DO NÚMERO MÁXIMO DOS PARAMETROS
12.
13. #define MAX_AVIOES_AEROPORTO 3    // Número máximo de aviões no aeroporto
14. #define TEMPO_POUSO 5             // Tempo que o avião fica parado no aeroporto
15. #define TEMPO_DECOLAGEM 3        // Tempo que o avião fica no ar
```

Bloco de código 2: Definição de números Máximos

- **Justificativa para a utilização de máximos:** Essa decisão foi tomada para garantir que o número de máximo de vagas na pista fosse fixo durante toda a execução do programa, assegurando consistência e previsibilidade. O uso de constantes facilita futuras modificações no código, permitindo modificar os parâmetros de forma mais fácil e modular.

4.1.3. Strucs – Estrutura de Dados

As estruturas de dados do tipo `struct` foram utilizadas para representar os estados dos aviões, armazenar dados gerados por eles e como estruturas auxiliares para armazenar contadores.

Primeiramente, foi criada uma enumeração chamada `Estado`, que define os quatro possíveis estados de um avião: voando, decolando, pousando e parado. Esse tipo de estrutura organiza os estados de forma clara e fácil de gerenciar. Por ser enumerado cada estado pode ser associado com um valor na utilização.

```

16. // Enum para definir os estados do avião
17.
18. typedef enum
19. {
20.     DECOLANDO, // Estado do avião ao decolar - 0
21.     POUSANDO,  // Estado do avião ao pousar - 1
22.     PARADO,    // Estado do avião parado - 2
23.     VOANDO,    // Estado do avião voando - 3
24. } Estado;

```

Bloco de código 3: Struct Estado

Em seguida, foi definida a estrutura Tráfego, que modela as informações do avião, incluindo seu ID, Latitude, Longitude, Altitude, Permissão e Estado (onde Estado refere-se à estrutura de dados previamente definida).

```

25. // Estrutura para definir os dados do avião
26.
27. typedef struct
28. {
29.     int id; // Identificador único do avião (usando PID)
30.     float latitude; // Posição latitude do avião (-90 a 90)
31.     float longitude; // Posição longitude do avião (-180 a 180)
32.     int altitude; // Altitude do avião em pés (1000 a 40000)
33.     char permissao[10]; // Status da permissão (Aprovado/Negado)
34.     Estado estado; // Estado atual do avião
    (DECOLANDO/POUSANDO/PARADO/VOANDO)
35. } Trafego;

```

Bloco de código 4: Struct Trafego

Por fim, foi criada a estrutura ThreadsArg, destinada a armazenar os contadores utilizados pela threads.


```

36. //Estrutura para contabilização
37.
38. typedef struct
39. {
40. int contador_pousos; // Contador para número total de pousos
41. int contador_decolagens; // Contador para número total de decolagens
42.
43. } ThreadArgs;

```

Bloco de código 5: Struct ThreadArgs

- **Justificativa para a utilização das structs:** Esse tipo de estrutura foi adotado para definir cadeias de informações padronizadas, possibilitando que essas estruturas fossem reutilizadas de maneira consistente em várias partes do código.

4.1.4. Assinaturas das funções

Para garantir a execução consistente das funções, foi necessário declarar as assinaturas das funções de forma adequada, visto que a linguagem C segue o modelo top-down. As funções definidas foram todas do tipo void, uma vez que nenhuma delas precisava retornar valor. As funções declaradas são as seguintes:

- **void aviao (int readfd, int writefd):** Esta função é responsável por simular o comportamento de um avião, gerar dados aleatórios para o descritivo da aeronave, enviar solicitações de pouso/decolagem para a torre através do pipe, receber respostas da torre e realizar ações. Essa função é descrita com mais detalhes na sessão [4.1.7. Função void aviao.](#)
- **void requisicao(int readfd, int writefd):** A função requisição simula o comportamento da torre de controle, recebe solicitações dos aviões, verifica se há vagas no aeroporto, autorizando ou negando o pouso, envia respostas para o avião, grava informações sobre as aeronaves em um arquivo de extensão .csv.e utiliza mutex para proteger o acesso aos contadores de pouso e decolagem. Essa função é descrita com mais detalhes na sessão [4.1.8 Função de Requisição.](#)

- **void handle_sigint(int sig):** A função `handle_sigint` é usada para interromper a execução do programa a qualquer momento, permitindo que o usuário finalize o processo com a combinação de teclas Ctrl + C. Esta assinatura pode ser visualizada em mais detalhes na sessão [4.1.10. Função handle_sigint.](#)
- **void thread_funcao(void *args):** Esta função define o comportamento de uma thread, sendo utilizada especificamente para o monitoramento das estatísticas do aeroporto, imprimindo periodicamente o número de decolagens e pousos. Além de utilizar mutex para sincronização de acesso aos contadores. O seu detalhamento pode ser observado na sessão [4.1.9 Função thread function.](#)
- **void iniciar_contador():** A função `iniciar_contador` é responsável pela inicialização da thread de monitoramento para a operação da aplicação. Sua descrição pode ser encontrada na sessão [4.1.11 Função de inicialização de contador.](#)

```

44. //Assinaturas das funções
45.
46. void controle(int readfd, int writefd); // Função que controla o
    tráfego dos aviões
47. void requisicao(int readfd, int writefd); // Função que processa as
    requisições dos aviões
48. void handle_sigint(int sig);           // interrupção do usuário
49. void thread_funcao(void *args); // Função da thread
50. void iniciar_contador(); // Função para iniciar a thread de
    monitoramento

```

Bloco de código 6: Assinatura de Funções

4.1.5. Variáveis Globais e inicialização

As variáveis globais foram utilizadas no código devido à necessidade de acesso e manipulação de dados em diferentes escopos, pois, se declaradas localmente, elas teriam o ciclo de vida pautado apenas na execução de cada escopo.

- **FILE *arquivos;:** Esta variável foi definida globalmente porque é utilizada em várias partes do programa para manipular arquivos, como na função `requisicao`. Sendo global, a variável `arquivos` é aberta no início da execução e fechada apenas no término do programa.

- **pthread_t monitor thread;** Esta variável representa a thread responsável por monitorar os eventos do sistema. Torná-la global facilita a criação e a junção da thread em diferentes pontos do programa, pois é necessário acessar o identificador da thread em diferentes partes do código, como na função `handle_sigint` para encerramento.
- **pthread_mutex_t contador_mutex;** O mutex `contador_mutex` é utilizado para garantir a sincronização do acesso a essas variáveis. Protege o acesso concorrente às variáveis `total_pausos` e `total_decolagens`.
- **int total_pausos, total_decolagens;** Essas variáveis são contadores auxiliares que mantêm o total de pausos e decolagens. Sua definição como variáveis globais permite que o valor total seja atualizado de forma simples e consistente, sem a necessidade de passá-las por referência entre várias funções. Elas podem ser incrementadas diretamente, independentemente do escopo onde são acessadas.
- **volatile e sig_atomic_t¹;** Indica se o programa deve ser encerrado. É acessada pela função `handle_sigint` para sinalizar que o programa deve terminar e pela função `thread_funcao` para verificar se deve continuar a execução.
 - **volatile:** Garante que o compilador não otimize a leitura dessa variável, pois seu valor pode ser alterado por um sinal externo (SIGINT).
 - **sig_atomic_t:** É um tipo de dado garantido como sendo atômico em operações de leitura e escrita, mesmo em ambientes multithread.

¹ KING, K. N. *C Programming: A Modern Approach*. 2. ed. Boston: W. W. Norton & Company, 2008.

- Capítulo 6: *Pointers and Dynamic Memory Allocation*
- Capítulo 9: *Signals and Interruption Handling*
- Capítulo 11: *The C Preprocessor*

