Universidade Federal de Pelotas
Centro de Desenvolvimento Tecnológico
Curso de Graduação em Ciência da Computação

Trabalho Final de Sistemas Operacionais

Implementação de um Simulador de Tráfego Aéreo para Análise de Concorrência em Sistemas Operacionais

---

Autor:
Guilherme Hepp da Fonseca

17 de agosto de 2025

Resumo

A gestão de concorrência é um desafio central em Sistemas Operacionais, sendo crucial para a estabilidade e eficiência de sistemas multi-threaded. Este trabalho explora na prática os problemas de *deadlock* (impasse) e *starvation* (fome) em um ambiente simulado de alta disputa por recursos. O objetivo principal foi desenvolver um simulador de controle de tráfego aéreo, utilizando a linguagem C e a biblioteca PThreads, para modelar e analisar o impacto de diferentes políticas de alocação de recursos em um aeroporto internacional. A metodologia envolveu a representação de aeronaves como threads concorrentes e dos recursos do aeroporto (pistas, portões, torre) como semáforos. Uma estratégia de prevenção de deadlock, baseada em espera com tempo limite (*timeout*), foi implementada para quebrar a condição de "reter e esperar". Adicionalmente, foi desenvolvida uma política de priorização para threads em estado crítico, a fim de mitigar a ocorrência de *starvation*. Os resultados da simulação demonstraram a eficácia da estratégia de prevenção, não registrando nenhum deadlock, e evidenciaram como a ordem de alocação de recursos pode levar a cenários de *starvation*, que foram mitigados pela política de prioridade. O projeto serve como uma validação prática da importância fundamental das estratégias de gerenciamento de concorrência.

Palavras-chave: Sistemas Operacionais, Concorrência, PThreads, Deadlock, Starvation, Semáforos.

Abstract

Concurrency management is a central challenge in Operating Systems, crucial for the stability and efficiency of multi-threaded systems. This work provides a practical exploration of deadlock and starvation problems in a simulated environment with high resource contention. The main objective was to develop an air traffic control simulator, using the C language and the PThreads library, to model and analyze the impact of different resource allocation policies at an international airport. The methodology involved representing aircraft as concurrent threads and airport resources (runways, gates) as semaphores. A deadlock prevention strategy, based on a timed wait, was implemented to break the "hold-and-wait"condition. Additionally, a priority policy for threads in a critical state was developed to mitigate the occurrence of starvation. The simulation results demonstrated the effectiveness of the prevention strategy, with no deadlocks recorded, and showed how the resource allocation order can lead to starvation scenarios, which were mitigated by the priority policy. The work serves as a practical validation of the importance of concurrency management strategies.

Keywords: Operating Systems, Concurrency, PThreads, Deadlock, Starvation, Semaphores.

Sumário

1 Introdução

O gerenciamento de recursos compartilhados é um aspecto essencial em sistemas concorrentes, especialmente em cenários onde múltiplos usuários ou processos competem por recursos limitados. Este trabalho apresenta a modelagem e simulação de um sistema de controle de tráfego aéreo em um aeroporto internacional com alta demanda de operações, onde diferentes tipos de voos — domésticos e internacionais — disputam três recursos principais: pistas, portões de embarque e a capacidade de atendimento da torre de controle.

A simulação foi desenvolvida em linguagem C, utilizando programação concorrente através da biblioteca PThreads para modelar o comportamento de cada avião como uma thread independente. Para garantir a sincronização e o controle de acesso aos recursos, foram utilizados semáforos POSIX, evitando condições de corrida e garantindo a exclusão mútua.

A principal fonte de conflito e potencial deadlock no sistema reside na diferença da ordem de alocação de recursos entre os voos. Voos internacionais, que possuem prioridade implícita, solicitam os recursos em uma sequência, enquanto voos domésticos o fazem em outra. Para prevenir o bloqueio total do sistema, foi implementada uma estratégia de espera com tempo limite (timeout), na qual uma thread libera os recursos que já possui se não conseguir obter os demais dentro de um curto intervalo. Essa abordagem quebra a condição de "reter e esperar"(hold-and-wait), uma das condições necessárias para a ocorrência de deadlock.

Adicionalmente, para analisar e mitigar o problema de starvation (fome), a simulação implementa uma política de escalonamento dinâmico. Voos domésticos, ao aguardarem por um tempo considerado crítico (60 segundos), têm sua prioridade elevada e passam a utilizar a mesma estratégia de alocação dos voos internacionais. Esta política visa aumentar a justiça (fairness) do sistema e reduzir o número de falhas operacionais, representadas por "quedas"de aeronaves que excedem 90 segundos de espera.

Ao final de cada execução, o sistema coleta e exibe estatísticas detalhadas, como o tempo médio de espera, a taxa de utilização dos recursos e o número de voos que completaram suas operações com sucesso ou falharam. Isso permite uma análise quantitativa da eficiência do aeroporto e da eficácia das políticas de gerenciamento de concorrência implementadas.

2 Desenvolvimento da Solução

A implementação do simulador de tráfego aéreo foi desenvolvida na linguagem C, utilizando a biblioteca PThreads para o gerenciamento de concorrência. A arquitetura foi estruturada em um conjunto de funções e uma estrutura de dados principal, onde cada componente possui uma responsabilidade bem definida no controle, criação e execução das threads. A seguir, é apresentada uma visão geral dos principais componentes da solução implementada.

2.1 Visão Geral da Implementação

O programa foi organizado de forma procedural, centrando a lógica em funções específicas e em uma estrutura de dados que representa o estado de cada avião.

- **Função `main()`:** Atua como o orquestrador principal da simulação. É responsável por inicializar os recursos (semáforos) com base nos argumentos de linha de comando, disparar a thread do monitor e entrar no laço principal de criação das threads de aviões. Ao final do tempo estipulado, ela encerra a criação de novas threads, aguarda a finalização das que estão em execução e, por fim, gera e exibe o relatório final de estatísticas.
- **Struct `aviao`:** É a estrutura de dados central que encapsula todas as informações de estado de uma aeronave, como seu ID, tipo (doméstico ou internacional), e as flags de controle para o estado de alerta e o estado crítico, que ativam a política de mitigação de starvation. Cada thread de avião recebe um ponteiro para uma instância única desta estrutura.
- **Função `threadAviao()`:** Representa o ciclo de vida completo de um avião. É a função executada por cada thread criada na `main`. Sua única responsabilidade é chamar, na sequência correta, as três principais fases da operação: `pousar()`, `desembarcar()` e `decolar()`. Ela também gerencia o início e o fim da contabilização do tempo de operação para as estatísticas.
- **Funções de Operação (`pousar()`, `desembarcar()`, `decolar()`):** São o coração da lógica de concorrência. Cada função é responsável por implementar a ordem de alocação de recursos específica para sua operação, diferenciando entre voos domésticos e internacionais. É dentro delas que a estratégia de prevenção de deadlock com `sem_timedwait` é aplicada. A função `pousar()` contém, adicionalmente, a lógica para monitorar o tempo de espera e ativar a política de prioridade dinâmica para voos domésticos em estado crítico.
- **Recursos Globais (Semáforos):** Os recursos físicos do aeroporto (pistas, portões e torre) são modelados como variáveis de semáforo globais (`sem_t`).

Eles são inicializados como semáforos contadores, permitindo que o número de "vagas"de cada recurso seja definido dinamicamente na inicialização do programa, atendendo ao requisito de flexibilidade para experimentação.

