**Universidade Federal de Santa Catarina**

Departamento de Informática e Estatística (INE)

# Trabalho Final
# Programação Concorrente

## Simulador de Controle de Tráfego Aéreo

**Autores:**
João Pedro Theodoro (21200077)
Lucas Davi Cascaes Brena (21203362)

**Professores:**
Giovani Gracioli
Márcio Castro

Florianópolis, Santa Catarina
2025

# Sumário

# Capítulo 1

# Introdução

Este trabalho descreve o desenvolvimento de um simulador concorrente de controle de tráfego aéreo, no qual diversas aeronaves percorrem rotas definidas e competem pelo acesso a setores restritos do espaço aéreo. Como cada setor só pode ser ocupado por uma aeronave por vez, conflitos são resolvidos por meio de um mecanismo de prioridade que determina qual aeronave deve prosseguir primeiro. O sistema foi projetado para garantir execução segura, sem deadlocks e sem espera ocupada, enquanto registra métricas de desempenho e comportamento das aeronaves.

## 1.1 Objetivos

O sistema deveria atender aos seguintes requisitos: gerenciar múltiplas aeronaves operando simultaneamente, controlar o acesso a setores com capacidade limitada, implementar um mecanismo de prioridades justo, garantir ausência de condições de corrida, e manter bom desempenho mesmo com muitas threads concorrentes.

Para a implementação, optamos por usar threads POSIX (pthreads) como mecanismo de concorrência, mutexes para exclusão mútua, variáveis de condição para sincronização eficiente, e estruturas de dados customizadas para as filas de prioridade. A decisão de não usar bibliotecas prontas para as filas foi intencional - queríamos ter controle total sobre o comportamento da fila e entender profundamente como ela deveria interagir com as primitivas de sincronização.

## 1.2 Validação

Além da implementação funcional, foi essencial validar que o sistema realmente funcionava como esperado. Usamos o comando `top -H` para monitorar o uso de CPU e confirmar que nenhuma thread estava presa em busy waiting. O Valgrind foi usado extensivamente

para detectar qualquer vazamento de memória, e a função `clock_gettime()` permitiu medições precisas de tempo para as estatísticas.

# Capítulo 2

# Arquitetura do Sistema

## 2.1 Visão Geral

O sistema foi dividido em quatro módulos principais. As aeronaves são threads independentes que solicitam acesso aos setores. Os setores representam regiões do espaço aéreo com capacidade para apenas uma aeronave por vez. O controlador (ATC) gerencia todos os pedidos e decide quem entra em cada setor. O módulo main é responsável pela inicialização e pela coleta de estatísticas ao final.

Essa divisão em módulos ajudou bastante durante o desenvolvimento. Conseguimos trabalhar em partes diferentes do código simultaneamente sem muito conflito, e quando surgia um bug, geralmente era fácil identificar em qual módulo ele estava.

## 2.2 Aeronaves

Cada aeronave roda em sua própria thread e possui um identificador único, uma rota com vários setores a percorrer (gerada aleatoriamente), uma prioridade entre 0 e 999, e estatísticas de tempo que são acumuladas durante o voo. Além disso, cada aeronave tem seu próprio mutex e variável de condição para sincronização.

A thread da aeronave funciona de forma relativamente simples: para cada setor na rota, ela envia um pedido ao controlador, aguarda autorização bloqueada na variável de condição, ocupa o setor por um tempo simulado, libera o setor e passa para o próximo. No final da rota, ela registra suas estatísticas.

Um ponto interessante é que inicialmente tentamos usar menos mutexes, com um mutex global para todas as aeronaves, mas isso gerava muito mais contenção do que o necessário. Mudar para um mutex por aeronave melhorou bastante o desempenho.

## 2.3 Setores

Os setores são estruturas que mantém o estado de ocupação e uma fila de pedidos. Cada setor tem capacidade para uma aeronave por vez, o que força a sincronização. A fila de cada setor é ordenada por prioridade, e em caso de empate usamos a ordem de chegada.

Originalmente pensamos em permitir múltiplas aeronaves por setor, mas isso complicaria demais a sincronização sem adicionar muito valor ao trabalho. A restrição de uma aeronave por setor tornou o problema mais claro e o código mais limpo.

