Manual - ThreadBet

Trabalho da disciplina Sistemas Operacionais

Daniel Jorge Manzano - 15446861
Nicolas Amaral dos Santos - 16304033

November 2025

1 Introdução

Este manual tem como objetivo servir como um guia ao usuário sobre como baixar, rodar e jogar o "ThreadBet", além de apontar resumidamente as aplicações e implementações de threads e semáforos presentes nele.

2 Instalação e Requisitos

Para a instalação, certifique-se de ter baixado todos os códigos necessários: *main.cpp*, *corredor.hpp* e *Makefile*, ou apenas baixe e extraia *trabalho_SO.zip*, que contém todos os arquivos necessários. Para encontrá-los, é possível acessar o repositório localizado em ThreadBet; a partir dele, clone o repositório localmente ou baixe o(s) arquivo(s) manualmente.

2.1 Compilação e Execução

Para compilar o jogo, é necessário ter um compilador C++ instalado (como g++). No terminal, navegue até a pasta do projeto e digite o comando

```
make
```

Com o código compilado, execute o jogo com o comando

```
make run
```

Nota: a qualquer momento, a execução pode ser interrompida forçadamente através do comando `Ctrl + C` (válido para Windows, Linux e macOS).
Para "limpar" a pasta (deletar o executável gerado), use o comando

```
make clean
```

3 Manual do Jogador

3.1 Fluxo do Jogo

Ao iniciar o **ThreadBet**, o usuário deve seguir os seguintes passos:

1. **Configuração do Sistema:** Selecionar o Sistema Operacional (Windows ou Linux/Mac) para garantir que a limpeza de tela funcione corretamente.
2. **Definição da Pista:** Escolher o tamanho da pista em metros (recomendado entre 500 e 1000 para uma boa visualização).
3. **Número de Corredores:** Definir quantos corredores participarão da prova (máximo de 20). É válido apontar que usar mais corredores torna as impressões da pista menos suaves em certos casos (existem pequenos "*glitches*" visuais em certos momentos da corrida, mas não são tão perceptíveis a ponto de atrapalhar a experiência).
4. **Análise e Aposta:** O sistema gerará corredores com atributos aleatórios (Velocidade Mínima, Máxima e Resistência). O jogador deve analisar esses dados e escolher um ID para apostar.
5. **Corrida:** A corrida inicia. O jogador pode acompanhar o progresso em tempo real e verificar o resultado do pódio da corrida, assim como o sucesso de sua aposta, ao fim da simulação.

3.2 Legenda Visual

O progresso dos corredores é simbolizado por uma sequência de hífens (-). Ao fim dessa sequência, um caracter simboliza o movimento do corredor; os caracteres, com seus respectivos significados, são:

- $>$: O corredor está se movendo normalmente.
- **X** : O corredor tropeçou (azar) e perdeu o turno. O visual aparece apenas durante o tempo que ele está parado.
- # : O corredor sofreu uma lesão grave e foi eliminado da prova.
- **F** : O corredor finalizou a prova.

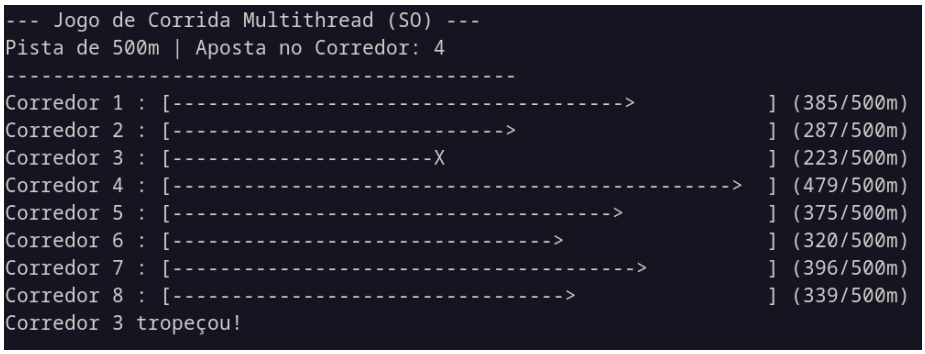

Figure 1: Exemplo de print da pista durante a execução

4 Implementação Técnica Principal

A implementação do *ThreadBet* baseia-se, principalmente, nos conceitos de programação concorrente estudados na disciplina de Sistemas Operacionais. Nesse sentido, exploramos, abaixo, particularidades acerca disso.

4.1 Threads (std::thread)

Para simular uma corrida real, onde cada competidor age de forma independente e simultânea, utilizamos a biblioteca `<thread>` do C++.

- **Por que foi utilizada?** Se utilizássemos uma abordagem sequencial (um loop simples), um corredor teria que terminar seu movimento para que o outro começasse, ou teríamos que simular "tiques" de relógio manualmente. Com threads, o Sistema Operacional se encarrega do escalonamento, permitindo que cada instância da função `correr()` seja executada em paralelo (ou pseudo-paralelo, dependendo do hardware).
- **Como foi utilizada?** Cada corredor é instanciado como uma thread separada dentro de um `std::vector<thread>`. A função `correr` recebe por referência o objeto do corredor e os vetores de estado compartilhados. Ao final, utilizamos `join()` na thread principal (*main*) para garantir que o programa aguarde o término de todas as execuções antes de encerrar.