- **Métricas e Geração de Relatório:** Um conjunto de variáveis globais é utilizado para registrar as métricas da simulação (total de quedas, sucessos, alertas, etc.). O acesso a estas variáveis é protegido por um mutex (`mutexContadores`) para evitar condições de corrida. Ao final da execução, um bloco de código na função `main` formata e imprime essas métricas no terminal e em um arquivo de texto.

2.1.1 Função Principal e Orquestração da Simulação (`main()`)

A função `main` é o ponto de entrada do programa e atua como o principal orquestrador da simulação, sendo responsável por três fases distintas: inicialização, execução e finalização.

A fase de inicialização, ilustrada na Figura 1, é responsável por configurar todo o ambiente da simulação. Primeiramente, ela interpreta os argumentos de linha de comando para definir a quantidade de recursos disponíveis (pistas, portões e capacidade da torre). Em seguida, inicializa os mecanismos de concorrência: os semáforos que representam os recursos do aeroporto e os mutexes que protegem o acesso às variáveis de estatísticas. Por fim, aloca a memória necessária para as threads e as estruturas dos aviões, e inicia a thread do monitor de deadlock.

```c
int main(int argc, char *argv[]) {
    if (argc < 4) {
        fprintf(stderr, "Uso: %s <num_pistas> <num_portoes> <capacidade_torre>\n", argv[0]);
        return 1;
    }
    srand(time(NULL));
    int numPistas = atoi(argv[1]);
    int numPortoes = atoi(argv[2]);
    int capacidadeTorre = atoi(argv[3]);
    sem_init(&semPistas, 0, numPistas);
    sem_init(&semPortoes, 0, numPortoes);
    sem_init(&semTorre, 0, capacidadeTorre);
    pthread_mutex_init(&mutexContadores, NULL);
    pthread_mutex_init(&mutexOperacao, NULL);
    pthread_mutex_init(&mutexMonitor, NULL);
    pthread_t* threads = malloc(sizeof(pthread_t) * MAX_AVIOES);
    struct aviao* avioes = malloc(sizeof(struct aviao) * MAX_AVIOES);
    if (!threads || !avioes) { perror("malloc"); return 1; }
    for (int j = 0; j < MAX_AVIOES; j++) {
        pthread_mutex_init(&avioes[j].mutexAviao, NULL);
    }
    struct monitor_args margs;
    margs.avioes = avioes;
    margs.numAvioesCriados = 0;
    pthread_t monitorThread;
    pthread_create(&monitorThread, NULL, threadMonitor, &margs);
    time_t inicio = time(NULL);
    int i = 0;
    printf("Iniciando simulacao por %d segundos...\n", TEMPO_TOTAL);
```

Figura 1: Inicialização dos parâmetros, semáforos, mutexes e alocação de memória.

Após a configuração, a simulação entra em sua fase de execução, demonstrada na Figura 2. A função `main` entra em um laço principal que opera por um tempo pré-determinado, criando novas threads de avião em intervalos de tempo aleatórios para simular um fluxo realista de chegadas. Quando o tempo da simulação se esgota, o laço de criação é interrompido. A função então entra em uma fase de espera, utilizando `pthread_join` para garantir que todas as aeronaves em operação concluam seus ciclos de vida antes de prosseguir para a geração do relatório.

```c
fflush(stdout);
while (time(NULL) - inicio < TEMPO_TOTAL && i < MAX_AVIOES && simulacaoAtiva) {
    avioes[i].id = i + 1;
    avioes[i].tipo = rand() % 2;
    avioes[i].tempoDeOperacao = rand() % 3 + 1;
    avioes[i].entrouEmAlerta = 0;
    avioes[i].emEstadoCritico = 0;
    avioes[i].anunciouPrioridade = 0;
    pthread_mutex_lock(&mutexContadores);
    totalAvioes++;
    if (avioes[i].tipo == 0) totalDomesticos++;
    else totalInternacionais++;
    pthread_mutex_unlock(&mutexContadores);
    pthread_create(&threads[i], NULL, threadAviao, &avioes[i]);
    i++;
    margs.numAvioesCriados = i;
    sleep(rand() % 4);
}
printf("\n>>> TEMPO DE SIMULACAO ESGOTADO. Nao serao criados novos avioes. Aguardando operacoes em andamento... <<<\n\n"
fflush(stdout);
simulacaoAtiva = 0;
for (int j = 0; j < i; j++) {
    pthread_join(threads[j], NULL);
}
pthread_join(monitorThread, NULL);
```

Figura 2: Laço principal de criação de threads e espera pela sua finalização.

A fase final do programa é a geração de relatórios e a limpeza dos recursos. Como detalhado nas Figuras 3 e 4, a função primeiramente formata um nome de arquivo único para o relatório baseado nos parâmetros da simulação. Em seguida, todas as métricas coletadas durante a execução (sucessos, quedas, alertas, etc.) são impressas de forma organizada tanto no terminal quanto no arquivo de texto gerado.

```c
sprintf(nome_arquivo, "relatorio_%dp_%dg_%dt.txt", numPistas, numPortoes, capacidadeTorre);
FILE *arquivo_relatorio;
arquivo_relatorio = fopen(nome_arquivo, "w");
if (arquivo_relatorio == NULL) {
    fprintf(stderr, "Erro ao criar o arquivo de relatorio '%s'.\n", nome_arquivo);
}
printf("\n--- RELATORIO FINAL ---\n");
printf("Configuracao: %d Pistas, %d Portoes, %d Torre(s) de Controle\n", numPistas, numPortoes, capacidadeTorre);
printf("----------------------------------------\n");
printf("Total de avioes criados: %d\n", totalAvioes);
printf(" - Domesticos: %d\n", totalDomesticos);
printf(" - Internacionais: %d\n", totalInternacionais);
printf("----------------------------------------\n");
printf("Total de avioes que completaram o ciclo: %d\n", avioesSucesso);
printf("Total de avioes que cairam (starvation): %d\n", avioesCaidos);
printf("Total de avioes que entraram em alerta critico: %d\n", avioesAlerta);
printf("Deadlocks detectados pelo monitor: %d\n", deadlocksOcorridos);
printf("----------------------------------------\n");
printf("Tempo total de simulacao (sec): %ld\n", time(NULL) - inicio);
printf("Pico de avioes simultaneos em operacao: %d\n", picoAvioesSimultaneos);
if (arquivo_relatorio != NULL) {
    fprintf(arquivo_relatorio, "--- RELATORIO FINAL ---\n");
    fprintf(arquivo_relatorio, "Configuracao: %d Pistas, %d Portoes, %d Torre(s) de Controle\n", numPistas, numPortoes,
    fprintf(arquivo_relatorio, "----------------------------------------\n");
    fprintf(arquivo_relatorio, "Total de avioes criados: %d\n", totalAvioes);
    fprintf(arquivo_relatorio, " - Domesticos: %d\n", totalDomesticos);
    fprintf(arquivo_relatorio, " - Internacionais: %d\n", totalInternacionais);
    fprintf(arquivo_relatorio, "----------------------------------------\n");
    fprintf(arquivo_relatorio, "Total de avioes que completaram o ciclo: %d\n", avioesSucesso);
    fprintf(arquivo_relatorio, "Total de avioes que cairam (starvation): %d\n", avioesCaidos);
    fprintf(arquivo_relatorio, "Total de avioes que entraram em alerta critico: %d\n", avioesAlerta);
    fprintf(arquivo_relatorio, "Deadlocks detectados pelo monitor: %d\n", deadlocksOcorridos);
    fprintf(arquivo_relatorio, "----------------------------------------\n");
    fprintf(arquivo_relatorio, "Tempo total de simulacao (sec): %ld\n", time(NULL) - inicio);
    fprintf(arquivo_relatorio, "Pico de avioes simultaneos em operacao: %d\n", picoAvioesSimultaneos);
}
```

Figura 3: Geração do nome do arquivo e impressão do corpo principal do relatório.