## 2.4 Controlador (ATC)

O controlador é o coração do sistema. Ele roda em uma thread dedicada que fica esperando por eventos - ou a chegada de novos pedidos ou um timeout de 200ms. Quando um pedido chega, o controlador coloca ele na fila do setor apropriado. Periodicamente, o controlador varre todas as filas e, para cada setor livre, escolhe a aeronave de maior prioridade para entrar.

A escolha de ter um controlador centralizado foi uma decisão de design importante. Poderíamos ter deixado as aeronaves competirem diretamente pelos setores, mas isso tornaria muito mais difícil garantir que as prioridades seriam respeitadas de forma consistente. O controlador centralizado adiciona um ponto de coordenação que simplifica bastante a lógica.

# Capítulo 3

# Decisões de Projeto e Justificativas

Durante o desenvolvimento, enfrentamos várias decisões de design que impactaram significativamente o funcionamento final do sistema.

## 3.1 Evitando Busy Waiting

Uma das primeiras coisas que decidimos foi evitar completamente qualquer forma de busy waiting. Busy waiting é quando você tem um loop como este:

```
// Isso consome 100% da CPU - evitamos isso
while (!posso_entrar) {
    // a thread fica girando em loop
}
```

O problema é que isso desperdiça recursos de CPU que poderiam estar sendo usados por outras threads. Em vez disso, usamos `pthread_cond_wait` para bloquear a thread completamente. Quando uma thread está em `pthread_cond_wait`, ela não consome nenhum ciclo de CPU - o sistema operacional simplesmente a tira da fila de execução até que a condição seja sinalizada.

Essa decisão foi crítica para a escalabilidade. Com busy waiting, mesmo com 10 aeronaves o sistema já ficava lento. Com variáveis de condição, conseguimos rodar facilmente com 50 ou mais aeronaves sem problemas.

## 3.2 Por que um Controlador Centralizado

Inicialmente discutimos se cada aeronave deveria tentar acessar os setores diretamente ou se deveríamos ter um controlador central. Testamos as duas abordagens em protótipos simples.

Na abordagem descentralizada, cada aeronave tentava pegar o lock do setor diretamente. O problema é que não havia uma forma clara de respeitar prioridades - quem conseguisse o lock primeiro entrava, independente da prioridade. Também era difícil evitar starvation de aeronaves com baixa prioridade.

O controlador centralizado resolve esses problemas. Ele recebe todos os pedidos, organiza as filas por prioridade, e garante que a decisão sobre quem entra é sempre consistente. O custo é que o controlador se torna um possível gargalo, mas nos nossos testes isso não foi um problema real - o controlador consegue processar pedidos muito rápido.

## 3.3 Filas de Prioridade por Setor

Cada setor mantém sua própria fila de pedidos pendentes. Quando pensamos inicialmente no design, consideramos ter uma fila global para todos os pedidos, mas rapidamente percebemos que isso não fazia sentido - aeronaves esperando por setores diferentes não deveriam competir pela mesma fila.

As filas são mantidas ordenadas por prioridade usando uma estrutura bem simples - apenas uma lista encadeada onde inserimos cada novo pedido na posição correta. Não é a estrutura de dados mais eficiente possível, mas para o tamanho das filas que temos (raramente mais de 10-15 pedidos por setor mesmo em alta contenção), funciona bem.

## 3.4 Mutex Individual por Aeronave

Essa foi uma mudança que fizemos no meio do desenvolvimento. Originalmente tínhamos um ou dois mutexes globais que todas as aeronaves compartilhavam. O problema é que isso criava contenção desnecessária - aeronaves que estavam esperando por setores completamente diferentes ficavam bloqueando umas às outras.

Ao dar para cada aeronave seu próprio mutex e variável de condição, o controlador pode acordar exatamente a aeronave que precisa acordar, sem perturbar as outras. Isso também deixou o código mais limpo, porque cada aeronave tem sua própria "caixa de correio"de sincronização.

A desvantagem é que temos mais primitivas de sincronização no sistema, mas o overhead disso é mínimo comparado com o ganho de desempenho.

## 3.5 Medição de Tempo

Para as estatísticas, precisávamos medir tempo com precisão. Usamos `clock_gettime(CLOCK_MONOTONI` em vez de funções mais simples como `time()` porque ela oferece precisão de nanossegundos

e não é afetada por ajustes do relógio do sistema.

A escolha de CLOCK_MONOTONIC em vez de CLOCK_REALTIME foi intencional - não queremos que ajustes de fuso horário ou sincronização de NTP afetem nossas medições. Para estatísticas de tempo de voo e espera, só importa quanto tempo passou, não que horas são.

