Relatório - Prática em Threads e Semáforos

Implementação do jogo Whack-a-mole utilizando programação *multithreading*

Link para o vídeo explicativo:
https://drive.google.com/file/d/1Iny7POb9eOaBTJAHo0Prn2hdJ5dCb3Ob/view

Informações Gerais

Trabalho prático da disciplina de Sistemas Operacionais I - SSC 0140, ICMC - USP São Carlos, pela professora Kalinka Regina Lucas Jaquie Castelo Branco.

O relatório tem como objetivo contextualizar a implementação de um jogo interativo que utilize de técnicas de controle de acesso à memória e programação multithreading - temas apresentados na disciplina.

O relatório acompanha um vídeo, que perpassa o código fonte, apresentando o projeto e também demonstrando seu funcionamento.

Integrantes:

- Enzo Nunes Sedenho - 13671810
- João Brasileiro Moreira da Silva - 13672957
- Luiz Schulz - 13732769
- Vicenzo D'Arezzo Zilio - 13671790

Contexto

A ideia:

O jogo "*Whack-a-Mole*" é um jogo interativo de um jogador. Sua estrutura básica envolve buracos, no qual "*moles*" - aqui referenciados como **toupeiras** - aparecem aleatoriamente. O objetivo do jogador é ser capaz de "bater" nas toupeiras emergentes antes que elas desapareçam novamente nos buracos. Cada toupeira atingida geralmente resulta em uma pontuação, e o jogador busca alcançar a pontuação mais alta possível dentro de um limite de tempo.

A justificativa:

Sobre o contexto de jogos, existem inúmeras possibilidades de projetos, cujas complexidades escalam muito rápido e dependem de recursos como *Threads e Semáforos* para um funcionamento pleno.

Projetando essa realidade ao escopo da disciplina de Sistemas Operacionais, optou-se por um jogo como o *Whack-a-Mole*, no qual se faz clara a definição e a relação da arquitetura do projeto em diferentes threads - leitura, toupeiras, jogadores, pontuação, ... - que configuram uma condição de corrida em relação a variáveis de controle e ao tabuleiro, evidenciando a necessidade das técnicas supracitadas de uma maneira didática.

Outro aspecto importante, tendo em vista o objetivo do projeto, é a facilidade de lidar com a interface gráfica. A definição do jogo e seu porte permitem a implementação dessa interface utilizando leituras coordenadas dentro da entrada padrão do processo: o terminal.

A ferramenta:

Para tal proposta, escolheu-se a linguagem de programação C++, que permite a manipulação da memória de maneira mais explícita, assim como a definição de classes e objetos que auxiliam na arquitetação do projeto.

A linguagem também dispõe de recursos como bibliotecas - descritas na seção de implementação - que auxiliam esse processo.

Implementação e estrutura

Classes:

- **Jogador** - Atributos e métodos que envolvem o usuário e seus movimentos;
- **Tela** - Métodos que coordenam e organizam a renderização da interface do terminal;
- **Toupeira** - Atributos e métodos que representam as toupeiras e seu comportamento;
- **WhackAMole** - Classe que centraliza e coordena as demais, representando o jogo em si. Ela inicia e encerra o jogo.

Bibliotecas:

- **Iostream:** Para o tratamento de entrada e saída.
- **Cstdlib:** Para geração de números aleatórios e chamadas de sistema.
- **Map, Vector:** Estruturas de dados utilizadas na implementação.
- **Unistd.h:** Utilizada para acessar parâmetros do terminal e os descritores de entrada e saída.
- **Termios.h:** Trata o buffer dos dispositivos de entrada e saída.
- **Thread:** Estrutura de dados que possibilita a criação de threads.
- **Mutex:** Semáforo binário para a sincronização das threads.
- **Sys/ioctl.h:** Utilizada para obter atributos do terminal.

Threads - dividindo o processamento:

Threads são importantes para dividir a execução de um processo. Porém, podemos aproveitar tal mecanismo para repartir morfologicamente a estrutura da aplicação, isto é, separar os diferentes entes do sistema em funcionamentos específicos e encapsulados e executados de forma concorrente - cada um com sua thread.

Sobre essa perspectiva, optamos por utilizar da morfologia de nossa arquitetura para separar o processo entre:

Jogador:

Representa o jogador interagindo com o tabuleiro, desde a leitura da ação até a execução da mesma sobre o tabuleiro. A representação do jogador enquanto thread nos permite delegar o processamento disjunto das operações de entrada (leitura dos comandos executados pelo jogador), tornando a interação entre o usuário e a aplicação mais fluida e coerente com o propósito do jogo.