Finalmente, após salvar os resultados, a função realiza a limpeza completa do ambiente. Este processo, visto na Figura 4, envolve a destruição de todos os mutexes e semáforos para liberar os recursos do sistema operacional, e a liberação de toda a memória alocada dinamicamente, garantindo que o programa termine de forma limpa e sem vazamentos de memória.

```c
    if (avioesSucesso > 0) {
        double media = somaTemposOperacao / (double)avioesSucesso;
        printf("Tempo medio de ciclo por aviao (sucesso): %.2f ms\n", media);
        if (arquivo_relatorio != NULL) {
            fprintf(arquivo_relatorio, "Tempo medio de ciclo por aviao (sucesso): %.2f ms\n", media);
        }
    } else {
        printf("Nenhum aviao completou as operacoes com sucesso.\n");
        if (arquivo_relatorio != NULL) {
            fprintf(arquivo_relatorio, "Nenhum aviao completou as operacoes com sucesso.\n");
        }
    }
    printf("--- FIM DO RELATORIO ---\n");
    if (arquivo_relatorio != NULL) {
        fprintf(arquivo_relatorio, "--- FIM DO RELATORIO ---\n");
        fclose(arquivo_relatorio);
        printf("\nRelatorio final tambem foi salvo em '%s'\n", nome_arquivo);
    }
    for (int j = 0; j < MAX_AVIOES; j++) pthread_mutex_destroy(&avioes[j].mutexAviao);
    free(threads);
    free(avioes);
    sem_destroy(&semPistas);
    sem_destroy(&semPortoes);
    sem_destroy(&semTorre);
    pthread_mutex_destroy(&mutexContadores);
    pthread_mutex_destroy(&mutexOperacao);
    pthread_mutex_destroy(&mutexMonitor);
    return 0;
}
```

Figura 4: Cálculo de métricas finais, fechamento do arquivo e liberação de todos os recursos.

2.1.2 Estrutura do Avião (`struct aviao`)

A estrutura `aviao`, implementada em C, é a representação central de uma aeronave dentro da simulação. Ela encapsula todos os dados de estado necessários para que uma thread execute seu ciclo de vida, tome decisões e seja monitorada. Cada thread de avião possui sua própria instância desta estrutura, garantindo o isolamento de seus dados. Os campos da estrutura são detalhados na Figura 5.

```c
struct aviao {
    int id;
    int tipo; // 0 doméstico, 1 internacional
    int tempoDeOperacao;
    int entrouEmAlerta;
    pthread_mutex_t mutexAviao;
    int emEstadoCritico;
    int anunciouPrioridade;
};
```

Figura 5: Campos da estrutura `aviao`.

A responsabilidade de cada campo é descrita a seguir:

- **`id` e `tipo`:** Campos de identificação básicos. O `id` é um inteiro único para cada avião, enquanto o `tipo` (0 para doméstico, 1 para internacional) determina a política de alocação de recursos que a thread seguirá, sendo o principal fator para a simulação dos cenários de prioridade e *starvation*.
- **`tempoDeOperacao`:** Armazena um valor aleatório que define por quanto tempo o avião utilizará os recursos (pouso, desembarque ou decolagem). Isso introduz variabilidade e realismo ao fluxo do sistema.
- **`mutexAviao`:** Um mutex individual para cada instância da estrutura. Embora a arquitetura atual minimize acessos concorrentes diretos ao estado de um avião, este mutex garante a atomicidade de futuras modificações, tornando a estrutura inerentemente *thread-safe*.
- **`entrouEmAlerta`, `emEstadoCritico` e `anunciouPrioridade`:** Este conjunto de flags gerencia o estado do avião em relação à política de mitigação de *starvation*.
  - `entrouEmAlerta` é ativada quando a espera por um recurso ultrapassa 60 segundos, servindo como um marcador para as estatísticas.
  - `emEstadoCritico` é a flag que efetivamente aciona a mudança na lógica de alocação, permitindo que um avião doméstico adote a estratégia prioritária para tentar evitar a queda.
  - `anunciouPrioridade` é uma flag de controle auxiliar, utilizada para garantir que a mensagem de tentativa de pouso prioritário seja impressa no log apenas uma vez, mantendo a saída do terminal limpa e legível.

2.1.3 Ciclo de Vida da Aeronave (`threadAviao`)

A função `threadAviao` representa a rotina de controle principal para cada thread de avião, definindo seu ciclo de vida completo desde a entrada no sistema até a sua saída. Conforme ilustrado na Figura 6, esta função orquestra a sequência de operações e gerencia o registro de métricas de sucesso ou falha.

```c
void *threadAviao(void *arg) {
    struct aviao *a = (struct aviao *)arg;
    struct timeval tstart, tend;
    gettimeofday(&tstart, NULL);
    pthread_mutex_lock(&mutexOperacao);
    avioesEmOperacao++;
    if (avioesEmOperacao > picoAvioesSimultaneos) picoAvioesSimultaneos = avioesEmOperacao;
    pthread_mutex_unlock(&mutexOperacao);
    if (pousar(a) != 0) {
        pthread_mutex_lock(&mutexOperacao);
        avioesEmOperacao--;
        pthread_mutex_unlock(&mutexOperacao);
        return NULL;
    }
    if (desembarcar(a) != 0) {
        pthread_mutex_lock(&mutexOperacao);
        avioesEmOperacao--;
        pthread_mutex_unlock(&mutexOperacao);
        return NULL;
    }
    if (decolar(a) != 0) {
        pthread_mutex_lock(&mutexOperacao);
        avioesEmOperacao--;
        pthread_mutex_unlock(&mutexOperacao);
        return NULL;
    }
    gettimeofday(&tend, NULL);
    long long tempo = (tend.tv_sec - tstart.tv_sec) * 1000LL + (tend.tv_usec - tstart.tv_usec) / 1000;
    pthread_mutex_lock(&mutexContadores);
    avioesSucesso++;
    somaTemposOperacao += tempo;
    pthread_mutex_unlock(&mutexContadores);
    pthread_mutex_lock(&mutexOperacao);
    avioesEmOperacao--;
    pthread_mutex_unlock(&mutexOperacao);
    return NULL;
}
```

Figura 6: Implementação da função `threadAviao`.

O fluxo de execução da thread pode ser dividido em quatro etapas principais:

1. **Entrada no Sistema:** Ao ser iniciada, a thread primeiro entra em uma seção crítica, protegida por um mutex, para incrementar o contador de aviões em operação (`avioesEmOperacao`) e atualizar a métrica de pico de voos simultâneos. Isso garante que a contagem de carga do sistema seja sempre consistente.

2. **Execução Sequencial das Operações:** A thread executa as três operações principais em uma sequência estrita: `pousar()`, `desembarcar()` e `decolar()`.

