Trabalho 1

Luana Pinho Bueno Penha - 2312082
Theo Canuto - 2311293

Sistemas Operacionais - Turma 3WB

Prof. Markus Endler

08 de Maio de 2025, Rio de Janeiro

Funcionamento geral do programa

O programa simula um sistema de controle de tráfego aéreo usando processos concorrentes em C. Ele simula uma torre de controle, controlando em quais pistas cada aeronave pode pousar e quais ações elas devem tomar caso tenham risco de colidir entre si.

Cada aeronave é implementada como um processo filho independente, criado a partir do processo principal. Esses processos compartilham dados por meio de uma memória compartilhada (utilizando shmget para alocar a memória e shmat para acessar a memória), onde é mantido um vetor de structs do tipo Aircraft, uma para cada aeronave em voo.

Struct Aircraft:

A Struct Aircraft é onde definimos as características de uma aeronave, com cada uma contendo:

- Um pid próprio
- dir_x: representando a direção no eixo x. Se a aeronave estiver vindo do oeste, determinamos seu x inicial como 1, e se estiver vindo do leste, determinamos seu valor inicial como -1. Utilizamos os valores 1 e -1 pois esse valor será multiplicado depois com o speed_x e, dependendo do valor 1 ou -1, isso dita se a aeronave irá aumentar ou diminuir no eixo x.
- Coordenadas x e y: representando onde a aeronave começa, com x = 1 caso venha do leste e x = 0 caso venha do oeste.
- delay: delay de 0 a 2 segundos gerado randomicamente.
- landing: pista onde a aeronave pretende pousar. Essa informação é importante para calcularmos possíveis colisões e tratar elas, mudando o aeronave de pista caso tenha uma ameaça de colisão.
- speed x: que começa com 0.05 como valor default
- speed_y: representa a velocidade da aeronave no eixo y. Esse valor é calculado com base na distância entre a posição inicial do aeronave no eixo y e o alvo (COORD_TARGET), ajustado proporcionalmente à velocidade no eixo x (speed_x). Isso é feito para garantir que a aeronave siga uma trajetória retilínea em direção ao ponto de pouso, formando um triângulo retângulo onde a hipotenusa é a trajetória da aeronave.
- status: Pode ser de quatro tipos diferentes: **1 - Landed**, representando as aeronaves que já conseguiram pousar; **2 - Dead**, representando as aeronaves que estão mortas (por causa de um tratamento de colisão ou por causa de um comando do usuário para matar as aeronaves); **3 - Paused,** representando as aeronaves que estão parados devido a um tratamento de colisão ou após um comando do usuário de pausar todas elas; **4 - Flying,** representando as aeronaves que estão voando.

O controle de execução segue uma política de escalonamento Round Robin, o que permite que todas as aeronaves tenham um tempo para executar seus processos, respeitando uma ordem cíclica e um tempo compartilhado.

O processo pai acessa todos os processos filhos, verifica quais suas posições (x,y) e qual sua pista de destino e faz um tratamento de colisão caso duas aeronaves estejam muito próximas e indo para a mesma pista.

Sinais

Implementamos o uso de sinais para realizar o tratamento de colisões e também caso o usuário queira realizar uma ação geral (como, por exemplo, pausar todas as aeronaves).

Utilizamos:

- **SIGUSR1**:

O sinal SIGUSR1 é utilizado de duas formas distintas no programa: tanto pelo processo pai para pausar e retomar as aeronaves, quanto internamente pelas próprias aeronaves para realizar a transição entre os estados FLYING e PAUSED.

Quando um processo filho (aeronave) recebe o sinal SIGUSR1, ele executa uma função (`sig1_handler`) que verifica seu estado atual. Se estiver voando (FLYING), a aeronave entra em estado PAUSED, interrompendo seu movimento até que receba um novo sinal. Caso já esteja em PAUSED, ao receber novamente o SIGUSR1, ele retorna ao estado FLYING e retoma seu deslocamento normalmente.

