DEPARTAMENTO DE ENGENHARIA INFORMATICA E SISTEMAS

UNIDADE CURRICULAR DE SISTEMAS OPERATIVOS II

SPACE
INVADERS

META №1

TAP TO PLAY

PLAYERS:

JOAO FERREIRA, 21220114

MARCO AGANTE, 21200692

GAME LAST SAVED ON:

MAIO DE 2018

Índice

1. Introdução

No âmbito da primeira meta do trabalho prático da unidade curricular de Sistemas Operativos II, apresentamos este breve relatório onde explicamos as escolhas, utilidade e implementações relativamente à memória partilhada, estruturas de dados e todos os aspetos relativos à sincronização.

O trabalho é composto por 4 projetos, sendo que apenas 3 deles possuem implementações: gateway, servidor e DlltpSO2.

2. DLL e Estruturas de dados

Seguindo o solicitado no enunciado, foi criada uma DLL responsável pelos dados e funções para gerir a zona de mensagens da memória partilhada, e a partir desta são usados no *gateway* e no servidor. Uma vez que também estamos a utilizar memória partilhada para o Jogo entre o servidor e o *gateway*, decidimos aproveitar para colocar também as estruturas do Jogo na DLL.

Uma vez que as funcionalidades existentes na DLL estarão em execução constante, decidimos utilizar a DLL de forma implícita.

2.1. Estruturas Mensagens e Jogo (na DLL)

A DLL começa por apresentar diversos *defines* que serão utilizados para iniciar diversos objetos, que existem tanto no *buffer* circular como no Jogo.

De seguida podemos ver diversos *HANDLEs* que irão ser utilizados para a sincronização e que serão explicados mais à frente. A DLL também detém vários *typedef enum* que são utilizados para simplificar definições das estruturas seguintes.

Finalmente, deparamo-nos com as estruturas relativas às mensagens. Existem 5 estruturas que estão associadas às mensagens, sendo que duas são para lidar com as mensagens inseridas pelo *gateway* no *buffer* e outras três para enviar as atualizações de jogo para o cliente. Estas são:

- typedef struct _MsgCLI - Esta estrutura será utilizada para definir o tipo de mensagem enviada pelo cliente para o gateway que por sua vez será colocada no *buffer*;
- typedef struct _BufferMensagens – Estrutura utilizada para definir o *buffer*, que por sua vez é composta por um *array* de MsgCli e pelos índices para percorrer o *buffer*;
- typedef struct _MsgPosicoesJogo– Estrutura semelhante ao Jogo, mas apenas com um item de cada estrutura relacionada com o mesmo. Será utilizada para organizar apenas informações relativas às atualizações que serão enviadas para o cliente;
- typedef struct _MsgSER – Estrutura utilizada para enviar as mensagens com as

informações de jogo do servidor;

- typedef struct Pontuacao – Estrutura que guarda a pontuação dos jogadores para a apresentação do TOP10 no final do jogo.

Como referido anteriormente, também as estruturas referentes ao Jogo foram criadas na DLL. Todas as seguintes estruturas de dados são uteis e relevantes para a inicialização e funcionamento de toda a lógica do "*Space Invaders*".

- typedef struct Area – Como praticamente todos os componentes do jogo são retângulos, decidimos ter uma estrutura que agrupa as características dos mesmos;
- typedef struct Invader – Estrutura responsável por armazenar informação relativa aos inimigos. Fundamental para a criação das *Threads* relativas ao controlo dos tipos de *Invaders*, das quais vamos falar mais à frente;
- typedef struct PowerUP – Estrutura que guarda informação dos *PowerUps*. De referir a variável duração, que influência o tempo de vida do *powerup*;
- typedef struct Defender – Estrutura mais complexa responsável por armazenar informações relativas aos Jogadores. É constituída por algumas das outras estruturas do jogo, como por exemplo *Pontuacao*, *PowerUP* e *Area*;
- typedef struct Disparos – Estrutura que representa as bombas e os tiros, uma vez que estes têm características idênticas. Existe um campo direção do tipo booleano, no qual as bombas serão *FALSE* (descer) e tiros *TRUE* (subir);
- typedef struct _Jogo – Estrutura que enverga o Jogo como um todo. Tendo em isso em conta, é constituída por *array's* de praticamente todas as estruturas que falámos anteriormente. A adicionar a isso temos também o Ciclo de vida.

2.2. Servidor

No projeto do servidor, temos apenas uma estrutura definida.

- typedef struct _Input – Estrutura utilizada para armazenar os valores que são configuráveis para o servidor.

Como apoio à verificação destes *Inputs* e de diversos mecanismos do Jogo, desde velocidade de itens até à probabilidade de tiros por parte dos Inimigos, existem múltiplos

#define. Estes *#define's* são em grande parte agrupados em 3, no qual apresentamos um número máximo, mínimo e *default*. Estes parâmetros ainda não foram acertados com valores mais lógicos, uma vez que ainda não temos o jogo totalmente funcional para testar os mesmos.