Após cada chamada, o valor de retorno da função é verificado. Se uma operação falhar (retornando um valor diferente de zero), a thread entra em um bloco de tratamento de falha: ela decrementa o contador de aviões em operação e encerra sua execução imediatamente com `return NULL;`. Essa estrutura garante que um avião que "caiu"durante o pouso, por exemplo, não prossiga para a fase de desembarque.

3. **Registro de Sucesso:** Apenas se todas as três operações forem concluídas com sucesso, a thread prossegue para a coleta de métricas de desempenho. O tempo total decorrido desde o início de sua execução é calculado. Em seguida, a thread entra em outra seção crítica para incrementar atomicamente o contador de sucessos (`avioesSucesso`) e somar seu tempo de ciclo ao total da simulação.
4. **Saída do Sistema:** Como etapa final, tanto em caso de sucesso quanto de falha (após a primeira etapa), a thread decrementa o contador `avioesEmOperacao` antes de terminar. Isso sinaliza que ela não está mais competindo por recursos do aeroporto.

Dessa forma, a função `threadAviao` não apenas impõe a ordem correta das operações, mas também garante um tratamento de falhas robusto e a coleta de dados estatísticos de forma segura em um ambiente concorrente.

2.1.4 Operação de Pousar(`pousar()`)

A função `pousar` é a mais complexa da simulação, pois é nela que ocorre o principal conflito por recursos e onde a política de mitigação de *starvation* é aplicada. A sua estrutura geral, como visto na Figura 7, consiste em um laço de repetição (`while`) que persiste até que a aeronave consiga alocar com sucesso uma pista e uma vaga na torre de controle.

```c
int pousar(struct aviao *a) {
    int obteveTorre = 0;
    int obtevePista = 0;
    if (!simulacaoAtiva) return -1;

    time_t inicioEspera = time(NULL);

    while (!(obtevePista && obteveTorre) && simulacaoAtiva) {

        if (a->tipo == 0) {
            int espera = time(NULL) - inicioEspera;
            if (espera >= 60 && !a->entrouEmAlerta) {
                pthread_mutex_lock(&mutexContadores);
                if (!a->entrouEmAlerta) {
                    a->entrouEmAlerta = 1;
                    avioesAlerta++;
                    a->emEstadoCritico = 1;
                    printf("Aviao %d (Domestico) em alerta critico (esperando ha %ds) [pouso]\n", a->id, espera);
                    fflush(stdout);
                }
                pthread_mutex_unlock(&mutexContadores);
            }
            if (espera >= 90) {
                pthread_mutex_lock(&mutexContadores);
                avioesCaidos++;
                pthread_mutex_unlock(&mutexContadores);
                printf("!!! Aviao %d (Domestico) CAIU apos 90s esperando por recursos para pouso.\n", a->id);
                fflush(stdout);
                if (obteveTorre) sem_post(&semTorre);
                if (obtevePista) sem_post(&semPistas);
                return -1;
            }
        }
```

Figura 7: Início da função `pousar` com a lógica de monitoramento de tempo para voos domésticos.

Dentro deste laço, para os voos domésticos, um cronômetro monitora o tempo de espera. Conforme o requisito do trabalho, se a espera atinge 60 segundos, o avião entra em "alerta crítico", e a flag `emEstadoCritico` é ativada. Se a espera chegar a 90 segundos, a simulação considera que o avião "caiu", a thread é encerrada e a falha é registrada nas estatísticas.

A alocação de recursos é dividida em estratégias distintas. Para voos internacionais, que possuem prioridade, a estratégia é sempre a mesma: tentar adquirir primeiro a pista e depois a torre (`Pista -> Torre`), como detalhado na Figura 8. Para evitar deadlocks, caso a thread não consiga o segundo recurso (torre) em um tempo limite, ela libera o primeiro (pista) e tenta novamente.

```c
// --- ESTRATÉGIA DE ALOCAÇÃO ---

// Estratégia para Internacionais (sempre prioritária)
if (a->tipo == 1) {
    if (!obtevePista) {
        if (sem_wait_seconds(&semPistas, 1) == 0) obtevePista = 1;
    }
    if (obtevePista && !obteveTorre) {
        if (sem_wait_seconds(&semTorre, 1) == 0) obteveTorre = 1;
        else {
            sem_post(&semPistas);
            obtevePista = 0;
        }
    }
}
```

Figura 8: Estratégia de alocação de recursos para voos internacionais na função `pousar`.

Para os voos domésticos, a estratégia é dinâmica, como mostra a Figura 9. Inicialmente, eles tentam alocar os recursos na ordem inversa (`Torre -> Pista`). No entanto, se o avião entrar no estado crítico (após 60 segundos de espera), sua estratégia é elevada: ele passa a tentar alocar os recursos na mesma ordem prioritária dos voos internacionais, aumentando sua chance de sucesso e evitando a queda.

```
    else {
        if (a->emEstadoCritico == 1) { // Lógica Prioritária para Doméstico Crítico
            if (a->anunciouPrioridade == 0) {
                printf("--- Aviao %d (Domestico CRITICO) tentando pouso prioritario ---\n", a->id);
                fflush(stdout);
                a->anunciouPrioridade = 1;
            }
            // Tenta pegar o que falta, usando a ordem Pista -> Torre
            if (!obtevePista) {
                if (sem_wait_seconds(&semPistas, 1) == 0) obtevePista = 1;
            }
            if (obtevePista && !obteveTorre) {
                if (sem_wait_seconds(&semTorre, 1) == 0) obteveTorre = 1;
                else {
                    sem_post(&semPistas);
                    obtevePista = 0;
                }
            }
        } else { // Lógica Padrão para Doméstico normal
            // Tenta pegar o que falta, usando a ordem Torre -> Pista
            if (!obteveTorre) {
                if (sem_wait_seconds(&semTorre, 1) == 0) obteveTorre = 1;
            }
            if (obteveTorre && !obtevePista) {
                if (sem_wait_seconds(&semPistas, 1) == 0) obtevePista = 1;
                else {
                    sem_post(&semTorre);
                    obteveTorre = 0;
                }
            }
        }
    }
}
```

Figura 9: Lógica de alocação dinâmica para voos domésticos, alternando entre a estratégia padrão e a prioritária.

Uma vez que ambos os recursos são adquiridos com sucesso, a thread sai do laço de espera, simula o tempo da operação de pouso e, ao final, libera os recursos, como ilustrado na Figura 10.

```
    if (!(obtevePista && obteveTorre)) {
        if (obtevePista) sem_post(&semPistas);
        if (obteveTorre) sem_post(&semTorre);
        return -1;
    }

    printf("Aviao %d (%s) iniciou o pouso.\n", a->id, a->tipo == 0 ? "Domestico" : "Internacional");
    fflush(stdout);
    sleep(a->tempoDeOperacao);
    printf("Aviao %d (%s) terminou o pouso.\n", a->id, a->tipo == 0 ? "Domestico" : "Internacional");
    fflush(stdout);

    sem_post(&semTorre);
    sem_post(&semPistas);
    return 0;
}
```

Figura 10: Finalização da operação de pouso com a simulação do uso e liberação dos recursos.