- **SIGUSR2**:

O sinal SIGUSR2 é utilizado exclusivamente pelo processo pai como parte da lógica de tratamento de colisões. Quando o pai detecta que duas aeronaves estão muito próximas e se dirigem para a mesma pista de pouso, ele toma uma ação preventiva para evitar uma colisão iminente. Para isso, ele envia o sinal SIGUSR2 para uma das duas aeronaves envolvidas (geralmente a que está mais distante ou aquela que ainda pode alterar sua trajetória).

Ao receber o sinal SIGUSR2, o processo filho executa uma função (`sig2_handler`) que altera sua pista de pouso (na variável landing da struct aircraft) para outra pista livre, previamente determinada dependendo se a aeronave está vindo do lado leste ou do lado oeste. Essa mudança é registrada no sistema, e a aeronave continua sua trajetória, agora em direção à nova pista.

- **SIGTSTP:**

O programa utiliza o sinal **SIGTSTP** (enviado pelo atalho Ctrl + Z pelo usuário) como forma de pausar e retomar a execução de todas as aeronaves controladas pelo sistema. Esse sinal é associado à função `pause_resume_handler`, que é chamada automaticamente sempre que o sinal é recebido pelo processo pai.

A função `pause_resume_handler` verifica se o sistema já está pausado ou não. Se o sistema não estiver pausado, a função envia o sinal **SIGSTOP** para todas as aeronaves que estão no estado FLYING ou PAUSED, interrompendo temporariamente sua execução. Em seguida, a variável global paused é definida como true, indicando que o sistema está pausado. Caso o sistema já esteja pausado, a função envia o sinal **SIGCONT** para todas as aeronaves, permitindo que elas retomem sua execução. Após isso, a variável paused é redefinida como

false. Esse mecanismo garante que o usuário tenha controle total sobre a execução das aeronaves, permitindo pausar e retomar o sistema de forma eficiente e sincronizada.

- **SIGQUIT**:

  O programa utiliza o sinal **SIGQUIT** (enviado pelo atalho Ctrl + \ pelo usuário) como forma de finalizar o programa. Esse sinal é associado à função `stop_program_handler`, que é chamada automaticamente sempre que o sinal é recebido pelo processo pai e da um exit(0), finalizando o programa.

- **SIGINT**:

  O programa utiliza o sinal **SIGINT** (enviado pelo atalho Ctrl + C pelo usuário) como forma de finalizar o programa. Esse sinal é associado à função `kill_handler`, que é chamada automaticamente sempre que o sinal é recebido pelo processo pai e, para cada aeronave que não tenha pousado, muda seu estado para (DEAD) e envia um **SIGKILL** no seu processo.

Lógica do Tratamento de Colisões

O tratamento de colisões feito na função `treat_collisions` visa garantir que duas aeronaves que se aproximam de uma mesma pista não colidam. A detecção e a prevenção de colisões são realizadas com base nas posições dos aeronaves e nas pistas de pouso desejadas.

A detecção de colisões ocorre de forma contínua, verificando se duas aeronaves estão próximas e indo para a mesma pista a cada vez que uma aeronave se move. Para isso, o sistema analisa as coordenadas x e y de cada aeronave e a sua pista de destino (landing). Caso duas delas tenham a mesma pista de pouso e suas posições estejam muito próximas, o sistema identifica uma colisão iminente.

- Tratamento de Colisões com **SIGUSR2**

O sinal SIGUSR2 é utilizado como a primeira tentativa de evitar colisões entre aeronaves que estão se aproximando da mesma pista. Quando o sistema detecta que duas aeronaves estão muito próximas e podem colidir, ele envia o SIGUSR2 para uma das delas, solicitando que ela altere sua pista de pouso para uma alternativa no mesmo lado (leste ou oeste).

O tratamento com SIGUSR2 funciona da seguinte forma:

1. Após `detect_collisions`: O sistema identifica que duas aeronaves estão muito próximas e que ambas estão direcionadas para a mesma pista de pouso.
2. `kill(ac[index].pid, SIGUSR2`: O processo pai envia o sinal SIGUSR2 para uma das aeronaves envolvidas na colisão iminente.
3. Alteração de Pista: A aeronave que recebe o SIGUSR2 altera sua pista de pouso para uma alternativa disponível no mesmo lado. Por exemplo:
   a. Se a pista inicial for 27, ela muda para 6.
   b. Se a pista inicial for 18, ela muda para 3.