```

51. // Variáveis globais
52.
53. FILE *arquivo = NULL; // Ponteiro para o arquivo
54. pthread_t monitor_thread; // identificador da thread de monitoramento
55. pthread_mutex_t contador_mutex = PTHREAD_MUTEX_INITIALIZER; // Mutex
    para proteger contadores
56. int total_pousos = 0; // Contador global de pousos
57. int total_decolagens = 0; // Contador global de decolagens
58. volatile sig_atomic_t should_terminate = 0; // Flag para indicar
    término do programa

```

Bloco de código 7: Declarações de variáveis globais

4.1.6. Função *int main()*

A função ***int main()*** é o ponto de entrada principal do programa, centralizando todos os componentes essenciais para sua execução do programa.

4.1.6.1. Declaração de variáveis locais e inicialização

A primeira parte do código da função ***int main()*** é dedicada à declaração de variáveis. Inicialmente, é declarada uma variável inteira chamada ***torre***, que armazenará o identificador do processo filho gerado pela chamada ***fork()*** (representando a "Torre de Controle").

Em seguida, são declarados dois vetores com dois elementos cada, correspondentes aos pipes. Esses pipes são responsáveis pela comunicação: um para as requisições e outro para o controle. Cada pipe é composto por duas extremidades, indicadas pelos índices 0 e 1, permitindo a comunicação bidirecional entre os subprocessos (avião e torre).

A existência de dois pipes é necessária devido ao fluxo cruzado de informações entre a torre e os aviões. Essa utilização garante maior segurança e organização na comunicação, evitando conflitos entre leitura e escrita simultânea, uma vez que cada pipe é dedicado a uma direção específica do fluxo de dados.

Após a declaração das variáveis, a função **srand** é utilizada para inicializar o gerador de números aleatórios, configurando-o com o tempo atual como semente.

Em seguida, a instrução **signal (SIGINT, handle_sigint)** configura o programa para capturar interrupções manuais, como o comando Ctrl + C. Quando essa interrupção ocorre, a função **handle_sigint** é chamada para tratar a situação adequadamente.

```
59. // Variáveis Locais
60.
61. int aviao; // Receber o código do fork
62. int piperequisicao[2]; // Requisição dos aviões → torre
63. int pipecontrole[2]; // Processamento de torre → avião
64. srand(time(NULL)); // Gera números aleatórios com base no
    tempo
65. signal(SIGINT, handle_sigint); // Gerenciamento de interrupções (CTRL
    + C)
```

Bloco de código 8: Variáveis Locais

4.1.6.2. Pipe

A criação dos pipes, pipe(piperequisicao) e pipe(pipecontrole), é fundamental para a comunicação bidirecional entre os aviões e a torre de controle. O primeiro pipe (piperequisicao) é utilizado para as mensagens enviadas pelos aviões para a torre, enquanto o segundo pipe (pipecontrole) gerencia as mensagens enviadas da torre para os aviões. Caso a criação de qualquer um dos pipes falhe (retornando um valor menor que 0), o programa detecta o erro, exibe uma mensagem apropriada para o usuário e encerra a execução imediatamente, garantindo que falhas críticas não afetem o funcionamento geral do sistema.

```
66. // Se der erro nos pipes
67.
68. if (pipe(piperequisicao) < 0 || pipe(pipecontrole) < 0)
69. {
70.     printf("A pista encontra-se interditada (ERRO AO CRIAR PIPES)");
71.     exit(1);
72. }
```

Bloco de código 9: Tratamento de erro caso a inicialização do Pipe apresente erro

4.1.6.3. Sub processos

A primeira etapa crucial na criação de subprocessos é a tratativa de erros durante a chamada ao `fork()`:

- **Processo filho torre:** Recebe um valor positivo correspondente ao PID
- **Processo filho aviões:** Recebe um valor igual a 0, indicando que foi criado com sucesso.
- **Falha no fork:** Caso o `fork()` retorne um valor negativo, significa que a criação do subprocesso falhou.

Neste caso, o código armazena o retorno do `fork()` na variável `torre`. Se o valor for negativo, o programa imprime uma mensagem de erro e finaliza a execução para evitar problemas futuros.

```
73. // Se o processo der erro na criação
74.
75. if ((torre = fork()) < 0)
76.     {
77.         printf("Erro da chamada de fork (processo não criado)");
78.         exit(1);
79.     }
```

Bloco de código 10: Criação do processo filho torre

4.1.6.4. Processo Filho – Torre

Se o `fork` for criado de maneira efetiva o código vai testar se ele é o processo filho, ou seja, se a `torre` for igual a 0 ela pode ser configurada como um processo filho assim ela invoca uma [função de inicialização do contador](#). Para otimizar a comunicação e evitar possíveis conflitos, o processo fecha as extremidades que não serão usadas, ressaltando que `pipe` requisicao serve como um canal para que os processos que representam os aviões enviem suas requisições (pouso, decolagem) para a torre de controle e o `pipe` controle é utilizado pela torre de controle para enviar suas respostas (autorização ou negação) aos aviões. Dessa maneira, nessa parte do código:

- **close(piperequisicao[1])**: Fecha o lado de escrita do pipe de requisição;
- **close(pipecontrole[0])**: Fecha o lado de leitura do pipe de controle.

A função `requisicao (piperequisicao[0], pipecontrole[1])` é chamada. Ela é necessária para processar as requisições dos aviões. Os parâmetros são os descritores de leitura do `piperequisicao` e de escrita do `pipecontrole`. Após completar a execução as extremidades restantes dos pipes são fechadas para liberar recursos:

- **close(piperequisicao[0])**: Fecha o lado de leitura do pipe de requisição;
- **close(pipecontrole[1])**: Fecha o lado de escrita do pipe de controle.

O subprocesso utiliza o comando `while (wait(NULL) > 0)`, um loop que aguarda o término de todos os processos filhos. Essa função retorna o PID do processo filho terminado ou -1 se não houver mais processos filhos. O loop continua enquanto houver processos filhos a serem esperados.