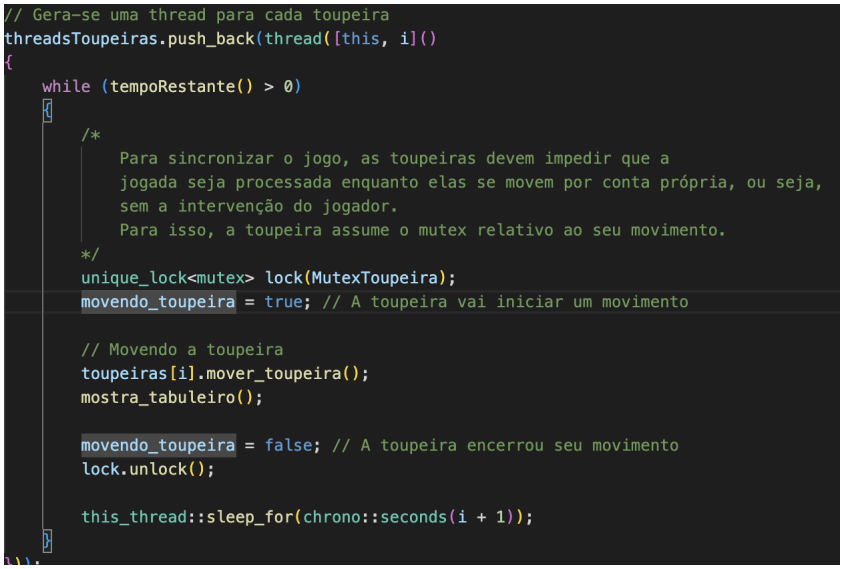
- Cada jogador deve ser mapeado a uma thread, no caso deste trabalho, implicando em apenas uma thread desta categoria.
- Por sua natureza ímpar, ela é evocada diretamente na função de execução do jogo, referenciando outras funções e classes;

```
// Gera-se uma thread para o jogador (leitura e tratamento da stdin)
thread jogadorThread([this]()
{
    Jogador jogador = Jogador();
    while (tempoRestante() > 0)
    {
        // Caso o movimento da toupeira nao esteja sendo processado:
        // Le (e trata) o input do jogador
        if(!movendo_toupeira)
            verifica_jogada(jogador);
    }
});
```

Toupeiras (*Moles*):

Referencia uma única toupeira e seu comportamento, que está associado a uma posição no tabuleiro. A utilização das threads associadas às toupeiras permite que as mesmas sejam desassociadas do tabuleiro, realizando seus movimentos de maneira independente, tornando o jogo mais dinâmico.

- Cada toupeira é mapeada a uma Thread, de forma que estas sejam executadas de maneira independente;
- Mantém-se uma lista de toupeiras e threads na classe principal - *WhackAMole* - permitindo tanto a manipulação individual quanto conjunta;



```
// Gera-se uma thread para cada toupeira
threadsToupeiras.push_back(thread([this, i]()
{
    while (tempoRestante() > 0)
    {
        /*
            Para sincronizar o jogo, as toupeiras devem impedir que a
            jogada seja processada enquanto elas se movem por conta própria, ou seja,
            sem a intervenção do jogador.
            Para isso, a toupeira assume o mutex relativo ao seu movimento.
        */
        unique_lock<mutex> lock(MutexToupeira);
        movendo_toupeira = true; // A toupeira vai iniciar um movimento

        // Movendo a toupeira
        toupeiras[i].mover_toupeira();
        mostra_tabuleiro();

        movendo_toupeira = false; // A toupeira encerrou seu movimento
        lock.unlock();

        this_thread::sleep_for(chrono::seconds(i + 1));
    }
}));
```

Semáforos - Lidando com a região crítica:

Tendo em vista as threads apresentadas, a condição de corrida se configura sobre o tabuleiro. Enquanto todas as threads de toupeiras acessam a região para serem reposicionadas conforme as regras, o jogador acessa-a para ocupar um espaço, buscando pontuar sobre uma das toupeiras.

tabuleiro → "MutexTabuleiro" ;

Uma vez definida a região crítica, é necessário garantir o acesso sincronizado à mesma. Para tal, utilizamos de um semáforo binário (Mutex) para assegurar a leitura e escrita sobre cada uma das casas do tabuleiro.

Vamos primeiramente ilustrar uma problemática que pode ser causada pela ausência deste semáforo na implementação:

Suponha que uma toupeira acabou de calcular uma posição válida no tabuleiro e, imediatamente antes de ocupá-la, o processo foi interrompido. Ao longo da interrupção, outra toupeira calculou e ocupou a mesma posição. Quando o processo for executado novamente, a primeira toupeira sobrescreverá a segunda no tabuleiro, prejudicando o funcionamento do jogo.

Para resolver esta (assim como muitas outras) questões, definimos um *mutex* (*MutexTabuleiro*) para guardar todas as operações relacionadas à leitura e escrita do tabuleiro.

Dessa forma, sempre que uma toupeira se move, o *MutexTabuleiro* guarda a execução do método *mover_toupeira()*, o qual é responsável por desocupar a posição atual da toupeira, calcular uma nova posição, e ocupar a posição recém calculada.

