

ICMC - Instituto de Ciências Matemáticas e da Computação

Disciplina: SSC0140 - Sistemas Operacionais I

Trabalho Final

Aluno: Alec Campos Aoki (15436800)

Aluno: Gabriel Phelipe Prado (15453730)

Aluno: Henrique Vieira Lima (15459372)

Aluno: Pedro Augusto Ferraro Paffaro (15483380)

Trabalho Final

1 Jogo de Sobrevivência

1.1 Introdução

O projeto tem como principal objetivo aplicar e demonstrar o uso de threads e semáforos no contexto de um jogo, lidando com conceitos como concorrência e sincronização de processos em sistemas operacionais.

O jogo consiste de um mapa 2D (10 x 20) no qual um jogador deve se movimentar e evitar colidir com zumbis que andam pelo mapa. O jogador e o zumbi operam em tempo real, controlados por múltiplas threads operando cada elemento concorrentemente.

2 Instalação e Execução

2.1 Requisitos

- Compilador: GCC
- Bibliotecas relevantes:
 - `pthread`: threads POSIX
 - `semaphore.h`: semáforos POSIX
 - `stdatomic.h`: operações atômicas
 - `math.h`: operações matemáticas

2.2 Compilação

Execute o seguinte comando no terminal:

```
gcc -o zombies zombies.c -lpthread -lm
```

- `-o zombies`: nome do executável
- `-lpthread`: link para a biblioteca de threads
- `-lm`: link para a biblioteca math.h

Ou rode:

```
make all
```

2.3 Execução

Para inicializar o jogo, execute o seguinte comando no terminal:

```
./zombies
```

Ou execute:

```
make run
```

3 O Jogo

3.1 Objetivo

O jogador deve sobreviver o maior tempo possível sem colidir com zumbis. Cada segundo sobre-vivido equivale a 10 pontos.

O jogo termina quando o personagem é tocado por algum zumbi, a não ser que esteja sob o efeito de invencibilidade do power up.

3.2 Controles

| Tecla | Função |
|-------|---------------------|
| W | Mover para cima |
| A | Mover para esquerda |
| S | Mover para baixo |
| D | Mover para direita |

3.3 Elementos

| Símbolo | Descrição |
|---------|-----------------------------------|
| | Jogador |
| | Zumbi |
| | Power-up (invencibilidade por 3s) |
| | Espaço livre |

4 Implementações

4.1 Threads (Concorrência)

O sistema usa 10 threads executando de forma concorrente:

| Thread | Qtd. | Responsabilidade |
|-----------------|--------|---|
| Principal | 1 | Captura entrada do usuário e renderiza o mapa |
| Zumbis | 6 | Movimentam cada zumbi individualmente |
| Spawn power ups | 1 | Gera power ups periodicamente |
| Pontuação | 1 | Incrementa a pontuação a cada segundo |
| Power up | 0 ou 1 | Controla duração da invencibilidade |

4.1.1 Criação das Threads

Threads dos Zumbis

```
pthread_t th_z[NUM_ZUMBIS];
for (int i = 0; i < NUM_ZUMBIS; i++){
    pthread_create(&th_z[i], NULL, thread_zumbi, &zumbis[i]);
}
```

São criadas 6 threads, uma para cada zumbi; cada thread recebe como argumento o ponteiro para sua struct Zumbi.

Todas executam a função `thread_zumbi()` de forma independente, de forma que cada zumbi se forme autonomamente usando um algoritmo simples de exploração aleatória com perseguição.

O uso de threads permite que os zumbis se movam simultaneamente (se fosse uma thread só para todos os zumbis, enquanto um se move todos os outros ficariam congelados).

Thread de Spawn de Power Ups

```
pthread_t th_spawn;
pthread_create(&th_spawn, NULL, thread_spawn_powerups, NULL);
```

Essa thread executa continuamente em segundo plano, tentando criar um power up no mapa a cada 1 a 3 segundos caso não haja nenhum power up ativo.

Usamos threads aqui para que os power ups spawnem independentemente das ações do jogador e da lógica do loop principal.

Thread de Pontuação

```
pthread_t thPontos;
pthread_create(&thPontos, NULL, threadPontuacao, NULL);
```

Incrementa a pontuação em 10 a cada um segundo que o jogador sobrevive, utilizando `atomic_fetch_add()` pra garantir a segurança da variável.

Usamos thread aqui para que o incremento ocorra independentemente da taxa de atualização do loop principal.

Thread de Power Up

```
pthread_t t;
pthread_create(&t, NULL, threadPowerUp, NULL);
pthread_detach(t);
```

É criada dinamicamente quando o jogador coleta um power up, ativando a invencibilidade por 3 segundos e desativando via `pthread_detach()`.

Usamos threads aqui para que o efeito do power up possa ser controlado independentemente e não bloquear o jogo durante o efeito de invencibilidade.

4.1.2 Sincronização das Threads

Esperamos as threads sincronizar quando o jogo acaba para encerrarmos o programa.

```
for (int i = 0; i < NUM_ZUMBIS; i++){
    pthread_join(th_z[i], NULL);
}
pthread_join(th_spawn, NULL);
pthread_join(th_pontos, NULL);
```

A thread principal aguarda cada thread terminar usando `pthread_join()` e só após todas finalizarem limpamos os semáforos.

Fazemos isso para não haver threads de zumbis no sistema, evitando condições de corrida na limpeza de recursos.