4. Verificação de Sucesso com a função `collision_risk`: Após a alteração, o sistema verifica se a mudança de pista é suficiente. Caso essa mudança gere colisão com alguma outra aeronave que está na nova pista, o sistema reverte a mudança de pista e utiliza outras estratégias, como o uso do **SIGUSR1,** para pausar a aeronave.

Esse mecanismo permite que o sistema reaja dinamicamente a situações de risco, ajustando as trajetórias das aeronaves para evitar colisões sem interromper a simulação. O uso do **SIGUSR2** é eficiente para resolver conflitos de pista de forma rápida e segura, mantendo a ordem no tráfego aéreo simulado, por isso foi escolhido para o tratamento inicial.

- Tratamento de Colisões com **SIGUSR1**

Caso a mudança de pista com o **SIGUSR2** não seja suficiente para evitar a colisão, o sistema utiliza o sinal **SIGUSR1** como uma segunda medida de segurança. O **SIGUSR1** é enviado para pausar temporariamente uma das aeronaves envolvidas na colisão iminente. Essa pausa permite que a outra aeronave prossiga com segurança até que as posições estejam suficientemente distantes para evitar o risco de colisão.

O tratamento com SIGUSR1 funciona da seguinte forma:

1. `kill(ac[index].pid, SIGUSR1`: O processo pai envia o sinal **SIGUSR1** para a aeronave que deve ser pausada. Isso altera o status da aeronave para PAUSED, interrompendo temporariamente seu movimento
2. Monitoramento Contínuo com a função `too_close`: Enquanto a aeronave está pausada, o sistema monitora continuamente a distância entre as duas aeronaves envolvidas. Caso a distância entre elas continue insuficiente, o sistema verifica se há outras aeronaves próximas que possam interferir.
3. Resolução Final: Se, mesmo após a pausa, a colisão não puder ser evitada (por exemplo, devido à presença de outras aeronaves ou impossibilidade de manobra), o sistema considera a colisão inevitável e envia um sinal **SIGKILL** para a aeronave pausada, marcando-a como DEAD na simulação.
4. Retomada Segura: Caso a distância entre as aeronaves se torne segura, o sistema envia novamente o **SIGUSR1** para a aeronave pausada, alterando seu status para FLYING e permitindo que ela retome sua trajetória.

Esse mecanismo de pausa temporária com **SIGUSR1** é essencial para lidar com situações em que a mudança de pista não é suficiente para evitar colisões. Ele garante que o sistema possa reagir dinamicamente a cenários mais complexos - como, por exemplo, com número maior de aeronaves, priorizando a segurança do tráfego aéreo simulado.

Testes realizados

Para não estender muito o relatório, somente as partes mais relevantes para cada teste foram mostradas (i.e. linhas de printf foram omitidas).

Teste 1 - Criação de 10 aeronaves:

Para esse teste, criamos 10 aeronaves utilizando os comandos:
Comando no compilador: gcc -o main main.c
./main 10

Resultado:

aircraft 930 data: delay: 0s, side: 1, x: 0.00 and y: 0.00, runway: 3, speed_x: 0.05, status: 4
aircraft 931 data: delay: 1s, side: -1, x: 1.00 and y: 0.80, runway: 6, speed_x: 0.05, status: 4
aircraft 932 data: delay: 0s, side: -1, x: 1.00 and y: 0.40, runway: 6, speed_x: 0.05, status: 4
aircraft 933 data: delay: 0s, side: -1, x: 1.00 and y: 0.90, runway: 27, speed_x: 0.05, status: 4
aircraft 934 data: delay: 2s, side: 1, x: 0.00 and y: 0.80, runway: 3, speed_x: 0.05, status: 4
aircraft 935 data: delay: 0s, side: 1, x: 0.00 and y: 0.30, runway: 18, speed_x: 0.05, status: 4
aircraft 936 data: delay: 0s, side: -1, x: 1.00 and y: 1.00, runway: 27, speed_x: 0.05, status: 4
aircraft 937 data: delay: 0s, side: 1, x: 0.00 and y: 0.90, runway: 18, speed_x: 0.05, status: 4
aircraft 938 data: delay: 0s, side: -1, x: 1.00 and y: 0.10, runway: 27, speed_x: 0.05, status: 4
aircraft 939 data: delay: 1s, side: 1, x: 0.00 and y: 0.80, runway: 18, speed_x: 0.05, status: 4

