NOME: Gabriel de Araujo Lima NUSP: 14571376
NOME: Luis Henrique Ponciano dos Santos NUSP: 15577760
NOME: Gabriel Demba NUSP: 15618344
NOME: Wiltord N M NUSP: 15595392

MANUAL TÉCNICO E DE USUÁRIO

Disciplina: SSC0140 - Sistemas Operacionais I **Universidade de São Paulo - ICMC**

---

1. Introdução do Manual

Linguagem: C++ (Standard C++17) [4]

Objetivo Geral: Controlar um personagem em um labirinto para atravessar uma ponte estreita e alcançar a bandeira do lado oposto antes do adversário.

Este jogo foi projetado para demonstrar na prática conceitos fundamentais de **concorrência e sistemas operacionais**, especificamente a utilização de **Threads** para simulação de tempo real e **Semáforos** para coordenação de acesso a recursos compartilhados (Exclusão Mútua Lógica).

2. Requisitos para Executar o Jogo

Para garantir a execução correta, o ambiente deve atender aos seguintes requisitos:

- **Sistema Operacional:** Linux (Recomendado devido ao uso nativo de pthreads e ncurses) ou Windows (via WSL - Windows Subsystem for Linux).
- **Compilador:** `g++` (GCC) com suporte a C++17.
- **Bibliotecas Externas:**
  - `ncurses`: Biblioteca para construção de interfaces de texto (TUI).
  - `pthread`: Biblioteca padrão POSIX para threads (geralmente já incluída no Linux).

Instalação de Dependências (Debian/Ubuntu/WSL):

```
Bash

sudo apt-get update
sudo apt-get install libncurses5-dev libncursesw5-dev build-essential
```

3. Instruções de Compilação

O projeto consiste em um único arquivo fonte (game.cpp). Para compilar, abra o terminal na pasta do arquivo e execute:

Comando de Compilação:

Utilizando MAKEFILE:

make
make run

Ou manualmente

```
Bash

g++ -std=c++17 game.cpp -pthread -lncurses -o semaphore_game
```

Como Limpar (remover executável antigo):

```
Bash

rm semaphore_game
```

4. Como Executar o Jogo

Após a compilação bem-sucedida, execute o binário gerado:

```
Bash

./semaphore_game
```

- **Para fechar:** Pressione a tecla q a qualquer momento ou aguarde um jogador vencer.

5. Como Jogar (Gameplay)

O jogo simula uma corrida em um ambiente compartilhado.

- **Objetivo:** Levar o seu personagem até a bandeira (F) localizada no lado oposto do mapa.
- **O Desafio:** Existe apenas uma ponte (CC) ligando os dois lados. Esta ponte é estreita e permite a passagem de **apenas um jogador por vez**.
- **Controles:**
    - **Jogador 1 (Ciano - Esquerda):** Teclas W (Cima), S (Baixo), A (Esquerda), D (Direita).
    - **Jogador 2 (Vermelho - Direita):** Setas Direcionais (↑, ↓, ←, →).

- **Regras:**
    - Paredes (#) bloqueiam o movimento.
    - Se a ponte estiver ocupada, você não conseguirá entrar nela até que o adversário saia.
    - Vence quem tocar a bandeira F no lado oposto primeiro.

6. Arquitetura Geral do Código

O código é estruturado de forma modular dentro de game.cpp:

1. **Recursos Compartilhados:** Matrizes globais base_map (lógica) e map_view (visual) acessadas por todas as threads.
2. **Struct Player:** Mantém o estado (x, y, direção) de cada jogador.
3. **Funções Core:**
    - move_player(): Lógica de colisão, acesso à região crítica e atualização de posição.
    - draw_map(): Renderização da interface gráfica.
    - thread_player(): Loop de vida da thread de cada jogador.
4. **Main:** Inicializa recursos, cria as threads e gerencia o input do usuário.