```

80. if (torre == 0)
81.     { // Processo filho
82.
83.         iniciar_contador();
84.
85.         // Torre fecha as extremidades que não vai usar
86.         close(piperequisicao[1]);
87.         close(pipecontrole[0]);
88.
89.         // Torre executa a requisição
90.         requisicao(piperequisicao[0], pipecontrole[1]);
91.
92.         // Fecha os descritores após uso
93.         close(piperequisicao[0]);
94.         close(pipecontrole[1]);
95.
96. // Espera todos os processos acabarem
97.     while (wait(NULL) > 0);
98. }
```

Bloco de código 11: Processo da criação da torre

4.1.6.5. Processos Filhos - Aviões

Essa parte do código define o comportamento do processo pai, que é responsável por criar os processos que representam os aviões. Lembrando que nessa parte o processo pai será a função main, dessa forma o processo pai fecha as extremidades dos pipes que não serão utilizadas por ele (escrita no `piperequisicao` e leitura no `pipecontrole`), uma vez que ele apenas criará os processos dos aviões. Assim:

- **`close(piperequisicao[0])`**: Fecha o lado de leitura do pipe de requisição;
- **`close(pipecontrole[1])`**: Fecha o lado de escrita do pipe de controle.

O processo pai entra em um laço **for**, onde cria múltiplos processos filhos. A quantidade de processos filhos é determinada pela constante **MAX_AVIOES_AEROPORTO**, multiplicada por 2. Dentro do laço, o pai chama `fork()` para criar subprocessos.

O processo filho entra no corpo do `if` e executa a função `aviao`, que simula o comportamento de um avião com a [função avião](#) (`pipecontrole[0]`, `piperequisicao[1]`), após isso Os descritores são fechados dentro do processo filho para evitar vazamentos de recursos. Dessa maneira:

- **`close(pipecontrole[0])`**: Fecha o lado de leitura do pipe de controle.
- **`close(piperequisicao[1])`**: Fecha o lado de escrita do pipe de requisição.

O processo filho é então encerrado usando `exit(0)`.

Após a criação de todos os processos filhos, o processo pai fecha novamente os descritores dos pipes, por segurança e entra um loop utilizando `wait(NULL)` para aguardar o término de todos os processos filhos. Isso garante que o processo pai não termine antes que todos os aviões tenham finalizado suas simulações.


```

99.  else
100. { // Processo filho: Criador dos aviões
101.  // Fecha as extremidades que não vai usar
102.  close(piperequisicao[0]);
103.  close(pipecontrole[1]);
104.
105.  // Cria os processos dos aviões
106.  for (int i = 0; i < MAX_AVIOES_AEROPORTO * 2; i++)
107.  {
108.      if (fork() == 0)
109.      {
110.  // Processo avião
111.      aviao(pipecontrole[0], piperequisicao[1]); // Executa a função
112.      aviao
113.      close(pipecontrole[0]); // Fecha o descritor
114.      close(piperequisicao[1]); // Fecha o descritor
115.      exit(0); // Encerra o processo
116.      }
117.      }
118.  // Fecha os descritores após criar todos os aviões
119.  close(pipecontrole[0]); // Fecha o descritor
120.  close(piperequisicao[1]); // Fecha o descritor
121.  // Aguarda todos os processos filhos terminarem
122.  while (wait(NULL) > 0);
123.  }

```

Bloco de código 12: Criação de Processos filhos

4.1.6.6. Retorno

Após a execução de todos os processos e finalização dos filhos, o processo principal retorna 0, indicando a execução bem-sucedida.

```

124.      return 0;

```

Bloco de código 13: Retorno da Função int main

4.1.7. Função Avião

4.1.7.1. Cabeçalho da função

A função `aviao` é definida com o tipo `void`, indicando que ela não retorna nenhum valor. O nome da função reflete sua responsabilidade de simular o comportamento de um avião, que faz solicitações de pouso ou decolagem.

A função recebe dois parâmetros: `int readfd` e `int writefd`, que representam os descritores de arquivo para leitura e escrita em pipes. Assim, ela envia uma mensagem para a torre de controle solicitando pouso ou decolagem, dependendo do estado atual do avião e aguarda a resposta da torre de controle (autorização ou negação).

```
125. void aviao(int readfd, int writefd)
126. {
```

Bloco de código 14: Assinatura da Função avião

4.1.7.2. Geração de dados

A função `srand(getpid())` inicializa o gerador de números aleatórios usando o identificador do processo atual (PID). Desse modo, garante que cada avião (subprocesso) tenha uma sequência única de números aleatórios.

A struct **Trafego** é usada para armazenar informações de tráfego aéreo, representada pela variável declarada juntamente com a instância de estrutura **aviao**. A inicialização dos dados ocorre de forma randômica:

- **ID**: Definido como o PID do processo atual, garantindo unicidade.
- **Latitude e Longitude**: Geradas aleatoriamente dentro dos intervalos geográficos permitidos (latitude de -90° a +90° e longitude de -180° a +180°).
 - $((float)rand() / RAND_MAX) * 180.0$ gera um número aleatório entre 0 e 180, que é então adicionado à latitude mínima. Isso garante que a latitude gerada esteja entre -90 e 90 graus, totalizando os 180 graus.
 - $((float)rand() / RAND_MAX) * 360.0$ gera um número aleatório entre 0 e 360, que é então adicionado à longitude mínima. Isso garante que a longitude gerada esteja entre -180 e 180 graus, cobrindo todo o globo terrestre.

- **Altitude:** Um valor aleatório entre 1.000 e 40.000 metros, considerando limites de voo comerciais.
 - **rand() % 39001** gera um número aleatório entre 0 e 39000, que é então adicionado à altitude mínima. Isso resulta em altitudes entre 1000 e 40000 metros.
- **Permissão de Voo:** Inicialmente definida como "negada", o que significa que o avião não pode pousar ou decolar no início.
- **Estado Inicial:** Configurado como "voando".

```

127.srand(getpid()); // Gera números aleatórios com base no PID
128.
129. // Função para manipular o tráfego do avião
130. Trafego aviao;
131.
132. // Preenche os dados de forma aleatória
133. aviao.id = getpid() // PID como ID único
134.// Gera latitude aleatória entre -90.0 e 90.0
135. aviao.latitude = -90.0 + ((float)rand() / RAND_MAX) * 180.0;
136.// Gera longitude aleatória entre -180.0 e 180.0
137. aviao.longitude = -180.0 + ((float)rand() / RAND_MAX) * 360.0;
138.// Gera altitude aleatória entre 1000 e 40000
139. aviao.altitude = 1000 + rand() % 39001;
140.// Seta permissão (negado como default)
141. snprintf(aviao.permissao, sizeof(aviao.permissao), "Negado");
142.;// Estado do avião (VOANDO como default)
143. aviao.estado = VOANDO

```

Bloco de código 15: Preenchimento inicial de parâmetros da aeronave

4.1.7.3. While – Laço de repetição

O loop *while* é utilizado para simular um processo contínuo, onde o avião solicita repetidamente permissão para pouso ou decolagem e aguarda a resposta da torre de controle. O processo continua indefinidamente até ser encerrado manualmente ou por algum evento.

Inicialmente, o avião está "voando" e sua permissão é "negada". O envio dos dados do avião para a torre de controle é feito através de *pipes*, com o código *write*

(writefd, &avião, sizeof(Trafego)). Se o envio falhar, um erro é reportado e o programa é encerrado com `exit(1)`..