5 Semáforos

Usamos 4 semáforos, todos inicializados com valor 1 (mutex binário).

| Semáforo | Função | Tipo |
|-----------|-------------------------------------|-------|
| sem_mapa | Controle do mapa | Mutex |
| sem_spawn | Controle dos power ups | Mutex |
| sem_power | Exclusividade do efeito do power up | Token |
| sem_state | Protege variáveis de estado | Mutex |

5.1 Criação

```
sem_unlink("/sem_mapa");
sem_unlink("/sem_spawn");
sem_unlink("/sem_power");
sem_unlink("/sem_state");

sem_mapa = sem_open("/sem_mapa", O_CREAT, 0644, 1);
sem_spawn = sem_open("/sem_spawn", O_CREAT, 0644, 1);
sem_power = sem_open("/sem_power", O_CREAT, 0644, 1);
sem_state = sem_open("/sem_state", O_CREAT, 0644, 1);
```

5.2 Uso dos Semáforos

5.2.1 sem_mapa

Usamos um semáforo para garantir exclusão mútua no acesso ao mapa, evitando condições de corrida entre as threads do jogador, dos zumbis e do spawn de power ups, que poderiam acabar ocupando a mesma posição na matriz.

Protegemos o mapa durante sua renderização, evitando que ele seja modificado enquanto é lido:

```
void desenhar_mapa(){
    sem_wait(sem_mapa); // Entra na regiao critica

    for (int i = 0; i < ALTURA; i++){
        for (int j = 0; j < LARGURA; j++){
            // Renderiza cada posicao do mapa
            switch(mapa[i][j]){ ... }
        }
    }

    sem_post(sem_mapa); // Sai da regiao critica
}
```

Quando o jogador se move:

```
sem_wait(sem_mapa); // Protege leitura/escrita do mapa

// Coleta power-up se presente
if (mapa[novo_x][novo_y] == 'P'){ ... }

// Verifica colisao com zumbi
if (mapa[novo_x][novo_y] == 'Z' && !invencivel_local){ ... }

// Atualiza posicao
mapa[jogador_x][jogador_y] = '.';
jogador_x = novo_x;
jogador_y = novo_y;
mapa[jogador_x][jogador_y] = 'J';

sem_post(sem_mapa); // Libera o mapa
```

Quando um zumbi se move:

```
sem_wait(sem_mapa);
mapa[z->x][z->y] = '.'; // Limpa posicao antiga
z->x = nx;
z->y = ny;
mapa[z->x][z->y] = 'Z'; // Marca nova posicao
sem_post(sem_mapa);
```

5.2.2 sem_state

Usamos esse semáforo para proteger variáveis que são lidas por múltiplas threads, como game_over, invencivel e a posição do jogador.

Protegemos game_over na thread principal:

```
sem_wait(sem_state);
if (game_over){
    sem_post(sem_state);
```

```

        break;
}
sem_post(sem_state);

```

As coordenadas do jogador e invencivel no caso de movimento e/ou colisões:

```

sem_wait(sem_state);
if (nx == jogador_x && ny == jogador_y && !invencivel){
    game_over = 1;
}
sem_post(sem_state);

```

```

sem_wait(sem_state);
if (nx == jogador_x && ny == jogador_y && !invencivel){
    game_over = 1;
}
sem_post(sem_state);

```

```

sem_wait(sem_state);
invencivel = 1; // Ativa efeito
sem_post(sem_state);

sleep(3);

sem_wait(sem_state);
invencivel = 0; // Desativa
sem_post(sem_state);

```

5.2.3 sem_spawn

Semáforo para proteger as variáveis power_ativo_mapa, power_x e power_y.

Usamos para garantir que a thread de spawn não cria um novo power up no mesmo instante da coleta:

```

if (mapa[novo_x][novo_y] == 'P'){
    sem_wait(sem_spawn);
    power_ativo_mapa = 0; // Remove indicação de power ativo
    sem_post(sem_spawn);
}

```

Ou que exista mais de um power up no mapa ao mesmo tempo:

```

sem_wait(sem_spawn); // Protege o estado do power up

int inv_local;
sem_wait(sem_state);
inv_local = invencivel;
sem_post(sem_state);

if (!power_ativo_mapa && !inv_local){

```

```
// ... encontra posição válida ...

power_x = x;
power_y = y;
power_ativo_mapa = 1; // Marca como ativo

sem_wait(sem_mapa);
mapa[x][y] = 'P';
sem_post(sem_mapa);

}

sem_post(sem_spawn);
```

5.2.4 sem_power

Semáforo para garantir que somente um power up está acionado por vez, evitando que duas invencibilidades de sobreponham e causem bugs.

Usamos um padrão de token único, onde sem_trywait() tenta decrementar o semáforo sem bloquá-lo:

```
if (sem_trywait(sem_power) == 0){
    pthread_t t;
    pthread_create(&t, NULL, thread_power_timer, NULL);
    pthread_detach(t);
}
```

5.3 Limpeza

```
sem_close(sem_mapa);
sem_unlink("/sem_mapa");
sem_close(sem_spawn);
sem_unlink("/sem_spawn");
sem_close(sem_power);
sem_unlink("/sem_power");
sem_close(sem_state);
sem_unlink("/sem_state");
```

sem_close() fecha o descritor do somáforo no processo atual enquanto sem_unlink() remove o semáforo do sistema operacional, evitando leaks ou requícios nos semáforos.

6 Operações Atômicas

```
atomic_int pontos = 0;
```

```
atomic_fetch_add(&pontos, 10);
```

6 OPERAÇÕES ATÔMICAS

Utilizamos operações atômicas (indivisíveis) na pontuação para evitar condições de corrida de forma mais simples que usando semáforos.