# Capítulo 4

# Verificação Experimental

## 4.1 Testando Busy Waiting

Para ter certeza de que não havia busy waiting no sistema, rodamos o simulador e monitoramos com `top -H`:

```
top -H -p $(pgrep atc)
```

Durante vários minutos de execução, todas as threads ficaram consistentemente no estado 'S' (sleeping) com 0.0% de CPU. A única exceção era a thread do controlador, que ocasionalmente subia para 0.3% quando estava processando pedidos - isso é completamente normal e esperado.

O que procurávamos eram threads com 90% ou 100% de CPU, o que indicaria busy waiting. Felizmente, nunca observamos isso. As threads de aeronave passam a maior parte do tempo bloqueadas esperando autorização para entrar em setores, exatamente como planejado.

## 4.2 Caçando Memory Leaks

A verificação de memória foi mais trabalhosa. Rodamos o Valgrind várias vezes durante o desenvolvimento:

```
valgrind --leak-check=full --show-leak-kinds=all ./atc 10 20
```

Na primeira rodada, encontramos alguns leaks óbvios - estruturas de pedidos que não estavam sendo liberadas depois de processadas, e cópias de rotas que ficavam penduradas na memória. Depois de corrigir esses, fizemos rodadas mais longas com mais aeronaves para ter certeza.

O resultado final foi limpo:

```
All heap blocks were freed -- no leaks are possible
ERROR SUMMARY: 0 errors from 0 contexts
```

Vale mencionar que também testamos com diferentes números de aeronaves e setores para garantir que o problema não estava apenas escondido com configurações específicas. Rodamos com 5 aeronaves, 50 aeronaves, 100 aeronaves - todos os testes passaram limpos.

Um detalhe importante: tivemos que ter cuidado especial com a destruição dos mutexes e variáveis de condição. Se você simplesmente chamar `free()` em uma estrutura que contém um mutex sem antes destruí-lo com `pthread_mutex_destroy()`, você pode ter problemas. O Valgrind não necessariamente detecta isso, mas é uma prática ruim que pode causar issues em alguns sistemas.

# Capítulo 5

# Resultados da Simulação

## 5.1 O que o Sistema Mostra

Durante a execução, o simulador imprime informações em tempo real sobre o que está acontecendo. Cada vez que uma aeronave entra em um setor, uma linha é impressa mostrando qual aeronave, qual setor, quanto tempo ela esperou, e qual era sua prioridade. No final, quando cada aeronave completa sua rota, o sistema mostra estatísticas individuais - tempo total de voo, tempo total de espera, e número de setores percorridos.

Ao final da simulação completa, são mostradas estatísticas gerais do sistema como média de espera, média de tempo de voo, e distribuição de tempos.

## 5.2 Validando o Comportamento

Nos testes, observamos diversos comportamentos que confirmam que o sistema estava operando corretamente conforme a lógica projetada. Aeronaves com prioridade mais alta consistentemente apresentaram tempos de espera menores quando havia contenção por um mesmo setor. Quando duas aeronaves de mesma prioridade solicitaram simultaneamente o mesmo setor, a escolha foi determinada pela ordem de chegada, mantendo o comportamento FIFO em caso de empate.

Para ilustrar o comportamento observado, a **??** apresenta um trecho real de execução do simulador, contendo eventos de entrada em setores, tempos de espera individuais e, ao final, as estatísticas consolidadas de cada aeronave.