4.2 Sincronização e Exclusão Mútua (std::mutex)

Com múltiplas threads acessando e modificando recursos compartilhados simultaneamente, surgem problemas de *Race Condition* (Condição de Corrida). Para resolver isso, utilizamos Semáforos Binários (implementados via `std::mutex`).

Foram identificadas três Regiões Críticas principais no código:

1. **O Pódio (mutexVencedor):** O vetor `podio` é um recurso compartilhado. Se dois corredores chegassem na linha de chegada no exato mesmo ciclo de processamento, ambos tentariam fazer um `push_back` ao mesmo tempo, podendo corromper a memória ou perder dados. O mutex garante que apenas um corredor por vez registre sua chegada.

2. **Gerador de Números Aleatórios (mutexRNG):** A função `rand()` ou geradores do `std::random` mantêm um estado interno. O acesso concorrente a esse gerador pode causar falhas de segmentação ou gerar números repetidos/previsíveis. O mutex garante que apenas um corredor gere os números por vez.

3. **Contador de Finalizados (mutexThreads):** Para evitar que o jogo entre em loop infinito caso um corredor seja eliminado (não chegando ao pódio), utilizamos um contador global de threads encerradas. Como todas as threads incrementam essa variável ao sair, o acesso precisa ser atômico/protegido.

5 Análise do Código

Tendo analisado em destaque as implementações relacionadas a *threads* e *semáforos*, neste tópico, temos como objetivo analisar o código como um todo.

5.1 Bibliotecas Utilizadas

Para a construção do *ThreadBet*, utilizamos a linguagem C++ (padrão C++11 ou superior), aproveitando suas bibliotecas padrão para gerenciamento de memória, controle de fluxo e, principalmente, programação concorrente. Abaixo, detalhamos as principais bibliotecas incluídas e suas respectivas funções no projeto:

- `<thread>`: Biblioteca fundamental para o projeto. Ela fornece a classe `std::thread`, que permite a criação e gerenciamento de linhas de execução paralelas. Utilizamos também o `std::this_thread::sleep_for` para controlar a velocidade da simulação, pausando as threads por períodos determinados.

- `<mutex>`: Essencial para a sincronização. Fornece a primitiva de exclusão mútua `std::mutex` e o gerenciador de escopo `std::lock_guard`. Essas ferramentas foram utilizadas para proteger regiões críticas (os acessos ao vetor de pódio, contador de threads finalizadas e gerador de números aleatórios), evitando condições de corrida (*Race Conditions*).

- `<vector>`: Utilizamos a estrutura de dados `std::vector` para gerenciamento de listas. Ele armazena as instâncias das threads, os objetos dos corredores, o estado visual da pista e a ordem de chegada (pódio), permitindo o redimensionamento automático conforme a entrada do usuário.

- `<random>`: Substitui o antigo `rand()` da linguagem C por uma geração de números aleatórios mais completa. Utilizamos o `std::default_random_engine` para o motor de geração e distribuições como `uniform_int_distribution` para calcular os passos e atributos dos corredores.
- `<chrono>`: Biblioteca de manipulação de tempo. Foi utilizada para duas funções principais: fornecer a semente (*seed*) baseada no relógio do sistema para o gerador aleatório (garantindo corridas diferentes a cada execução) e definir as unidades de tempo (milissegundos e segundos) para as pausas das threads.
- `<iostream>`: Biblioteca padrão de entrada e saída. Responsável pela interação com o usuário via terminal através de `std::cin` (leitura de configurações e apostas) e `std::cout` (impressão da pista e resultados).
- `<cstdlib>`: Incluída para permitir chamadas de sistema através da função `std::system()`, utilizada especificamente para limpar a tela do terminal (`"cls"` ou `"clear"`), criando o efeito de animação quadro a quadro.
- `"corredor.hpp"`: Um cabeçalho local (criado para o projeto) que define a estrutura `Corredor`. Essa modularização permite separar a lógica dos atributos do atleta (velocidade, resistência, estado de lesão) da lógica principal de controle de threads.

5.2 Variáveis Globais e Controle de Concorrência

Para permitir a comunicação entre as diversas *threads* e a *main*, utilizamos variáveis globais. A decisão de torná-las globais justifica-se pela necessidade de acesso compartilhado constante, o que exige, em contrapartida, um controle rigoroso de acesso:

- **Recursos Compartilhados:**
  - `vector<int> podio`: Armazena a ordem de chegada dos atletas. É a região crítica mais importante do jogo.
  - `default_random_engine gerador`: O motor de aleatoriedade. Como não é *thread-safe* por padrão, seu acesso deve ser protegido.
  - `int threadsFinalizadas`: Contador atômico (protegido) que permite à *thread* principal saber quando a corrida acabou, independente de os corredores terem cruzado a linha de chegada ou terem sido eliminados.
- **Semáforos (Mutexes):**
  - `mutexVencedor`: Garante a exclusão mútua na escrita do pódio.
  - `mutexRNG`: Garante a exclusão mútua na geração de números aleatórios.
  - `mutexThreads`: Protege o incremento do contador de threads finalizadas.