```
195.While(1)
196.{
197. // Se ele solicitou permissão para pouso, ele envia os dados para
    a torre
198. if (write(writefd, &aviao, sizeof(Trafego)) < 0)
199. {
200.     perror("Erro ao enviar dados para a torre");
201.     exit(1);
202. }
```

Bloco de código 16: Início da Solicitação

4.1.7.3.1. Solicitações

O avião solicita permissão de pouso ou decolagem dependendo do seu estado. Inicialmente, o estado é "voando", portanto, ele solicita uma decolagem, mas como a permissão começa como "negada", o pedido será rejeitado.

A solicitação é enviada para a torre de controle, e a função `read` aguarda a resposta da torre, armazenando-a na variável `resposta`. Se a leitura falhar, o programa é encerrado **read (readfd, &resposta, sizeof(Trafego))**. A permissão recebida (aprovada ou negada) é impressa no console.

```
203. // Mensagem de solicitação do avião (pouso ou decolagem)
204. printf("Avião %d solicitando %s\n", aviao.id,
205. aviao.estado == VOANDO ? "pouso" : "decolagem");
206.
207. // Se ele vai pousar/decolar, primeiro tem que aguardar resposta
    da torre
208. Trafego resposta;
209. if (read(readfd, &resposta, sizeof(Trafego)) < 0)
210. {
211.     perror("Erro ao receber resposta da torre");
212.     exit(1);
213. }

214. // Mensagem de resposta (Aprovado ou negado)
215. printf("Avião %d recebeu resposta: %s\n", aviao.id,
    resposta.permissao);
```

Bloco de código 17: Resposta da requisição

4.1.7.3.2. Procedimento Aprovado

Caso a permissão seja aprovada e o avião esteja no estado "parado", ele pode iniciar a decolagem. O sistema imprime uma mensagem informando que o avião está decolando. O avião então realiza a decolagem por um tempo determinado aleatoriamente (de 0 a 19 segundos).

Após a decolagem, o estado do avião muda para "voando". O avião então voa por um tempo aleatório entre 0 e 19 segundos, o que simula o tempo de voo.

Se o procedimento for aprovado e o avião estiver no estado "voando", ele precisará realizar um pouso.

```
216.// Procedimento caso aprovado
217.if (strcmp(resposta.permissao, "Aprovado") == 0)
218.{
219.    // Procedimento caso aprovado e estiver parado
220.    if (aviao.estado == PARADO)
221.{
222.printf("Avião %d iniciando decolagem...\n\n", aviao.id);
223.        sleep(TEMPO_DECOLAGEM);
224.        aviao.estado = VOANDO;
225.
226.        int tempo_voando = rand() % 10 + 10;
227.        printf("Avião %d ficará voando por %d segundos\n\n",
                aviao.id, tempo_voando);
228.        sleep(tempo_voando);
229. // Procedimento caso aprovado e estiver voando
230. }
231.
232.else if (aviao.estado == VOANDO)
233.{
234.    printf("Avião %d iniciando pouso...\n\n", aviao.id);
235.    sleep(TEMPO_POUSO);
236.    aviao.estado = PARADO;
237.
238.    int tempo_parado = rand() % 10 + 10;
239.    printf("Avião %d ficará parado por %d segundos\n\n", aviao.id,
            tempo_parado);
240.    sleep(tempo_parado);
241. }
242. }
```

Bloco de código 18: Procedimento caso tenha sido aprovado

4.1.7.3.3. Procedimento Negado

Se o procedimento for negado, o avião tomará ações diferentes dependendo de seu estado. Se o avião estiver no ar e a permissão for negada, ele entrará em uma fase de espera antes de solicitar novamente o pouso.

Caso o avião esteja no solo e a decolagem seja negada, ele também aguardará antes de fazer uma nova solicitação. Isso representa a necessidade de aguardar uma vaga na pista para iniciar a decolagem.

Para simular essa espera, o código introduz um atraso de 5 segundos utilizando a função ***sleep***. Durante esse período, o avião não realiza nenhuma ação, aguardando passivamente por uma nova oportunidade. Esse atraso serve como uma representação simplificada da análise de espaço aéreo realizada pela torre de controle, que avalia a disponibilidade de pistas e a presença de outros aviões antes de autorizar um pouso ou decolagem.

```
243. // Procedimento caso negado
244. else
245. {
246.     // Procedimento caso negado e estiver voando
247.     if (aviao.estado == VOANDO)
248.     {
249.         printf("Avião %d aguardando novo pedido para
           pouso...\n\n", aviao.id);
250.     }
251.     // Procedimento caso negado e estiver parado
252.     else if (aviao.estado == PARADO)
253.     {
254.         printf("Avião %d aguardando novo pedido para
           decolagem...\n\n", aviao.id);
255.     }
256.     sleep(5);
257. }
258. }
259. }
```

Bloco de código 19: Procedimento caso tenha sido negado

4.1.8. Função Requisição

4.1.8.1. Cabeçalho da Função

A função `requisicao` simula o comportamento da torre de controle, recebendo solicitações de pouso e decolagem dos aviões, processando essas solicitações e enviando as respostas correspondentes, controlando as solicitações de pousos e decolagens com base na situação do aeroporto (se está lotado ou não). A função é declarada como `void`, o que significa que ela não retorna nenhum valor. Os parâmetros **`int readfd`** e **`int writefd`** representam os descritores de arquivos para leitura e escrita em *pipes*, usados para comunicação com o "avião".

```
187.     void requisicao(int readfd, int writefd)
```

Bloco de código 20: Assinatura da função

4.1.8.2. Abrindo um Arquivo

Além da comunicação entre os processos pai e filho, o código também lida com a criação de um arquivo, no qual serão armazenadas informações sobre os aviões que solicitam pouso ou decolagem. O arquivo é aberto com a extensão `.csv` para manter a persistência dos dados, como latitude, longitude, altitude e permissão. Caso o arquivo não seja criado corretamente (por erro), o programa será encerrado. O cabeçalho do arquivo é preenchido com as colunas necessárias, incluindo ID do avião, latitude, longitude, altitude e permissão.

```
260.     arquivo = fopen("avioes_pipe.csv", "w");
261.         if (arquivo == NULL)
262.         {
263.             perror("Erro ao criar o arquivo");
264.             exit(1);
265.         }
266.         // Cabeçalho do arquivo
267.         fprintf(arquivo, "ID, Latitude, Longitude, Altitude, Permissao\n"
);
```

Bloco de código 21: Arquivo `avioes_pipe.csv`

4.1.8.3. Inicialização

Nesta parte do código, são inicializadas as variáveis necessárias, como a struct **Trafego** que contém as informações dos aviões. A variável **size_t bits_lidos** armazena a quantidade de bits lidos do pipe, e a variável **avioes_no_aeroporto** conta o número de aviões estacionados no aeroporto, começando com 0.

```
188. Trafego aviao;
189. // Quantidade de bytes lido no read abaixo (para determinar
    erros)
190.
191. size_t bytesRead;
192.
193. // Contador de aviões no aeroporto
194. int avioes_no_aeroporto = 0;
```

Bloco de código 22: Inicialização

4.1.8.4. While

Um loop infinito é iniciado para ler as solicitações dos aviões utilizando o descritor de leitura do pipe (readfd). O loop continua executando enquanto houver dados no pipe. Para cada leitura bem-sucedida, os dados da estrutura aviao são preenchidos e uma mensagem informando o tipo de solicitação (pouso ou decolagem) é exibida. O loop só será interrompido quando a leitura retornar um valor menor ou igual a zero, indicando que não há mais dados para processar.