O sistema cria 10 aeronaves dando valores de delay aleatórios, escolhendo um lado (leste ou oeste), tendo seu x = 0 ou 1 dependendo do lado escolhido, um y aleatório, uma pista escolhida, velocidade, e todos começando no status (FLYING).

Teste 2 - Criação com um número inválido de aeronaves:

Para esse teste, chamamos a main tentando criar 0 aeronaves.
./main 0
Number of aircrafts must be between 1 and 40

Depois, chamamos a main tentando criar 41 aeronaves.
./main 41
Number of aircrafts must be between 1 and 40

Nos dois casos deu erro, já que estabelecemos que devem ser criadas entre 1 e 40 aeronaves.

Teste 3 - Implementação dos movimentos das aeronaves:

Para esse teste, criamos 5 aeronaves e, após o início dos processos, os aviões começaram a se deslocar em direção às suas pistas de destino, com o processo pai controlando possíveis colisões através de uma memória compartilhada, contendo informações de todos os filhos.

aircraft 14757 - x: 0.95, y: 0.14, runway: 27

aircraft 14757 - x: 0.90, y: 0.18, runway: 27
aircraft 14759 - x: 0.95, y: 0.95, runway: 27
aircraft 14759 - x: 0.90, y: 0.90, runway: 27
aircraft 14761 - x: 0.95, y: 0.77, runway: 27
aircraft 14761 - x: 0.90, y: 0.74, runway: 27
aircraft 14757 - x: 0.85, y: 0.22, runway: 27
aircraft 14757 - x: 0.80, y: 0.26, runway: 27
aircraft 14758 - x: 0.05, y: 0.86, runway: 18
aircraft 14759 - x: 0.85, y: 0.85, runway: 27
aircraft 14760 - x: 0.05, y: 0.77, runway: 3
aircraft 14760 - x: 0.10, y: 0.74, runway: 3
aircraft 14761 - x: 0.85, y: 0.71, runway: 27

Teste 4 - Tratamento de colisão entre aeronaves:

Após criar 6 aeronaves, suas posições e pistas foram atualizadas ao longo do tempo até que todas pousassem ou fossem removidas (mortas).

aircraft 8457 - x: 0.95, y: 0.41, runway: 27
aircraft 8458 - x: 0.05, y: 0.50, runway: 3
[...]
aircraft 8457 - x: 0.80, y: 0.44, runway: 27

Durante a simulação, uma possível colisão foi detectada entre os aeronaves 8457 e 8459, ambos com:

- Mesma pista de destino (27),
- Mesma coordenada X (0.80),
- Coordenadas Y próximas, indicando rota de colisão iminente.

O sistema então emitiu os seguintes avisos:

- oops! aircrafts 8457 and 8459 might collide. let's fix it…
- changing runway of 8457 (SIGUSR2)
- scheduler: SIGUSR2 received, runway changed from 27 to 6

Essa ação alterou a pista do aeronave 8457 para evitar a colisão, redirecionando-o com segurança:

- aircraft 8457 - x: 0.70, y: 0.46, runway: 6

Teste 5 - Terminando o programa através do SIGQUIT (Control + \ )

Para esse teste, criamos aeronaves e pressionamos Control + \ para encerrar o programa, atestando que a função `stop_program_handler` funciona.

aircraft 12077 - x: 0.95, y: 0.23, runway: 6
aircraft 12078 - x: 0.05, y: 0.05, runway: 18
aircraft 12079 - x: 0.95, y: 0.14, runway: 6
aircraft 12079 - x: 0.90, y: 0.18, runway: 6
aircraft 12080 - x: 0.05, y: 0.86, runway: 18
aircraft 12081 - x: 0.95, y: 0.95, runway: 6
aircraft 12082 - x: 0.05, y: 0.77, runway: 3
aircraft 12083 - x: 0.95, y: 0.77, runway: 6
oops! aircrafts 12084 and 12086 might collide. let's fix it...
changing runway of 12084 (SIGUSR2)
aircraft 12085 - x: 0.95, y: 0.68, runway: 27
aircraft 12085 - x: 0.90, y: 0.66, runway: 27
^\finishing process...