5.3 Estrutura `Corredor` (corredor.hpp)

A lógica de negócio de cada atleta foi encapsulada na `struct Corredor`. Isso promove a modularidade e limpa o código principal. Seus principais componentes são:

- **Atributos:** Cada corredor possui velocidade mínima e máxima, resistência (probabilidade de fadiga) e estados de controle como `turnosParado` (contador para penalidades de tempo) e a flag `eliminado` (para lesões graves).
- **Método `darPasso()`:** É a função principal do corredor. A cada chamada, ele calcula o resultado do movimento:
  1. Verifica se o corredor está penalizado (`turnosParado > 0`).
  2. Calcula chance de eliminação (0.02%).
  3. Calcula chance de tropeço grave (0.05%, gerando 15 turnos de penalidade).
  4. Calcula a fadiga baseada na resistência, o que pode afetar a escolha da velocidade no passo (caso o corredor "esteja fadigado", corre com a velocidade mínima no passo)
  5. Retorna a distância percorrida no turno.
- **Método `mostrarAtributos()`**: Uma função básica que apenas imprime os atributos `id`, `velMin`, `velMax` e `resistencia` do corredor.

5.4 Função `correr`

Esta função representa o ciclo de vida de uma *thread*. Ela é executada paralelamente para cada corredor instanciado. Seu funcionamento consiste em um *loop* executado enquanto o corredor não completar a pista e não for eliminado.

Destaques da implementação:

- **Região Crítica do RNG:** Ao chamar `meuCorredor.darPasso()`, utilizamos um `lock_guard` no `mutexRNG` para garantir que os números aleatórios sejam gerados com segurança.
- **Atualização de Estado Visual:** A função atualiza um vetor de caracteres (`estadosCorredores`) compartilhado, permitindo que a *main* saiba se deve desenhar o corredor correndo (`>`), parado (`X`) ou eliminado (`#`).
- **Simulação de Tempo:** O comando `sleep_for(50ms)` é crucial para ditar o ritmo da corrida e impedir que o processador execute a simulação instantaneamente.

5.5 Funções auxiliares de impressão

Para separar a lógica de processamento da lógica de apresentação, criamos funções específicas para lidar com a saída no terminal, o que facilita a manutenção e a legibilidade do código.

- `impressao_pista`: É a função de renderização principal. A cada ciclo de atualização (frame), ela calcula a posição visual proporcional de cada corredor baseada no tamanho total da pista e desenha a barra de progresso. Também verifica o vetor `estadosCorredores` para decidir se imprime o símbolo de movimento (`>`), tropeço (`X`), eliminação (`#`) ou finalizado (`F`).
- `impressao_podio`: Responsável por acessar o vetor global `podio` ao final da execução e formatar a saída para exibir os IDs dos três primeiros colocados.
- `impressao_resultado`: Realiza a comparação lógica entre a variável da aposta inserida pelo usuário e o primeiro elemento do vetor `podio`, exibindo a mensagem de vitória ou derrota correspondente.
- `limpa_terminal`: Abstrai a chamada de sistema (`system("cls")` para Windows ou `system("clear")` para Unix). É invocada no início de cada iteração do loop visual na *main* para apagar o quadro anterior, criando a ilusão de "animação" de movimento contínuo.

5.6 Função `gera_corredores`

Esta função centraliza as operações referentes à criação e impressão inicial dos corredores e seus respectivos atributos. Para a criação, gera estatísticas aleatórias referentes às velocidades mínima e máxima e à resistência; após a criação, imprime os atributos com `mostrarAtributos`. Como retorno, gera o vetor de corredores com atributos inicializados.

5.7 Função `main`

A função principal atua como o processo "pai" da aplicação. Ela é responsável por três estágios:

1. **Inicialização:** Configura o ambiente, lê os parâmetros do usuário (tamanho da pista, apostas) e instancia os objetos `Corredor`.
2. **Execução e Renderização:** Cria as threads usando `emplace_back` e entra em um laço de repetição visual. Enquanto o contador global `threadsFinalizadas` for menor que o total de participantes, a *main* limpa o terminal e redesenha a pista baseada nos vetores compartilhados. Isso separa a lógica de processamento (nas threads) da lógica de apresentação (na main).
3. **Sincronização Final:** Após o "fim da corrida", executa o método `.join()` em todas as threads para garantir o encerramento do programa e exibe os resultados.

6 Conclusão

O desenvolvimento do *ThreadBet* cumpriu o objetivo de ilustrar o gerenciamento de processos em Sistemas Operacionais. A implementação da execução paralela com threads e do controle de concorrência com semáforos resultou em uma aplicação lúdica que, além de funcional, serviu para consolidar o entendimento a respeito da aplicação dessas ferramentas.