```
268. while ((bytesRead = read(readfd, &aviao, sizeof(Trafego))) > 0)
269. {
270. printf("Torre recebeu solicitação de %s do avião %d\n",
    aviao.estado == VOANDO ? "pouso" : "decolagem", aviao.id);
```

Bloco de código 23: Enviando uma solicitação

4.1.8.4.1. Processamento de Solicitações de Pouso

Se o avião estiver solicitando pouso, o código verifica se o aeroporto não está lotado. Se o número de aviões no aeroporto for menor que o máximo permitido, a solicitação de pouso será autorizada, e o contador de aviões no aeroporto será incrementado. Uma mensagem de sucesso será exibida. Caso contrário, se o aeroporto já estiver cheio, a solicitação será negada, e uma mensagem de erro será exibida, indicando que o aeroporto está lotado.

```
271. // Procedimento caso a torre receber uma mensagem de um avião
    voando
272. if (aviao.estado == VOANDO)
273. {
274.     // Procedimento caso o aeroporto não estiver lotado
275.     if (avioes_no_aeroporto < MAX_AVIOES_AEROPORTO)
276.     {
277.         snprintf(aviao.permissao, sizeof(aviao.permissao),
            "Aprovado");
278.         avioes_no_aeroporto++;
279.         printf("Torre: Pouso autorizado para avião %d. Aviões no
            aeroporto: %d\n",
280.             aviao.id, avioes_no_aeroporto);
281.     }
282.
283. // Procedimento caso o aeroporto estiver lotado
284. else
285. {
286.     snprintf(aviao.permissao, sizeof(aviao.permissao), "Negado");
287.     printf("Torre: Pouso negado para avião %d. Aeroporto lotado.
            Aviões no aeroporto: %d\n",
288.         aviao.id, avioes_no_aeroporto);
289. }
290. }
```

Bloco de código 24: Procedimento de processamento de pouso

4.1.8.4.2. Processamento de Solicitações de Decolagem

Para a solicitação de decolagem, o código verifica se o avião está parado no aeroporto. Se estiver, a permissão para decolagem será concedida, o número de aviões no aeroporto será decrementado, e uma mensagem de sucesso será exibida.

Caso contrário, a solicitação de decolagem será negada, e uma mensagem de erro será exibida.

```
291. // Procedimento caso a torre receber uma mensagem de um avião parado
292.
293. else if (aviao.estado == PARADO)
294. {
295.     snprintf(aviao.permissao, sizeof(aviao.permissao), "Aprovado");
296.     avioes_no_aeroporto--;
297.     printf("Torre: Decolagem autorizada para avião %d. Aviões no
        aeroporto: %d\n",
298.         aviao.id, avioes_no_aeroporto);
299. }
```

Bloco de código 25: Procedimento de processamento de decolagem

4.1.8.4.3. Escrita no Arquivo

Após o processamento das solicitações (seja de pouso ou decolagem), os dados do avião (ID, latitude, longitude, altitude e permissão) são registrados no arquivo **.csv**. A função **fprintf** é usada para escrever essas informações no arquivo.

```
300. // Escreve os dados no arquivo
301. fprintf(arquivo, "%d,%.6f,%.6f,%d,%s\n",
302.         aviao.id, aviao.latitude, aviao.longitude,
            aviao.altitude, aviao.permissao);
```

Bloco de código 26: Escrita em um arquivo

4.1.8.4.4. Resposta

Após registrar as informações no arquivo, a torre de controle envia a resposta de volta ao avião através do pipe, informando se a solicitação foi aprovada ou negada. A estrutura **aviao** é escrita no pipe, e se ocorrer um erro na escrita, o programa encerra com um código de erro. Caso contrário, uma mensagem é exibida no monitor, indicando que a pista foi liberada para o avião.

```

303. // Envia resposta para o avião
304.     if (write(writefd, &aviao, sizeof(Trafego)) < 0)
305.     {
306.         perror("Erro ao enviar resposta para o avião");
307.         fclose(arquivo);
308.         exit(1);
309.     }
310.
311.     printf("Torre enviou resposta ao avião %d: %s\n",
        aviao.id, aviao.permissao);
312.
313.     // Libera a pista
314.     if (strcmp(aviao.permissao, "Aprovado") == 0)
315.     {
316.         printf("Torre: Pista liberada\n\n");
317.     }

```

Bloco de código 27: Resposta

4.1.8.4.5. Contagem

Quando uma solicitação de pouso ou decolagem é aprovada, o contador correspondente (pousos ou decolagens) é incrementado de forma segura, utilizando uma mutex. Isso garante que apenas uma thread possa acessar e modificar esses contadores por vez. O contador de pousos ou decolagens é incrementado, dependendo da operação realizada. Após a atualização do contador, o mutex é liberado, permitindo que outra thread acesse a região crítica.

Além disso, um pequeno atraso (sleep) é inserido para simular a espera entre a análise de uma solicitação e o próximo pedido. Caso ocorra algum erro durante a leitura do pipe, o programa trata a falha e exibe uma mensagem de erro, encerrando o processo de forma controlada.

```

318. if (aviao.estado == VOANDO && strcmp(aviao.permissao,
    "Aprovado") == 0)
319.     {
320.         pthread_mutex_lock(&contador_mutex);
321.         total_pousos++;
322.         pthread_mutex_unlock(&contador_mutex);
323.     }
324.     else if (aviao.estado == PARADO &&
    strcmp(aviao.permissao, "Aprovado") == 0)
325.     {
326.         pthread_mutex_lock(&contador_mutex);
327.         total_decolagens++;
328.         pthread_mutex_unlock(&contador_mutex);
329.     }
330.
331.     // Aguarda 3 segundos antes de processar a próxima
    solicitação
332.     sleep(4);
333. }
334.
335. // Verificação de erro
336. if (bytesRead < 0)
337. {
338.     perror("Erro ao ler os dados do pipe");
339. }
340. }

```

Bloco de código 28: Contabilizador de pousos e decolagens

4.1.9. Função `thread_function`

4.1.9.1. Cabeçalho da função

O objetivo dessa função é ser executada por uma thread, e ela recebe um argumento `args` do tipo `void *`. Isso permite que a função possa manipular qualquer tipo de dado, não se limitando a tipos como `int`, `char`, entre outros. O tipo `void *` funciona como um ponteiro genérico, permitindo que qualquer tipo de dado seja processado pela thread.

Ao receber o argumento `args`, a função converte esse ponteiro para o tipo específico que é esperado.

```

184. void thread_function(void *args)
185. {
186. ThreadArgs *threadArgs = (ThreadArgs *)args; // Argumentos da
    thread

```

Bloco de código 29: Assinatura da função

4.1.9.2. While com condição de terminação

A thread entra em um loop infinito que continua até que a variável `should_terminate` seja definida como verdadeira, indicando que a thread deve ser encerrada. Antes de acessar e imprimir os valores dos contadores, ela adquire o **mutex contador_mutex**. **threadArgs->contador_pousos = total_pousos** que copia o valor do contador de pousos global para a estrutura local da thread. **ThreadArgs->contador_decolagens = total_decolagens**, copia o valor do contador de decolagens global para a estrutura local da thread.