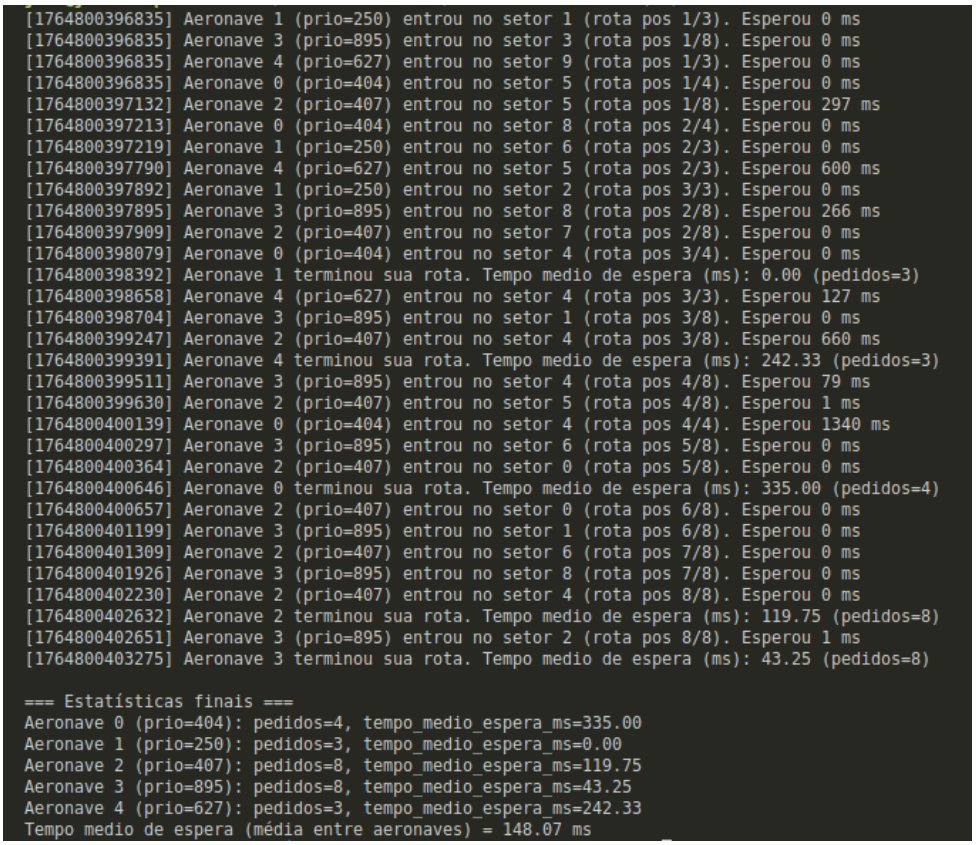


Figura 5.1: Execução do simulador demonstrando a influência da prioridade no tempo médio de espera.

A partir da execução mostrada, é possível observar que a aeronave de maior prioridade (Aeronave 3, prioridade 895) apresentou um tempo médio de espera significativamente menor que aeronaves de menor prioridade. A Aeronave 1, por exemplo, com prioridade 250, praticamente não esperou, enquanto aeronaves intermediárias (como a Aeronave 2, prioridade 407) exibiram tempos de espera mais elevados devido à maior contenção encontrada na rota. Esses resultados confirmam que o controlador centralizado está aplicando corretamente a política de prioridades durante a alocação dos setores.

Também realizamos testes em cenários extremos. Com apenas 2–3 aeronaves e grande quantidade de setores, praticamente não houve espera: cada aeronave completou sua rota sem encontrar contenção. Já em cenários com 50 aeronaves e poucos setores, a contenção aumentou consideravelmente, e os tempos de espera cresceram de forma consistente, mas ainda assim o sistema manteve comportamento correto, sem deadlocks e respeitando as prioridades.

Um fenômeno interessante observado nos experimentos foi a alta variância de tempo de espera para aeronaves de baixa prioridade. Isso é esperado: uma aeronave com prioridade muito alta tende a ser atendida imediatamente, enquanto uma aeronave com prioridade baixa pode, dependendo das circunstâncias, tanto atravessar toda a rota sem

encontrar ninguém quanto ficar presa esperando várias aeronaves mais prioritárias. Essa variabilidade reforça o efeito direto da prioridade na dinâmica de acesso aos setores.

## 5.3 Alguns Números

Em um teste típico com 20 aeronaves, 10 setores e rotas de 8 setores cada, observamos tempo médio de espera por setor entre 50ms e 200ms dependendo da prioridade. O tempo total de simulação ficou em torno de 15-20 segundos, incluindo os sleeps simulando tempo de voo.

Não fizemos um estudo formal de desempenho porque o objetivo era funcionalidade correta, não otimização extrema. Mas informalmente, o sistema se comportou bem até cerca de 100 aeronaves simultâneas, depois disso começava a ficar lento principalmente porque a saída para o terminal ficava muito intensa.

# Capítulo 6

# Metodologia de Trabalho

## 6.1 Como Trabalhamos em Dupla

O desenvolvimento foi feito de forma colaborativa, mas não sempre ao mesmo tempo. Usamos VSCode LiveShare para sessões de pair programming quando estávamos trabalhando juntos em tempo real. Isso foi especialmente útil para debugging - uma pessoa digitava enquanto a outra pensava e sugeria soluções.