7. THREADS - O porquê foram utilizadas

O projeto utiliza **multithreading** para permitir que os jogadores se movam simultaneamente, sem que a ação de um bloqueie o outro.

7.1. Lista de Threads Existentes

As threads do jogo são:

1. **Thread T1 (Player 1 Worker):** Controla o Jogador 1.
2. **Thread T2 (Player 2 Worker):** Controla o Jogador 2
3. Gerenciamento da interface (ncurses), **leitura de inputs** do teclado (`getch()`), **renderização** contínua do mapa (`draw_map()`), inicialização e sincronização final.

Propósito: Permitir que a lógica de cada jogador (leitura de input, cálculo de colisão, decisão de movimento) ocorra de forma independente.

Cenário sem Threads: Se o jogo rodasse em um único loop monolítico (sequencial), o Jogador 1 teria que se mover, depois o sistema esperaria o Jogador 2 se mover. Isso criaria um jogo baseado em turnos, eliminando a fluidez e a competição de velocidade em tempo real.

A utilização de threads resolve os seguintes problemas:

1. **Ações Simultâneas:** Permite que ambos os jogadores enviem comandos de movimento no mesmo milissegundo sem que um bloqueie o outro.
2. **Independência de Processamento:** Se um jogador ficar preso esperando um recurso (a ponte), a thread do outro jogador continua rodando livremente, mantendo o jogo vivo.

7.2. O Papel de Cada Thread

- As threads **T1 e T2** monitoram constantemente se há um comando de movimento pendente para seus respectivos jogadores. Elas executam a movimentação e "dormem" (sleep) brevemente para definir a velocidade do jogo. Isso permite a sensação de "Tempo Real".
- A thread **Main** fica responsável exclusivamente por ler o teclado (input) e redesenhar a tela (output), garantindo que a interface não trave enquanto a lógica é processada.

7.3. THREADS - Como foram utilizadas

Utilizamos `std::thread` da biblioteca padrão do C++, que abstrai `pthread_create`.

- **Código:** `thread t1(thread_player, ref(p1));`
- **Parâmetros:** Passamos a função `thread_player` e a referência do objeto `p1` ou `p2` que a thread deve controlar, permitindo que a mesma função lógica controle entidades diferentes.

7.3.1 Como funcionam internamente

A função `thread_player` executa um loop `while(playing)`:

1. **Verificação:** Checa se há uma direção definida (`p.direction`).
2. **Ação:** Se houver, chama `move_player()` para tentar atualizar a posição.
3. **Sleep:** Executa `this_thread::sleep_for(...)`. Isso libera a CPU voluntariamente, permitindo que a outra thread execute (troca de contexto).

7.4. Encerramento

O encerramento seguro é feito via variável de controle global `playing`.

Sinalização: Uma variável atômica/booleana compartilhada (`playing`) serve como flag de parada. Quando um jogador vence, `playing` vira `false`.

Join: A thread principal executa `t1.join()` e `t2.join()`. Isso bloqueia a `main` até que as threads dos jogadores terminem suas tarefas, garantindo que a memória não seja liberada enquanto as threads ainda tentam acessá-la (evitando *Segmentation Fault*).

8. SEMÁFOROS - O porquê foram utilizados

O jogo implementa um semáforo chamado `sem_RC`.

Os semáforos são a peça central da sincronização lógica do jogo, resolvendo o problema de acesso à ponte.

Esse é o problema do **Acesso a Recurso Limitado**.

- O mapa possui uma ponte estreita (marcada como `C`). Pela lógica do jogo, a ponte não suporta dois jogadores ao mesmo tempo.
- Sem controle, as duas threads poderiam entrar na coordenada `(x,y)` da ponte simultaneamente, causando sobreposição gráfica e inconsistência lógica.

8.1. Lista de Semáforos

- **Nome:** sem_RC
- **Tipo:** Semáforo Binário (Inicializado com 1).
- **Objetivo:** Controlar o acesso à Região Crítica Lógica (a Ponte).