Após isso, o programa foi encerrado com exit(0).

Teste 6 - Pausando todas as aeronaves (com Control + Z) e as despausando depois

Para esse teste, criamos 15 aeronaves e depois da criação delas, pressionamos Control + Z, atestando que a função `pause_resume_handler` funciona.

aircraft 9326 data: delay: 0s, side: 1, x: 0.00 and y: 0.40, runway: 18, speed_x: 0.05, status: 4
[...]
aircraft 9320 - x: 0.15, y: 0.29, runway: 18
aircraft 9321 - x: 0.95, y: 0.32, runway: 6
[…]

^Z (Parando todas as aeronaves através do Control + Z)
pausing all aircrafts...
pausing aircraft 9319
pausing all aircrafts...
pausing aircraft 9319
pausing aircraft 9320
pausing aircraft 9320
pausing aircraft 9321
pausing aircraft 9322
pausing aircraft 9323
pausing aircraft 9324
pausing aircraft 9325
pausing aircraft 9326

^Z (Despausando todas as aeronaves paradas através do Control + Z)
resuming all aircrafts...

resuming aircraft 9319
aircraft 9319 - x: 0.85, y: 0.85, runway: 27
resuming aircraft 9320
oops! aircrafts 9320 and 9326 might collide. let's fix it...
changing runway of 9320 (SIGUSR2)
resuming aircraft 9321
pausing aircraft 9321

FINAL STATS:

successfully landed aircrafts: 8
killed aircrafts: 0
100.00% of the aircrafts landed successfully.

Teste 7 - Enviando SIGINT para todas as aeronaves (com Control + C)

Para esse teste, criamos 15 aeronaves e antes de qualquer uma delas pousar, pressionamos Control + C para enviar um **SIGKILL** em todas, atestando que a função `kill_handler` funciona.

aircraft 1085 data: delay: 0s, side: -1, x: 1.00 and y: 0.10, runway: 6, speed_x: 0.05, status: 4
[...]
aircraft 1099 data: delay: 0s, side: -1, x: 1.00 and y: 0.10, runway: 27, speed_x: 0.05, status: 4

oops! aircrafts 1085 and 1090 might collide. let's fix it...
changing runway of 1085 (SIGUSR2)
scheduler: SIGUSR2 received, runway changed from 6 to 27
changing runway of 1085 didn't work. changing it back
scheduler: SIGUSR2 received, runway changed from 27 to 6
pausing aircraft 1085 (SIGUSR1)

[Outras operações…]

^Ckilling all aircrafts in traffic.

FINAL STATS:

successfully landed aircrafts: 0
killed aircrafts: 15
0.00% of the aircrafts landed successfully.

Para outra versão desse teste, esperamos algumas das aeronaves pousarem para garantir que o usuário apenas conseguiria matar aeronaves que não tivessem pousado ainda.

Teste 8 - Enviando SIGINT para aeronaves após algumas aeronaves terem pousado

Para esse teste, criamos 35 aeronaves e, assim que qualquer uma delas pousar, pressionamos Control + \ para enviar um **SIGINT** para todas, matando os processos das aeronaves que ainda não pousaram.

aircraft 13734 landed at runway 6 in (x,y) = (0.50,0.50)
[…]
im still too close [13737 status 3, 13747 status 4]
aircraft 13737 PAUSED at (0.40, 0.48)
im still too close [13737 status 3, 13747 status 4]
aircraft 13737 PAUSED at (0.40, 0.48)
[…]
aircraft 13737 PAUSED at (0.40, 0.48)
im still too close [13737 status 3, 13747 status 4]
aircraft 13737 PAUSED at (0.40, 0.48)
^Ckilling all aircrafts in traffic.
killing all aircrafts in traffic.
aircraft 13756 landed at runway 27 in (x,y) = (0.50,0.50)

FINAL STATS:

successfully landed aircrafts: 2
killed aircrafts: 33
5.00% of the aircrafts landed successfully.