```

165.     while (!should_terminate)
166.     {
167.         pthread_mutex_lock(&contador_mutex); // Bloqueia o
            acesso ao contador
168.
169.         // Imprime estatísticas do aeroporto
170.         threadArgs->contador_pousos = total_pousos;
171.         threadArgs->contador_decolagens = total_decolagens;
172.
173.         printf("\n=== Estatísticas do Aeroporto ===\n");
174.         printf("Total de pousos: %d\n", threadArgs-
            >contador_pousos);
175.         printf("Total de decolagens: %d\n", threadArgs-
            >contador_decolagens);
176.         printf("=====\n\n");
177.
178.         pthread_mutex_unlock(&contador_mutex); //
            Desbloqueia o acesso ao contador
179.         sleep(10); // Aguarda 10 segundos antes de imprimir
            as estatísticas novamente
180.     }
181.     free(args);
182.     //return NULL; // Retorna NULL

```

Bloco de código 30: Monitoramento das estatísticas do aeroporto

Após imprimir as estatísticas, a thread libera o mutex, permitindo que outras threads acessem os contadores. A thread pausa por 10 segundos antes de repetir o loop, após isso libera a memória alocada para a estrutura **ThreadArgs**.

4.1.10. Função *handle_sigint*

A função `handle_sigint` é chamada quando o programa recebe uma interrupção de sinal, como quando o usuário pressiona CTRL+C no terminal. Isso ocorre devido à captura do sinal SIGINT. O manipulador de sinal é configurado para alterar a variável `should_terminate` para 1, o que indica que o programa deve ser encerrado.

Se o arquivo estiver aberto (`arquivo != NULL`), a função grava as estatísticas finais de pousos e decolagens no arquivo `avioes_pipe.csv`, adicionando uma linha com essas informações. Em seguida, o arquivo é fechado, e uma mensagem de sucesso é exibida para indicar que a gravação foi concluída corretamente.

A função **`pthread_join (monitor_thread, NULL)`** aguarda a thread de monitoramento terminar sua execução. Isso é essencial para garantir que o programa só seja encerrado após a finalização correta da thread de monitoramento.

O mutex `contador_mutex` é destruído com `pthread_mutex_destroy (&contador_mutex)`, liberando qualquer recurso associado à proteção das variáveis compartilhadas, como os contadores de pousos e decolagens. A função `exit (0)` encerra o programa de forma limpa, retornando o código de saída 0, o que indica que não houve erro durante a execução.

```

341. // Handle no caso de interrupção (CTRL + C)
342. void handle_sigint(int sig)
343. {
344.     should_terminate = 1;
345.
346.     if (arquivo != NULL)
347.     {
348.         // Adiciona uma linha em branco para separar os dados das
estatísticas
349.         fprintf(arquivo, "\n=== Estatísticas Finais ===\n");
350.         fprintf(arquivo, "Total de pousos realizados,%d\n",
total_pousos);
351.         fprintf(arquivo, "Total de decolagens realizadas,%d\n",
total_decolagens);
352.
353.         fclose(arquivo);
354.         printf("\nEstatísticas gravadas e arquivo
'avioes_pipe.csv' fechado com sucesso.\n");
355.     }
356.
357.     pthread_join(monitor_thread, NULL);
358.     pthread_mutex_destroy(&contador_mutex);
359.     exit(0);
360. }

```

Bloco de código 31: Função de interrupção

4.1.11. Função de inicialização do contador

Esta função é responsável por inicializar e criar a thread de monitoramento, A estrutura ThreadArgs é alocada dinamicamente usando malloc, e seus campos (**contador_pousos** e **contador_decolagens**) são inicializados com o valor 0, pois ainda não houve nenhuma operação de pouso ou decolagem.

A função pthread_create é chamada para criar a thread. O primeiro parâmetro &monitor_thread passa a referência do identificador da thread. O segundo parâmetro NULL indica que a thread será criada com as configurações padrão. O terceiro parâmetro (void *) thread_funcao especifica a função que a thread executará, que é thread_function. O argumento da thread será args, que é um ponteiro para os argumentos que serão passados para a função

Caso a criação da thread falhe, a função imprime uma mensagem de erro e encerra o programa com **exit (1)**.