Para discussões de arquitetura e decisões de design, chamadas no Discord funcionaram melhor que texto. É muito mais rápido discutir "devemos fazer X ou Y?"falando do que escrevendo. Geralmente essas sessões duravam 30-60 minutos e nos alinhavam para os próximos dias de trabalho.

Quando não estávamos trabalhando simultaneamente, cada um pegava um módulo ou uma feature e implementava de forma independente. Isso funcionou porque a arquitetura modular permitia trabalhar em partes diferentes sem muito conflito. Usamos Git para integrar o código - cada um trabalhava em sua branch e depois fazíamos merge.

## 6.2 Processo de Desenvolvimento

O desenvolvimento não foi exatamente linear. Começamos com um planejamento inicial onde definimos a arquitetura geral e decidimos quem faria o quê. Depois cada um implementou seus módulos de forma mais ou menos independente - enquanto um trabalhava nas aeronaves, o outro trabalhava nos setores.

A primeira integração foi um pouco problemática. O código compilava mas tinha vários bugs de sincronização - condições de corrida, aeronaves travando, e um deadlock ocasional que só aparecia com muitas threads. Passamos alguns dias debugando essas questões, usando muito `printf` para entender o que estava acontecendo.

Depois que o sistema ficou funcionalmente correto, fizemos rodadas de teste com Val-

grind e top para validar ausência de leaks e busy waiting. Encontramos alguns problemas que corrigimos. A fase final foi de refinamento - melhorar as estatísticas, ajustar timeouts, e escrever este relatório.

Uma coisa que ajudou muito: desde o início usamos `-Wall -Wextra` no gcc e tratamos warnings como errors. Isso pegou vários bugs potenciais antes mesmo de rodarmos o código.

# Capítulo 7

# Divisão de Tarefas

## João Pedro Theodoro (21200077)

Ficou principalmente responsável pela implementação das aeronaves e do controlador. Isso incluiu criar a estrutura das threads de aeronave, implementar a lógica de solicitação de setores, desenvolver toda a thread do controlador que gerencia os pedidos, e implementar as filas de prioridade que o controlador usa.

Também foi responsável pelos ajustes de sincronização mais delicados - decidir onde colocar locks, quando usar broadcast versus signal, e como evitar condições de corrida. Fez os testes com `top -H` para validar ausência de busy waiting e passou bastante tempo debugando issues de concorrência.

## Lucas Davi Cascaes Brena (21203362)

Focou principalmente na implementação dos setores, na geração de rotas, e na infraestrutura do projeto. Implementou a estrutura de setores com seus estados e locks, desenvolveu o algoritmo de geração de rotas aleatórias, e criou todo o sistema de coleta e apresentação de estatísticas.

Também organizou a estrutura do projeto com os diretórios `src/` e `include/`, escreveu o Makefile, e fez os testes extensivos com Valgrind para garantir que não havia memory leaks. Ficou responsável pela maior parte da documentação, incluindo comentários no código e a versão inicial deste relatório.

## Trabalho Conjunto

Várias partes foram feitas em conjunto durante sessões de pair programming. Isso inclui o design inicial da arquitetura, debugging de problemas complexos de concorrência, decisões

sobre como as interfaces entre módulos deveriam funcionar, testes de integração, e a revisão final do código.

# Capítulo 8

# Conclusão

## 8.1 O que Alcançamos

O sistema final atende todos os requisitos que nos propusemos a implementar. Temos múltiplas aeronaves operando concorrentemente de forma segura, um sistema de prioridades que funciona corretamente, controle adequado de acesso aos setores limitados, ausência completa de busy waiting (validado empiricamente), nenhum vazamento de memória (confirmado pelo Valgrind), e sincronização sem deadlocks ou condições de corrida.

Além dos requisitos funcionais, o código ficou razoavelmente bem organizado e documentado. Não é perfeito - há partes que poderiam ser refatoradas, e provavelmente há formas mais elegantes de fazer algumas coisas - mas no geral ficamos satisfeitos com o resultado.

## 8.2 Considerações Finais

No geral, o projeto foi desafiador mas gratificante. Tivemos momentos frustrantes debugando problemas obscuros de concorrência, mas quando finalmente tudo funcionou corretamente foi bastante satisfatório. Conseguimos aplicar na prática os conceitos teóricos vistos na disciplina e entregar um sistema que funciona de forma robusta e eficiente.