Esses foram os principais testes, realizados de forma incremental e cronológica conforme o código foi sendo desenvolvido. Com isso, nossa lógica passo a passo de implementação é dada a seguir:

1. Código capaz de criar n aeronaves e armazená-las num vetor de memória compartilhada entre os n+1 processos (n filhos e o pai).
2. Verificações de que o número passado como argumento ao rodar o programa seria um inteiro válido (entre 1 e 40) de aeronaves. Além disso, nessa fase implementamos também a verificação de outros possíveis erros (como, por exemplo, no momento de criação da memória compartilhada).
3. Ainda sem nenhuma lógica de colisão, implementamos o movimento das aeronaves utilizando a BASE_SPEED para o x e o cálculo explicado previamente para o y.
4. Aplicação da lógica de colisão.
5. Implementação da interface onde o usuário encerra o programa com Control + \ ao invés de Control + C.
6. Implementação da interface onde o usuário pode controlar o ciclo das aeronaves, pausando, resumindo e matando elas.

Dificuldades de implementação

Ao longo do desenvolvimento do trabalho, encaramos dois grandes problemas. Primeiramente, tivemos uma grande dificuldade ao implementar a interface gráfica. Estávamos utilizando a função `getchar` inicialmente para saber qual caractere o usuário estava clicando no teclado e, dependendo do caracter, chamávamos ou a função que dava kill em todas as aeronaves, ou a função que finaliza o programa, ou a função que pausa e despausa todas as aeronaves. Porém, essa função não era sofisticada o suficiente para atender às nossas necessidades por diversas razões, mas principalmente por compatibilidade com a matéria que estamos vendo nas aulas e por não conseguir captar com a precisão necessária o que o usuário fornecia como input.

Após isso, começamos a pegar os caracteres do teclado através de sinais, e não de `getchar`. Implementamos manipuladores para sinais como **SIGINT**, **SIGQUIT** e **SIGTSTP**, que nos permitiram associar ações específicas a cada comando do usuário. Por exemplo, o **SIGINT** foi utilizado para matar todas as aeronaves em trânsito, o **SIGQUIT** para finalizar o programa de forma ordenada, e o **SIGTSTP** para pausar ou retomar todas as aeronaves simultaneamente. Essa abordagem tornou o sistema mais robusto e responsivo, permitindo que as ações fossem tratadas de forma assíncrona e independente do fluxo principal do programa.

Conclusões e aprendizados

O desenvolvimento deste trabalho nos proporcionou uma compreensão prática e aprofundada sobre os conceitos fundamentais de sistemas operacionais vistos nas aulas, como processos, sinais, memória compartilhada e sincronização. A implementação de um sistema de controle de tráfego aéreo simulando aeronaves em movimento nos desafiou a lidar com problemas reais de concorrência e comunicação entre processos.

Entre os principais aprendizados, destacamos a importância de manipular sinais para controlar o comportamento dos processos de forma assíncrona, o que nos permitiu implementar funcionalidades como pausa, retomada e alteração de pistas de pouso. Além disso, o uso de memória compartilhada foi essencial para garantir que todos os processos tivessem acesso às informações atualizadas sobre as aeronaves, reforçando a necessidade de sincronização para evitar inconsistências.

Também aprendemos a lidar com situações de risco, como colisões iminentes, utilizando estratégias como alteração de pistas e pausas temporárias, demonstrando a importância de prever e tratar cenários críticos em sistemas concorrentes. Por fim, enfrentamos desafios técnicos, como a substituição de `getchar` por sinais para capturar comandos do usuário, o que nos ensinou a buscar soluções mais robustas e adequadas às necessidades do sistema.