```
144. // Adicione esta função após a main() para iniciar a thread
145.
146. void iniciar_contador()
147. {
148.     ThreadArgs *args = malloc(sizeof(ThreadArgs));
149.     // Aloca memória para os argumentos da thread
150.
151.     args->contador_pousos = 0;           // Parâmetro 1
152.     args->contador_decolagens = 0;      // Parâmetro 2
153.
154.     // Cria a thread
155.     if (pthread_create(&monitor_thread, NULL, (void *)
156.         thread_function, args) != 0)
157.     {
158.         perror("Erro ao criar thread de monitoramento");
159.
160.     // Erro ao criar a thread
161.         free(args);                     // Libera a memória alocada
162.         exit(1);                       // Encerra o processo
163.     }
164. }
```

Bloco de código 32: Criação da Thread

4.2. Funcionamento da aplicação – Resumo

4.2.1. Inicialização do Sistema

O programa começa criando dois pipes de comunicação essenciais: um para as requisições dos aviões para a torre (**piperequisicao**) e outro para as respostas da torre para os aviões (**pipecontrole**). O sistema utiliza o PID (**Process ID**) como identificador único para cada avião, garantindo que não haja conflitos de identificação.

4.2.2. Estrutura da Torre de Controle

A torre de controle é implementada como um processo filho principal que gerencia todas as operações do aeroporto. Ela mantém um registro constante do número de aviões no aeroporto, que não pode exceder o limite máximo definido (**MAX_AVIOES_AEROPORTO = 3**). A torre também é responsável por registrar todas as operações em um arquivo CSV (**avioes_pipe.csv**), mantendo um histórico detalhado de todas as movimentações.

4.2.3. Funcionamento dos Aviões

Cada avião é criado como um processo filho independente. Inicialmente, todos os aviões começam no estado **VOANDO**, com posições (latitude e longitude) e altitude geradas aleatoriamente. Os aviões constantemente alternam entre estados de voo e solo, solicitando permissões para pouso quando estão voando e para decolagem quando estão no solo.

4.2.4. Processo de Pouso:

Quando um avião deseja pousar, ele envia uma solicitação para a torre através do pipe de requisição. A torre verifica se há espaço disponível no aeroporto (menos de 3 aviões). Se houver espaço, a permissão é concedida e o avião inicia o procedimento de pouso, que leva 5 segundos (**TEMPO_POUSO**). Após pousar, o avião permanece no solo por um tempo aleatório entre 10 e 19 segundos antes de solicitar decolagem.

4.2.5. Processo de Decolagem

Quando um avião no solo deseja decolar, ele envia uma solicitação à torre. Como não há limite para aviões no ar, a decolagem é sempre autorizada. O processo de decolagem leva 3 segundos (**TEMPO_DECOLAGEM**). Após decolar, o avião permanece voando por um período aleatório entre 10 e 19 segundos antes de solicitar novo pouso.

4.2.6. Monitoramento e Estatísticas

A thread separada monitora continuamente as estatísticas do aeroporto, exibindo a cada 10 segundos o número total de pousos e decolagens realizados. Esta thread utiliza um mutex para garantir acesso seguro aos contadores globais, evitando condições de corrida quando múltiplos processos tentam atualizar estes valores simultaneamente.

4.2.7. Registro de Operações

Todas as operações são registradas em um arquivo CSV que contém informações detalhadas de cada avião: ID, posição (latitude e longitude), altitude e status da permissão.

4.2.8. Tratamento de Interrupções

O sistema implementa um tratamento especial para o sinal SIGINT (Ctrl+C), garantindo um encerramento do programa. Quando o usuário solicita o término, o sistema finaliza a thread de monitoramento, registra as estatísticas finais no arquivo CSV, fecha adequadamente todos os arquivos abertos e libera os recursos utilizados.

4.2.9. Diagrama esquemático

Finalmente após a colocação de todos esses pontos a imagem abaixo representa um diagrama com os componentes e processos existentes dentro da aplicação.

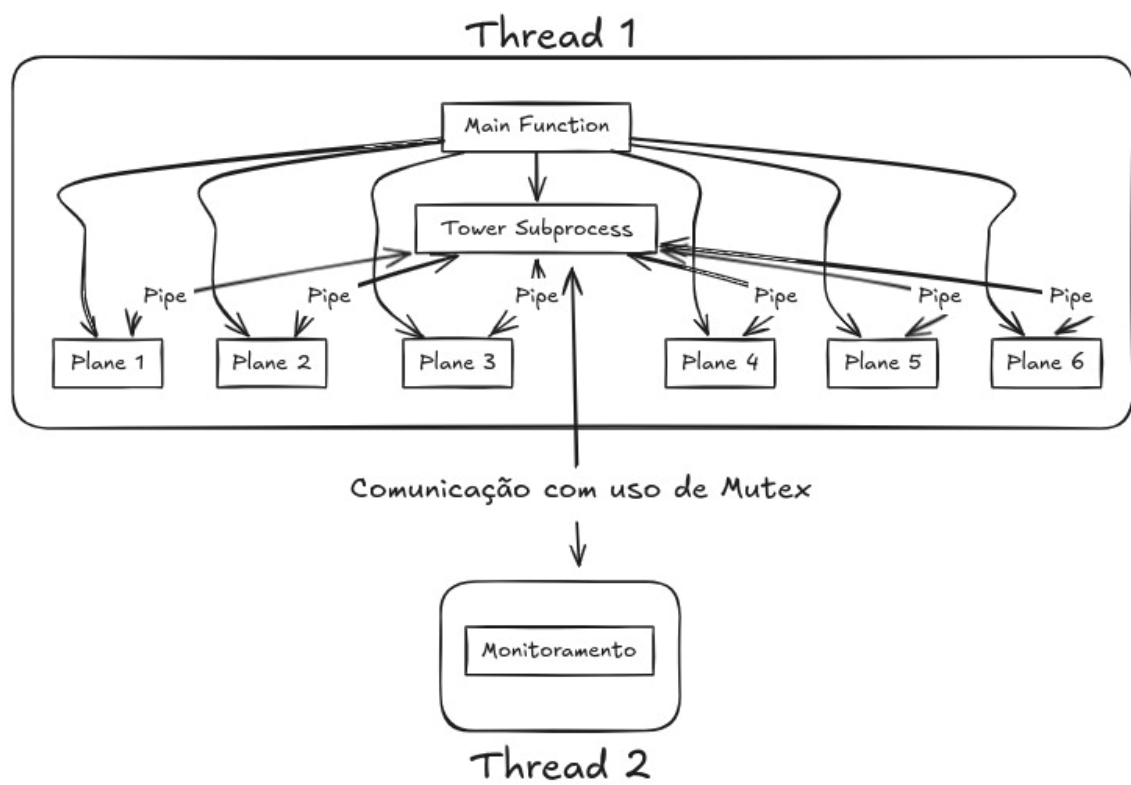
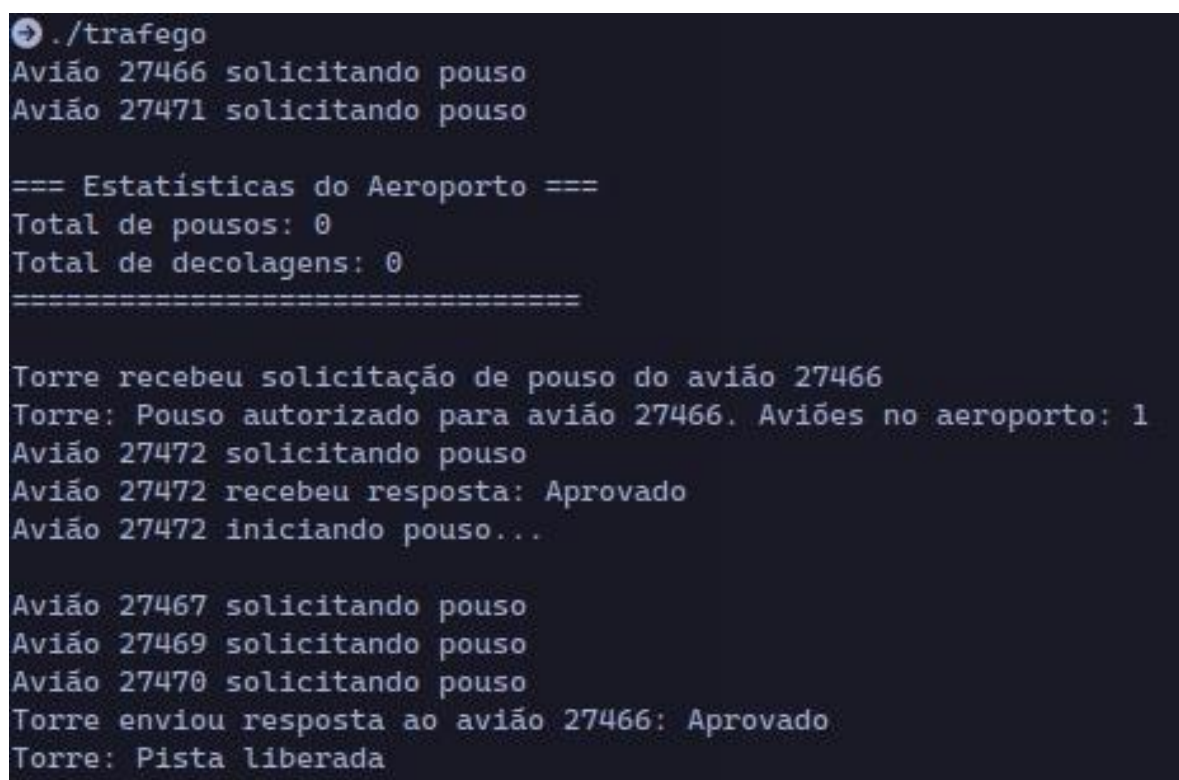


Figura 4: Diagrama dos processos concorrentes na aplicação – Elaborado pelos autores.

5. TELAS (PRINTS) DA EXECUÇÃO

A seguir, serão apresentados screenshots que ilustram as diferentes etapas da execução do programa. A tela inicial é iniciada. Em seguida, serão exibidos exemplos de como o programa simula decolagens e pousos, atualizando o contador de aeronaves e registrando os eventos no arquivo CSV. Por fim, será apresentada a tela final, indicando o término da simulação e o fechamento do arquivo CSV.

A imagem abaixo a simulação do tráfego aéreo foi iniciada. Todos os processos e thread foram acionados para dar início à simulação. Este trecho da simulação demonstra o momento em que a torre de controle recebe múltiplas solicitações de pouso. A torre, seguindo um protocolo de atendimento, autoriza o pouso de um dos aviões e inicia o processo de liberação da pista.