8.2. O Problema Resolvido

A ponte no mapa é representada pelos caracteres C. A regra do jogo diz que a ponte é estreita demais para dois jogadores. Sem o semáforo, os dois jogadores poderiam entrar na ponte ao mesmo tempo, sobrescrevendo-se visualmente e violando a lógica do jogo.

8.3. SEMÁFOROS - Como foram utilizados

Eles são inicializados na main em:

sem_init(&sem_RC, 0, 1);

Onde **valor 1** indica que existe 1 vaga disponível na ponte (Recurso Livre).

As chamadas ocorrem dentro da função `move_player()`, que é a região onde a lógica de posição é alterada.

- **Entrada (Wait):** Ao detectar que o próximo passo é uma célula `C` (Ponte), executamos `sem_trywait(&sem_RC)`. Escolhemos `trywait` (não bloqueante) para que o jogo não congele; se a ponte está cheia, o jogador apenas não move naquele frame.
- **Saída (Post):** Ao detectar que o jogador estava em `C` e moveu para um chão comum, executamos `sem_post(&sem_RC)`.

8.4. Sincronização

O semáforo garante a **Exclusão Mútua** na ponte. Se T1 está na ponte, o semáforo vale 0. Se T2 tentar entrar, sem_trywait falha e T2 é impedido de avançar até que T1 saia e incremente o semáforo para 1 novamente.

9. Exemplos de Trechos de Código Comentados

Criação das Threads (Paralelismo):

```
// Cria thread para o Player 1 e inicia a execução da função thread_player
thread t1(thread_player, ref(p1));
// O jogo passa a processar P1 independentemente do Main e de P2.
thread t2(thread_player, ref(p2));
```

Uso de Mutex (Proteção de Dados Visuais):

```
// Adquire o lock do mutex mtx_map; Protege a escrita na matriz compartilhada.
unique_lock<mutex> lock(mtx_map);
// Desenha jogador na nova posição
map_view[p.x][p.y] = p.symbol;
lock.unlock(); // Libera o mutex
```

Uso de Semáforo (Região Crítica da Ponte):

```
// Se tenta entrar na ponte (C) e não está nela:
if (next_base == 'C' && current_base != 'C') {
    // Tenta decrementar. Se != 0, a ponte está cheia e o jogador não entra.
    if (sem_trywait(&sem_RC) != 0) return;
}
// ... movimenta o jogador ...
if (just_exited_critical) {
    sem_post(&sem_RC); // Libera a ponte para o próximo
}
```

10. Problemas de Concorrência Evitados

1. **Condição de Corrida de Dados (Data Race):** Evitada pelo uso de std::mutex (mtx_map). Sem isso, uma thread poderia ler o mapa para

desenhar enquanto a outra está escrevendo, causando "flickering" ou caracteres errados na tela.

2. **Inconsistência Lógica:** Evitada pelo `sem_t` (`sem_RC`). Sem isso, dois jogadores atravessariam a ponte juntos, o que é fisicamente impossível no design do jogo.
3. **Deadlock:** Evitado pelo uso de `sem_trywait` em vez de `sem_wait`. Se a ponte está ocupada, a thread não bloqueia indefinidamente segurando outros recursos (como o mutex); ela simplesmente desiste do movimento naquele frame e tenta novamente depois.

11. Testes Realizados

O jogo foi testado nos seguintes cenários para validar a concorrência:

- **Movimento Simultâneo:** Ambos os jogadores movendo-se livremente no mapa aberto. As threads responderam sem travar.
- **Teste de Bloqueio da Ponte:** O Jogador 1 entrou na ponte e parou. O Jogador 2 tentou entrar e foi corretamente bloqueado na entrada até que o Jogador 1 saísse.
- **Vitória:** Verificado se a condição de vitória encerrava corretamente ambas as threads e o programa.