2.1.5 Operação de Desembarque (`desembarcar()`)

A função `desembarcar` gerencia a segunda fase do ciclo de vida da aeronave. Conforme os requisitos, esta operação necessita de um portão e uma vaga na torre de controle. A Figura 11 detalha a lógica de alocação de recursos, que também difere entre os tipos de voo para manter o potencial de conflito no sistema. Voos internacionais solicitam `Portão -> Torre`, enquanto voos domésticos solicitam `Torre -> Portão`.

```c
int desembarcar(struct aviao *a) {
    int obteveTorre = 0;
    int obtevePortao = 0;
    if (!simulacaoAtiva) return -1;

    if (a->tipo == 1) {
        while (!(obtevePortao && obteveTorre) && simulacaoAtiva) {
            if (!obtevePortao) {
                if (sem_wait_seconds(&semPortoes, 1) == 0) obtevePortao = 1;
            }
            if (obtevePortao && !obteveTorre) {
                if (sem_wait_seconds(&semTorre, 1) == 0) obteveTorre = 1;
                else {
                    sem_post(&semPortoes);
                    obtevePortao = 0;
                }
            }
        }
    } else {
        time_t inicio = time(NULL);
        while (!(obteveTorre && obtevePortao) && simulacaoAtiva) {
            if (!obteveTorre) {
                if (sem_wait_seconds(&semTorre, 1) == 0) obteveTorre = 1;
            }
            if (obteveTorre && !obtevePortao) {
                if (sem_wait_seconds(&semPortoes, 1) == 0) obtevePortao = 1;
                else {
                    sem_post(&semTorre);
                    obteveTorre = 0;
                }
            }
```

Figura 11: Lógica de alocação de recursos na função `desembarcar`.

Similarmente à função de pouso, a estratégia de espera com *timeout* e liberação de recursos é empregada para prevenir deadlocks. Para voos domésticos, um timer de 60 segundos também é implementado, mas ele apenas registra um alerta para fins estatísticos, sem levar a uma "queda", como visto na Figura 12. Após a alocação bem-sucedida, a operação é simulada e os recursos são liberados.

```c
            }
            int espera = time(NULL) - inicio;
            if (espera >= 60 && !a->entrouEmAlerta) {
                pthread_mutex_lock(&mutexContadores);
                if(!a->entrouEmAlerta){
                    a->entrouEmAlerta = 1;
                    avioesAlerta++;
                    printf("Aviao %d (Domestico) esperando para desembarcar (esperando ha %ds)\n", a->id, espera);
                    fflush(stdout);
                }
                pthread_mutex_unlock(&mutexContadores);
            }
        }
    }

    if (!(obtevePortao && obteveTorre)) {
        if (obtevePortao) sem_post(&semPortoes);
        if (obteveTorre) sem_post(&semTorre);
        return -1;
    }

    printf("Aviao %d (%s) iniciou o desembarque.\n", a->id, a->tipo == 0 ? "Domestico" : "Internacional");
    fflush(stdout);
    sleep(a->tempoDeOperacao);
    printf("Aviao %d (%s) terminou o desembarque.\n", a->id, a->tipo == 0 ? "Domestico" : "Internacional");
    fflush(stdout);

    sem_post(&semTorre);
    sem_post(&semPortoes);
    return 0;
}
```

Figura 12: Lógica de alerta e finalização da operação de desembarque.

2.1.6 Operação de Decolagem (`decolar()`)

A decolagem é a fase final e a que mais exige recursos, necessitando simultaneamente de um portão, uma pista e a torre de controle. A complexidade da alocação de três recursos aumenta a chance de contenção. A Figura 13 mostra a implementação da estratégia de alocação para voos internacionais, que seguem a ordem `Portão -> Pista -> Torre`.

```c
int decolar(struct aviao *a) {
    int obteveTorre = 0;
    int obtevePista = 0;
    int obtevePortao = 0;
    if (!simulacaoAtiva) return -1;

    if (a->tipo == 1) {
        while (!(obtevePortao && obtevePista && obteveTorre) && simulacaoAtiva) {
            if (!obtevePortao) {
                if (sem_wait_seconds(&semPortoes, 1) == 0) obtevePortao = 1;
            }
            if (obtevePortao && !obtevePista) {
                if (sem_wait_seconds(&semPistas, 1) == 0) obtevePista = 1;
                else {
                    sem_post(&semPortoes);
                    obtevePortao = 0;
                }
            }
            if (obtevePortao && obtevePista && !obteveTorre) {
                if (sem_wait_seconds(&semTorre, 1) == 0) obteveTorre = 1;
                else {
                    sem_post(&semPistas);
                    sem_post(&semPortoes);
                    obtevePista = 0;
                    obtevePortao = 0;
                }
            }
        }
```

Figura 13: Estratégia de alocação de recursos para voos internacionais na função `decolar`.

Os voos domésticos, por sua vez, seguem a ordem inversa, `Torre -> Portão -> Pista`, e também possuem um monitor de alerta para longas esperas, como detalhado na Figura 14. A mesma técnica de prevenção de deadlock é utilizada, onde a falha na aquisição de qualquer recurso na sequência acarreta na liberação de todos os recursos já adquiridos.

```
} else {
    time_t inicio = time(NULL);
    while (!(obteveTorre && obtevePortao && obtevePista) && simulacaoAtiva) {
        if (!obteveTorre) {
            if (sem_wait_seconds(&semTorre, 1) == 0) obteveTorre = 1;
        }
        if (obteveTorre && !obtevePortao) {
            if (sem_wait_seconds(&semPortoes, 1) == 0) obtevePortao = 1;
            else {
                sem_post(&semTorre);
                obteveTorre = 0;
            }
        }
        if (obteveTorre && obtevePortao && !obtevePista) {
            if (sem_wait_seconds(&semPistas, 1) == 0) obtevePista = 1;
            else {
                sem_post(&semPortoes);
                sem_post(&semTorre);
                obtevePortao = 0;
                obteveTorre = 0;
            }
        }
        int espera = time(NULL) - inicio;
        if (espera >= 60 && !a->entrouEmAlerta) {
            pthread_mutex_lock(&mutexContadores);
            if(!a->entrouEmAlerta){
               a->entrouEmAlerta = 1;
               aviosAlerta++;
               printf("Aviao %d (Domestico) esperando para decolar (esperando ha %ds)\n", a->id, espera);
               fflush(stdout);
            }
            pthread_mutex_unlock(&mutexContadores);
        }
    }
}
```

Figura 14: Estratégia de alocação e lógica de alerta para voos domésticos na decolagem.

Ao obter com sucesso os três recursos, a aeronave simula o tempo da operação de decolagem e, por fim, libera todos os recursos para que possam ser utilizados por outras aeronaves no sistema, conforme ilustra a Figura 15.

```c
    if (!(obtevePortao && obtevePista && obteveTorre)) {
        if (obtevePortao) sem_post(&semPortoes);
        if (obtevePista) sem_post(&semPistas);
        if (obteveTorre) sem_post(&semTorre);
        return -1;
    }

    printf("Aviao %d (%s) iniciou a decolagem.\n", a->id, a->tipo == 0 ? "Domestico" : "Internacional");
    fflush(stdout);
    sleep(a->tempoDeOperacao);
    printf("Aviao %d (%s) terminou a decolagem.\n", a->id, a->tipo == 0 ? "Domestico" : "Internacional");
    fflush(stdout);

    sem_post(&semTorre);
    sem_post(&semPistas);
    sem_post(&semPortoes);
    return 0;
}
```

Figura 15: Finalização da operação de decolagem.