```
➔ ./trafego
Avião 27466 solicitando pouso
Avião 27471 solicitando pouso

=== Estatísticas do Aeroporto ===
Total de pousos: 0
Total de decolagens: 0
=====

Torre recebeu solicitação de pouso do avião 27466
Torre: Pouso autorizado para avião 27466. Aviões no aeroporto: 1
Avião 27472 solicitando pouso
Avião 27472 recebeu resposta: Aprovado
Avião 27472 iniciando pouso...

Avião 27467 solicitando pouso
Avião 27469 solicitando pouso
Avião 27470 solicitando pouso
Torre enviou resposta ao avião 27466: Aprovado
Torre: Pista liberada
```

Figura 5: Captura de tela da inicialização do programa

A próxima imagem apresenta a sequência de eventos que demonstra como a torre de controle autoriza pousos e decolagens, considerando a capacidade do aeroporto e a segurança das operações. O programa registra solicitações, autorizações, status das aeronaves e tempo de espera no solo ou em voo.

```

Torre recebeu solicitação de pouso do avião 23309
Torre: Pouso autorizado para avião 23309. Aviões no aeroporto: 3
Torre enviou resposta ao avião 23309: Aprovado
Avião 23309 recebeu resposta: Aprovado
Avião 23309 iniciando pouso...

Torre: Pista liberada

Torre recebeu solicitação de decolagem do avião 23314
Torre: Decolagem autorizada para avião 23314. Aviões no aeroporto: 2
Torre enviou resposta ao avião 23314: Aprovado
Torre: Pista liberada

Avião 23314 recebeu resposta: Aprovado
Avião 23314 iniciando decolagem...

Avião 23309 ficará parado por 11 segundos

Avião 23313 solicitando pouso
Avião 23310 solicitando decolagem
Avião 23314 ficará voando por 12 segundos

```

Figura 6: Solicitações de decolagem e pouso

Nesta parte do processo, uma thread especializada incrementa os contadores de pousos e decolagens a cada vez que uma aeronave realiza uma dessas operações.

```

Torre recebeu solicitação de pouso do avião 23312
Torre: Pouso autorizado para avião 23312. Aviões no aeroporto: 3
Torre enviou resposta ao avião 23312: Aprovado
Torre: Pista liberada

Avião 23312 recebeu resposta: Aprovado
Avião 23312 iniciando pouso...

=== Estatísticas do Aeroporto ===
Total de pousos: 9
Total de decolagens: 6
=====

Torre recebeu solicitação de pouso do avião 23313
Torre: Pouso negado para avião 23313. Aeroporto lotado. Aviões no
Torre enviou resposta ao avião 23313: Negado
Avião 23313 recebeu resposta: Negado
Avião 23313 aguardando novo pedido para pouso...

Avião 23312 ficará parado por 15 segundos

```

Figura 7: Demonstração do contador.

A imagem mostra um trecho da saída de um simulador de controle de tráfego aéreo. A primeira linha indica que o avião 23312 está aguardando para decolar. A segunda linha registra uma solicitação de pouso do avião 23315. A última linha confirma que as estatísticas de voo foram salvas em um arquivo CSV após o usuário interromper o programa pressionando Ctrl+C

```
Avião 23312 ficará parado por 15 segundos
Avião 23315 solicitando pouso
^C
Estatísticas gravadas e arquivo 'avioes_pipe.csv' fechado com sucesso.
```

Figura 8: Fim da execução do programa

E finalmente o print abaixo apresenta o registro histórico das solicitações de voo de um determinado período, incluindo informações sobre a localização e altitude dos voos. A coluna "Permissão" indica se a solicitação foi aprovada ou negada pelos controladores de tráfego aéreo, além da contabilização das estatísticas finais.

```
ID, Latitude, Longitude, Altitude, Permissao
23309, 81.679573, 130.339798, 22485, Aprovado
23310, -33.828327, 98.967484, 34052, Aprovado
23312, 5.107494, 36.012432, 4098, Aprovado
23314, 44.041248, -26.884007, 20363, Negado
23313, 69.354797, -175.766998, 24384, Negado
23315, -71.576393, -58.302277, 9526, Negado
23315, -71.576393, -58.302277, 9526, Negado
23314, 44.041248, -26.884007, 20363, Negado
23309, 81.679573, 130.339798, 22485, Aprovado
23310, -33.828327, 98.967484, 34052, Aprovado
23313, 69.354797, -175.766998, 24384, Aprovado
23312, 5.107494, 36.012432, 4098, Aprovado
23315, -71.576393, -58.302277, 9526, Aprovado
23314, 44.041248, -26.884007, 20363, Aprovado
23310, -33.828327, 98.967484, 34052, Negado
23309, 81.679573, 130.339798, 22485, Negado
23313, 69.354797, -175.766998, 24384, Aprovado
23310, -33.828327, 98.967484, 34052, Aprovado
23312, 5.107494, 36.012432, 4098, Negado
23315, -71.576393, -58.302277, 9526, Aprovado
23309, 81.679573, 130.339798, 22485, Aprovado
23314, 44.041248, -26.884007, 20363, Aprovado
23312, 5.107494, 36.012432, 4098, Aprovado
23313, 69.354797, -175.766998, 24384, Negado

=== Estatísticas Finais ===
Total de pousos realizados, 9
Total de decolagens realizadas, 6
```

Figura 9: Arquivo .csv utilizado para permanência de dados

6. REFERENCIAS

KING, K. N. **C Programming: A Modern Approach**. 2. ed. Boston: W. W. Norton & Company, 2008.

PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS. **Código-fonte: chat_de_pobre** Disponível em: https://pucminas.instructure.com/courses/207863/files/11946268?module_item_id=4521672. Acesso em: 12 nov. 2024.

PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS. **Código-fonte: threadSample.c**. Disponível em: https://pucminas.instructure.com/courses/207863/files/11946268?module_item_id=4521672. Acesso em: 12 dez. 2024.

PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS. **Guia completo para elaborar e formatar trabalhos científicos**. Disponível em: <https://www.pucminas.br/biblioteca/DocumentoBiblioteca/ABNT-GUIA-COMPLETO-Elaborar-formatar-trabalho-cientifico.pdf>. Acesso em: 12 dez. 2024.

TANENBAUM, Andrew S. **Sistemas operacionais modernos**. 4. ed. São Paulo: Pearson, 2016.

TODCANI, Simão Sirineo; OLIVEIRA, Rômulo Silva de; CARISSIMI, Alexandre Silva. **Sistemas operacionais e programação concorrente**. 2. ed. São Paulo: Pearson Prentice Hall, 2010.