Além disso, quando o jogador realiza uma jogada, a leitura do tabuleiro e a ocupação da casa acertada são protegidas pelo *mutex*, ao fim da verificação, quando o usuário desocupar a posição, também acionamos um “lock” no mutex com o intuito de proteger o tabuleiro.

movendo_toupeira → "MutexToupeira" ;

Além do acesso restrito ao tabuleiro, o jogo também necessita de uma sincronização entre o processamento das jogadas e o movimento das toupeiras, uma vez que tais rotinas são realizadas em threads independentes. Assim, definimos a variável *booleana* “movendo_toupeira”, que determina se uma toupeira está realizando movimento, esta variável é a região crítica protegida pelo mutex “MutexToupeira”.

Tendo isso em mente, vejamos novamente uma problemática que exemplifica a necessidade de tal sincronização entre jogador e toupeiras:

Suponha que o jogador realize uma jogada que acertaria uma toupeira. Após o cálculo da posição acertada pelo jogador, este processo é interrompido e a toupeira que se localizava na casa acertada se move. Note que, quando a thread relativa ao jogador voltar a processar a jogada, um acerto será identificado, uma vez que o cálculo realizado antes da preempção corresponde ao *id* de uma toupeira, mas esta toupeira já não se encontra na posição escolhida pelo jogador. Ou seja, o jogador ganharia pontos por acertar uma casa vazia.

Com o intuito de solucionar este problema, utiliza-se da variável de controle *movendo_toupeira*, que é disposta para a sincronização entre o processamento da entrada e o movimento das toupeiras em suas respectivas threads.

Neste sentido, sempre que a toupeira for se mover, a variável *movendo_touepira* deve ser identificada como *true*, e ao fim do movimento da toupeira (antes da thread “dormir”) a variável é identificada como *false*. Para garantir a sincronização do estado desta variável entre as thread de toupeira, utiliza-se o mutex *MutexToupeira*, que guarda a operação de atribuição de valores à variável *movendo_toupeira*.

Desta forma, sempre que uma toupeira estiver se movendo, o comando do jogador aguarda o fim do movimento da mesma para ser processado, evitando erros de sincronização como nos moldes descritos anteriormente.

Configuração:

Para organizar e encapsular o código, as variáveis são definidas de acordo com a seguinte descrição:

Regiões Críticas e Semáforos Binários: São definidas em um arquivo de variáveis globais: *"globais.cpp”* e *"globais.h”*.

- ***Pragma Once:*** Diretiva da linguagem para garantir que esse arquivp de cabeçalho seja incluído apenas uma vez, evitando problemas de inclusão múltipla (problemas relacionados a duplicação de inclusões durante o processo de compilação);

Métodos de Acesso

- **Toupeira:** Método "*mover_toupeira()*” definido na classe *Toupeira*
- **Jogador:** Método *"ocupa_posicao_jogador()*" e *"desocupa_posicao_jogador()"* definidos na classe *Jogador*.

Execução do Acesso:

A chamada das classes e dos métodos que executam, efetivamente, o acesso e

sincronização das threads durante o jogo está no método *"inciarJogo()"* da classe *WhackAMole.*

Manual de Instalação

Requisitos mínimos:

Para o funcionamento da aplicação, é necessária a utilização de um sistema operacional LINUX, visto que a implementação do programa utiliza algumas biblioteca que não são disponibilizadas em outros sistemas operacionais, além de que o jogo utiliza o terminal como sua interface gráfica exclusiva, e para tal se baseia nos padrões e métodos reservados ao UNIX.

Além disso, é necessário que haja instalado o C++11, ou qualquer versão mais recente.

Inicialização:

A inicialização do jogo é bastante simples, uma vez que todas as bibliotecas utilizadas são padrão da linguagem C++, não há necessidade de instalar nenhum pacote adicional.

Para executar o jogo, basta compilar o programa com o arquivo Makefile fornecido utilizando o seguinte comando:

make all; make run

Feito isso, é esperado que o jogo esteja em funcionamento e as instruções iniciais sejam exibidas no terminal, bastando apenas realizar as interações indicadas pela aplicação e dar início ao jogo efetivamente.

Manual do Jogo

O jogo é uma adaptação clássica do "Whack-a-Mole", no qual o jogador é desafiado a acertar o maior número possível de toupeiras em um período de tempo determinado, acumulando pontos durante o processo. Inspirado nesse clássico, nossa versão exibe um tabuleiro 3 x 3 representando todas as possíveis posições das marmotas. Três marmotas, identificadas pelo caractere ‘O’, são posicionadas aleatoriamente no grid.

O tabuleiro é mapeado para as teclas do teclado da seguinte forma (considere as letras abaixo dispostas em um grid 3 x 3 para melhor compreensão):

O jogador deve observar a posição das marmotas no tabuleiro e pressionar a tecla correspondente à coordenada da toupeira no teclado. Dessa forma, pontos são acumulados e exibidos no final do jogo.

As toupeiras possuem um tempo de espera em cada posição. Dessa forma, caso o jogador não acerte a toupeira, ela se retira de sua posição atual e ocupa outra.

Além disso, caso o usuário acerte a toupeira, é exibido um ‘X’ na cor verde indicando o acerto, e a toupeira acertada muda de posição.