2.2 Prevenção de Deadlock

Uma das principais preocupações no desenvolvimento de sistemas concorrentes é a ocorrência de *deadlock* (impasse), uma condição na qual duas ou mais threads ficam permanentemente bloqueadas, cada uma esperando por um recurso que a outra detém. Para este trabalho, a prevenção de deadlock foi um requisito central. A estratégia adotada foi a de quebrar a condição de **"Reter e Esperar"** (*Hold-and-Wait*), uma das quatro condições necessárias de Coffman para a ocorrência de um deadlock.

A condição de "Reter e Esperar"ocorre quando uma thread já detém pelo menos um recurso e solicita recursos adicionais que estão, no momento, alocados para outras threads. Para quebrar essa condição, foi implementada uma política de "tentativa e recuo"(*release-and-retry*). Em vez de uma thread usar a chamada bloqueante `sem_wait()`, que esperaria indefinidamente, foi utilizada a função POSIX `sem_timedwait()`, que permite esperar por um recurso por um período de tempo limitado.

Para simplificar o uso desta função, que exige o cálculo de um *timestamp* absoluto no futuro, foi criada a função auxiliar `sem_wait_seconds`, como ilustrado na Figura 16.

```c
// Função auxiliar para esperar um semáforo com timeout em segundos
int sem_wait_seconds(sem_t *s, int seconds) {
    struct timespec ts;
    if (clock_gettime(CLOCK_REALTIME, &ts) == -1) return -1;
    ts.tv_sec += seconds;
    while (1) {
        int r = sem_timedwait(s, &ts);
        if (r == 0) return 0;
        if (errno == EINTR) continue;
        return -1;
    }
}
```

Figura 16: Função auxiliar para encapsular a chamada `sem_timedwait`.

Esta função recebe um semáforo e um tempo relativo em segundos, calcula o tempo absoluto de expiração e chama `sem_timedwait`. Se o recurso for adquirido com sucesso, ela retorna 0. Caso o tempo se esgote (ou ocorra outro erro), ela retorna -1.

A lógica de alocação nas funções de operação, como a `pousar()`, utiliza esta função auxiliar da seguinte maneira: uma thread tenta adquirir seu primeiro recurso. Se bem-sucedida, ela chama `sem_wait_seconds` para tentar adquirir o segundo recurso. Se esta chamada falhar (retornar -1), indicando um *timeout*, a thread imediatamente libera o primeiro recurso que havia adquirido, através de `sem_post()`, e reinicia seu ciclo de tentativa.

Essa abordagem garante que nenhuma thread permaneça bloqueada segurando um recurso enquanto espera por outro. Ao liberar o que já possui, ela permite que outras threads progridam, desfazendo a condição de espera circular que levaria ao deadlock. A eficácia desta estratégia é comprovada pelos resultados da simulação, que não registraram **nenhum deadlock** mesmo nos cenários de mais alta contenção de recursos.

2.3 Tratamento de Starvation

O segundo grande desafio de concorrência abordado neste trabalho é o *starvation* (fome), uma condição na qual uma thread é perpetuamente impedida de acessar os recursos de que necessita. No contexto desta simulação, o problema foi desenhado para que os voos domésticos, por possuírem uma estratégia de alocação de menor prioridade, fossem as vítimas potenciais de *starvation*. A abordagem para lidar com este problema foi dividida em duas etapas principais: detecção e mitigação.

A detecção do problema é realizada através de um cronômetro individual para cada

voo doméstico na crítica fase de pouso. Como ilustrado na Figura 17, a cada iteração do laço de espera por recursos, o tempo decorrido é verificado. Se a espera atingir 60 segundos, a thread é considerada vítima de *starvation* e entra em "estado de alerta crítico", ativando a flag `emEstadoCritico`. Se, mesmo assim, a thread não conseguir os recursos e a espera atingir 90 segundos, uma falha operacional é simulada: o avião "cai", a thread é encerrada e o evento é contabilizado nas estatísticas.

```
while (!(obtevePista && obteveTorre) && simulacaoAtiva) {

    if (a->tipo == 0) {
        int espera = time(NULL) - inicioEspera;
        if (espera >= 60 && !a->entrouEmAlerta) {
            pthread_mutex_lock(&mutexContadores);
            if (!a->entrouEmAlerta) {
                a->entrouEmAlerta = 1;
                avioesAlerta++;
                a->emEstadoCritico = 1;
                printf("Aviao %d (Domestico) em alerta critico (esperando ha %ds) [pouso]\n", a->id, espera);
                fflush(stdout);
            }
            pthread_mutex_unlock(&mutexContadores);
        }
        if (espera >= 90) {
            pthread_mutex_lock(&mutexContadores);
            avioesCaidos++;
            pthread_mutex_unlock(&mutexContadores);
            printf("!!! Aviao %d (Domestico) CAIU apos 90s esperando por recursos para pouso.\n", a->id);
            fflush(stdout);
            if (obteveTorre) sem_post(&semTorre);
            if (obtevePista) sem_post(&semPistas);
            return -1;
        }
    }
```

Figura 17: Lógica de detecção de starvation com timers de 60s (alerta) e 90s (queda).

Para cumprir o requisito de tentar evitar as quedas, uma política de mitigação dinâmica foi implementada, conforme detalhado na Figura 18. Uma vez que a flag `emEstadoCritico` de um avião doméstico é ativada, sua lógica de alocação de recursos é alterada. Ele abandona sua estratégia padrão (`Torre -> Pista`) e passa a executar a estratégia prioritária, normalmente reservada aos voos internacionais (`Pista -> Torre`).

```
// Estratégia para Domésticos (muda se ficar crítico)
else {
    if (a->emEstadoCritico == 1) { // Lógica Prioritária para Doméstico Crítico
        if (a->anunciouPrioridade == 0) {
            printf("--- Aviao %d (Domestico CRITICO) tentando pouso prioritario ---\n", a->id);
            fflush(stdout);
            a->anunciouPrioridade = 1;
        }
        // Tenta pegar o que falta, usando a ordem Pista -> Torre
        if (!obtevePista) {
            if (sem_wait_seconds(&semPistas, 1) == 0) obtevePista = 1;
        }
        if (obtevePista && !obteveTorre) {
            if (sem_wait_seconds(&semTorre, 1) == 0) obteveTorre = 1;
            else {
                sem_post(&semPistas);
                obtevePista = 0;
            }
        }
```

Figura 18: Implementação da política de mitigação, onde o avião doméstico crítico adota a estratégia de alocação prioritária.

Essa elevação de prioridade dinâmica dá à thread uma chance significativamente maior de adquirir os recursos necessários e concluir sua operação de pouso com sucesso, prevenindo a falha. Essa abordagem dupla permite que a simulação não apenas evidencie a ocorrência de *starvation*, mas também teste ativamente uma estratégia para combatê-lo, oferecendo uma análise mais rica sobre o gerenciamento de justiça (*fairness*) em sistemas concorrentes.

3 Passos para Compilar e Executar

Para que a simulação seja compilada e executada, é necessário um ambiente de desenvolvimento C que forneça o compilador GCC e suporte à biblioteca PThreads. O desenvolvimento deste projeto foi realizado em um sistema operacional Windows, utilizando o ambiente MinGW (Minimalist GNU for Windows), que oferece o compilador e uma camada de compatibilidade para a biblioteca PThreads. Alternativamente, a compilação é nativamente suportada em sistemas operacionais baseados em Linux ou no Windows Subsystem for Linux (WSL).

3.1 Compilação

O código-fonte, contido no arquivo `Trafego_Aereo.c`, deve ser compilado utilizando o comando a seguir. A flag `-lpthread` é essencial para vincular a biblioteca PThreads ao executável final.

gcc `Trafego_Aereo.c` -o simulador -lpthread

Este comando gerará um arquivo executável chamado `simulador`.

3.2 Execução

Após a compilação bem-sucedida, o programa pode ser executado através do terminal (como o PowerShell ou CMD no Windows). O simulador requer três argumentos de linha de comando para definir a configuração do aeroporto, na seguinte ordem:

1. Número de pistas disponíveis.
2. Número de portões de embarque.
3. Capacidade da torre de controle (operações simultâneas).

Por exemplo, para executar a simulação com a configuração base descrita no trabalho (3 pistas, 5 portões e capacidade 2 na torre), o comando é:

./simulador 3 5 2

Ao ser executado, o programa exibirá um log em tempo real das operações no terminal e, ao final da simulação, salvará um relatório completo em um arquivo de texto com nome dinâmico (ex: `relatorio_3p_5g_2t.txt`).

4 Resultados das Simulações

4.1 Cenário de Estresse: Configuração Base (3/5/2)

Para analisar o comportamento do sistema sob alta carga, uma simulação de 5 minutos (300 segundos) foi executada utilizando a configuração de recursos base: 3 pistas, 5 portões e capacidade 2 na torre de controle. O objetivo deste teste foi verificar a eficácia das políticas de prevenção de deadlock e mitigação de starvation em um cenário de saturação.

Os resultados, apresentados no relatório final da Figura 19, indicam um colapso do sistema por excesso de demanda. De um total de 194 aeronaves criadas, 32 (aproximadamente 16%) "caíram"devido à *starvation*, por não conseguirem alocar recursos de pouso em 90 segundos. O pico de 151 aeronaves em operação simultânea competindo por um conjunto de apenas 10 recursos principais (3 pistas + 5 portões + 2 vagas na torre) confirma o estado de saturação extrema. É notável, no entanto, que mesmo neste cenário caótico, nenhum deadlock foi detectado, comprovando a robustez da estratégia de prevenção implementada.

```
Aviao 129 (Internacional) iniciou o desembarque.
!!! Aviao 137 (Domestico) CAIU apos 90s esperando por recursos para pouso.
Aviao 143 (Internacional) terminou o pouso.
Aviao 154 (Domestico) esperando para desembarcar (esperando ha 60s)
Aviao 129 (Internacional) terminou o desembarque.
Aviao 40 (Internacional) iniciou o desembarque.
Aviao 156 (Domestico) em alerta critico (esperando ha 60s) [pouso]
--- Aviao 156 (Domestico CRITICO) tentando pouso prioritario ---
Aviao 40 (Internacional) terminou o desembarque.
Aviao 157 (Domestico) em alerta critico (esperando ha 60s) [pouso]
--- Aviao 157 (Domestico CRITICO) tentando pouso prioritario ---
Aviao 182 (Domestico) iniciou o pouso.
Aviao 160 (Domestico) em alerta critico (esperando ha 60s) [pouso]
--- Aviao 160 (Domestico CRITICO) tentando pouso prioritario ---
Aviao 182 (Domestico) terminou o pouso.

>>> TEMPO DE SIMULACAO ESGOTADO. Nao serao criados novos avioes. Aguardando operacoes em andamento... <<<

Aviao 167 (Internacional) iniciou o pouso.
Aviao 106 (Domestico) iniciou a decolagem.
Aviao 106 (Domestico) terminou a decolagem.
Aviao 50 (Internacional) iniciou o desembarque.
Aviao 50 (Internacional) terminou o desembarque.
Aviao 167 (Internacional) terminou o pouso.

--- RELATORIO FINAL ---
Configuracao: 3 Pistas, 5 Portoes, 2 Torre(s) de Controle
----------------------------------------
Total de avioes criados: 194
 - Domesticos: 92
 - Internacionais: 102
Total de avioes que cairam (starvation): 32
Total de avioes que entraram em alerta critico: 68
Deadlocks detectados pelo monitor: 0
----------------------------------------
Tempo total de simulacao (sec): 306
Pico de avioes simultaneos em operacao: 151
Tempo medio de ciclo por aviao (sucesso): 35702.33 ms
--- FIM DO RELATORIO ---

Relatorio final tambem foi salvo em 'relatorio_3p_5g_2t.txt'
```

Figura 19: Relatório final da simulação de 5 minutos sob estresse (3 Pistas, 5 Portões, 2 Torres).

Apesar do grande número de falhas, a análise detalhada do log de execução revela que a política de mitigação de *starvation* para aviões em estado crítico estava funcionando como o esperado. A Figura 20 apresenta um trecho do log que ilustra este comportamento de forma clara.

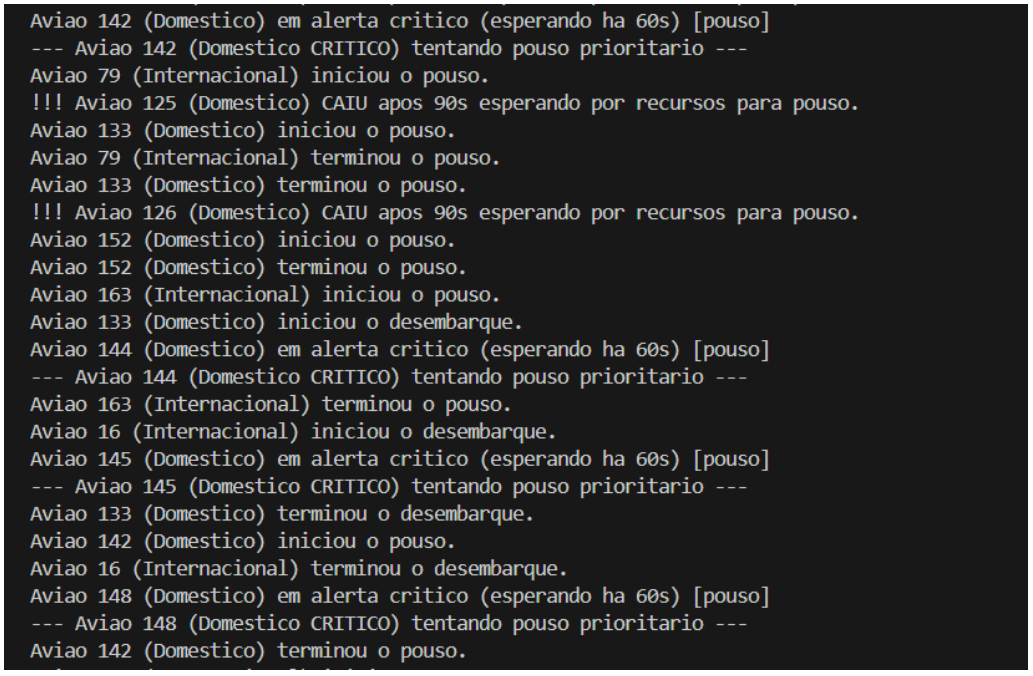

Figura 20: Trecho do log de execução mostrando a política de prioridade em ação.

É possível observar o caso do Avião 142 (Doméstico):

1. Primeiramente, ele entra em alerta crítico após 60 segundos de espera pelo pouso.
2. Em seguida, o sistema eleva sua prioridade, e a mensagem `--- Aviao 142 (Domestico CRITICO) tentando pouso prioritario ---` é exibida.
3. Após essa mudança de estratégia, o Avião 142 consegue adquirir os recursos necessários e finaliza seu pouso com sucesso (`Aviao 142 (Domestico) terminou o pouso`), sendo efetivamente salvo da queda.

No entanto, o mesmo trecho do log também mostra os aviões 125 e 126 caindo, evidenciando que, em um cenário de saturação extrema, a política de mitigação ajuda, mas não pode garantir a salvação de todas as threads. Isso confirma que a configuração base de recursos é insuficiente para a demanda, levando a um colapso por *starvation* que nem mesmo uma política de prioridade dinâmica consegue resolver por completo.

4.2 Cenário Otimizado: Reforço na Torre de Controle (3/5/4)

Com base na análise do cenário de estresse, a hipótese levantada foi de que a torre de controle, por ser um recurso requisitado em todas as fases do ciclo de vida da aeronave, representava o principal gargalo do sistema. Para validar esta hipótese, uma nova simulação de 5 minutos foi executada, mantendo o número de pistas e portões, mas dobrando a capacidade de atendimento simultâneo da torre de controle para 4.

Os resultados, apresentados na Figura 21, demonstram uma melhora drástica e imediata em todas as métricas de desempenho do aeroporto.

```
--- RELATORIO FINAL ---
Configuracao: 3 Pistas, 5 Portoes, 4 Torre(s) de Controle
----------------------------------------
Total de avioes criados: 194
 - Domesticos: 105
 - Internacionais: 89
----------------------------------------
Total de avioes que completaram o ciclo: 47
Total de avioes que cairam (starvation): 14
Total de avioes que entraram em alerta critico: 56
Deadlocks detectados pelo monitor: 0
----------------------------------------
Tempo total de simulacao (sec): 304
Pico de avioes simultaneos em operacao: 136
Tempo medio de ciclo por aviao (sucesso): 55838.34 ms
--- FIM DO RELATORIO ---
```

Figura 21: Relatório final da simulação com a torre de controle reforçada (3 Pistas, 5 Portões, 4 Torres).

O impacto mais significativo foi na redução do número de quedas por *starvation*, que diminuiu em mais de 56%, caindo de 32 para 14 falhas operacionais. Com mais vagas disponíveis na torre, as filas de espera foram reduzidas, permitindo que a política de mitigação para aviões em estado crítico se tornasse muito mais eficaz.

Além disso, a vazão do sistema, medida pelo número de ciclos completos, aumentou

mais de 230%, saltando de um número baixo na simulação anterior para 47 voos bem-sucedidos. O número de alertas críticos e o pico de aviões simultâneos também apresentaram redução, indicando um sistema mais fluido e menos congestionado.

Esta comparação de resultados valida a hipótese inicial: a torre de controle era, de fato, o principal gargalo de recursos do sistema. Dobrar sua capacidade de atendimento foi a intervenção mais eficaz para aumentar a estabilidade e a eficiência do aeroporto, mesmo sob uma demanda contínua e elevada.

5 Dificuldades Encontradas

Durante a implementação da simulação do tráfego aéreo, diversos desafios técnicos foram encontrados, especialmente relacionados à natureza complexa da programação concorrente. As principais dificuldades e as soluções aplicadas são detalhadas a seguir.

1. **Compreensão dos Mecanismos de Sincronização:** Um desafio inicial foi a compreensão aprofundada dos semáforos POSIX e sua aplicação correta no problema. A distinção entre um semáforo contador e um binário (mutex) foi crucial para modelar corretamente a torre de controle. O entendimento de que a torre, apesar de ser uma entidade única, possuía múltiplas capacidades de atendimento simultâneo, levou à escolha acertada do semáforo contador, inicializado com a capacidade especificada como argumento do programa.
2. **Depuração de Código Concorrente:** A depuração de múltiplas threads se mostrou uma tarefa particularmente complexa devido à sua natureza não-determinística. Bugs relacionados a condições de corrida ou impasses nem sempre ocorriam em todas as execuções, dificultando a identificação da causa raiz. Um problema prático derivado disso foi a inconsistência na exibição de logs no terminal. Muitas vezes, mensagens de status importantes, como a mudança para o estado prioritário, pareciam estar ausentes. A causa foi identificada como o buffer de saída do `printf`, e a solução foi forçar a descarga deste buffer com `fflush(stdout)` após cada impressão, garantindo um feedback visual confiável e em tempo real do estado da simulação.
3. **Implementação de uma Política de Prioridade Robusta:** O desenvolvimento de uma política para mitigar o *starvation* sem introduzir novos problemas foi o desafio mais significativo. Uma versão inicial da política de prioridade dinâmica, na qual um avião doméstico em estado crítico mudava sua estratégia de alocação de recursos, inadvertidamente causava um novo tipo de deadlock. A análise minuciosa dos logs de execução revelou que a thread alterava sua intenção de alocação, mas não liberava o recurso que já possuía

(a torre), mantendo a condição de espera circular que levava ao impasse. A solução foi refinar a lógica para que a thread em estado crítico liberasse os recursos previamente adquiridos antes de tentar novamente com a nova estratégia prioritária, resolvendo o deadlock e tornando a política de mitigação eficaz.

6 Conclusão e Trabalhos Futuros

Este trabalho logrou êxito em desenvolver um simulador concorrente de tráfego aéreo, que validou na prática os conceitos teóricos de Sistemas Operacionais. A simulação demonstrou com sucesso como a disputa por recursos limitados, combinada a diferentes estratégias de alocação, pode levar a condições críticas de *deadlock* e *starvation*. A implementação de uma política de prioridade dinâmica e de uma estratégia de prevenção de deadlock permitiu uma execução segura em ambiente multithread, mesmo sob alta carga.

A principal conclusão extraída dos experimentos é dupla. Primeiramente, a estratégia de prevenção de deadlock baseada em espera com *timeout* (`sem_timedwait`) mostrou-se totalmente robusta, não registrando nenhum impasse mesmo nos cenários de maior estresse e colapso do sistema. Em segundo lugar, a análise dos diferentes cenários demonstrou que, embora políticas de escalonamento dinâmico, como a elevação de prioridade para voos críticos, possam mitigar o *starvation*, a solução mais eficaz para um sistema saturado é o reforço dos recursos gargalos. A simulação comprovou que o aumento da capacidade da torre de controle teve um impacto significativamente maior na redução de falhas operacionais e no aumento da vazão do aeroporto.

Para trabalhos futuros, diversas expansões são possíveis. A sugestão mais imediata seria a implementação de uma interface gráfica, que permitiria uma melhor visualização do fluxo de aeronaves e da utilização dos recursos em tempo real, facilitando a análise dos resultados. Outra melhoria seria testar políticas de escalonamento mais complexas para evitar o *starvation*, como o "envelhecimento"(*aging*), onde a prioridade de uma thread aumenta progressivamente com seu tempo de espera. Por fim, a simulação poderia ser expandida para incluir mais realismo, com eventos aleatórios como fechamento de pistas por mau tempo, ou até explorar a integração com algoritmos de aprendizado de máquina para otimizar a alocação de recursos com base em padrões de demanda.