3. Memória partilhada

No trabalho existem dois componentes que têm de pertencer à memória partilhada, um *buffer* circular e o Jogo. Com isto em mente, decidimos ter dois blocos de memória partilhada diferentes, um para o *buffer* e outro para o Jogo.

Para tal inicializámos um Ponteiro para cada um destes na DLL:

- PBufferMensagens mensagens = NULL;
- PJogo jogo = NULL;

Também para cada um deles é reservado um bloco de memória com o tamanho máximo possível a cada um deles. É posteriormente feito o mapeamento de ambos. As questões relativas à sincronização são referidas no tópico seguinte.

4. Sincronização

Existem dois pontos críticos onde a sincronização tem um papel fundamental no bom funcionamento de todo o projeto. Estes são o *Buffer* Circular e na atualização do jogo.

4.1. Buffer Circular (Gateway -> Servidor)

Deparamo-nos aqui com um cenário no qual teremos diversas *Threads* a simultaneamente escrever no *buffer*, enquanto existe uma só a ler as mensagens do mesmo. Com o uso de dois Semáforos e dois Mutexs, conseguimos salvaguardar que nunca pomos em causa a integridade dos dados. Um dos semáforos começa com o número máximo de mensagens que o buffer pode ter, e o outro com o mínimo.

A utilização destes mecanismos pode ser verificada nas seguintes funções da DLL:

void EnviaMensagem() e void TrataMensagem().

Para verificar que este *buffer* estava funcional, utilizamos as funções anteriores que na prática se traduzem na impressão no ecrã do servidor as mensagens que são inseridas no gateway.

4.2. Atualização do jogo (Servidor->Gateway)

A outra situação na qual existem problemas de sincronização é quando pretendemos informar o *Gateway* que existiram mudanças no jogo. Foram utilizados dois *Mutex* e dois eventos para salvaguarda este problema. Um dos eventos será utilizado para informar o *Gateway* que existe uma nova alteração para ir ler, e o outro para informar o Servidor que já pode voltar a fazer alterações na memoria partilhada. Os *mutexes* são utilizados para assegurar a integridade dos dados.

A utilização destes mecanismos pode ser verificada nas seguintes funções da DLL: `void MoveInvaderBase(int id, int x, int y, int num)` e `void RecebeAtualizacao(int id)`.

Para verificar que todas as funcionalidades da memória partilhada relativa ao Jogo estavam bem implementadas, utilizámos um mecanismo intermédio e estritamente utilizado para testes nesta primeira meta. Decidimos criar uma função que produz um movimento aleatório e associar a mesma ao invader que se encontra na posição 10 do array de *Invaders* no Jogo (`jogo->Invaders[10].id_invader`). Assim, cada vez que o servidor move o *Invader* para a nova posição, são desencadeados todos os mecanismos de sincronização e desta forma essa mudança de posição é sinalizada ao *gateway*, que imprime as novas posições no ecrã.

5. Funções e Threads

5.1. Funções DLL

Na DLL temos 6 funções:

- `void IniciaSinc()` – Efetua a criação de todos os mecanismos de sincronização que vão ser utilizados no projeto e verifica se a mesma existiu sem erros;
- `void AcabaSinc()` – Garante que os mecanismos inicializados anteriormente são terminados corretamente;

As funções seguintes, ainda não estão a funcionar a 100% uma vez que a implementação de outras funcionalidades das quais estas dependem ainda não estão realizadas. Sendo assim, para o *buffer*, fez-se um mecanismo de *input-output* para testar que a troca de mensagens está funcional. Para o jogo, imprimiu-se no ecrã do *gateway* as mudanças de posição geradas no servidor.

- void EnviaMensagem() – Função utilizada pela *Thread* que será criada no *Gateway* para enviar mensagens para o *buffer*. Aqui podemos observar os mecanismos de sincronização explicados anteriormente, em funcionamento;
- void TrataMensagem() – Função utilizada pela *Thread* que será criada no Servidor para receber as mensagens que estão no *buffer*. A sincronização vai de acordo à anterior, garantindo a troca de mensagens de forma correta.
- void MoveInvaderBase(int id, int x, int y, int num) - Função utilizada pela *Thread* que faz a gestão dos Invaders do tipo Base, para provocar uma mudança de posição aleatoriamente. Realçar novamente que esta função é usada meramente por motivos de teste.
- void RecebeAtualizacao(int id)- Função que recebe o id de um *invader* e vai imprimir as coordenadas no jogo desse mesmo *invader*.

Estas duas funções relativas ao Jogo, são desencadeadas através dos mecanismos de sincronização que descrevemos anteriormente. Estes garantem que nunca existirá conflito na utilização do jogo, e assim garantir o correto funcionamento do mesmo.

5.2. Funções e Threads Servidor

No Servidor temos já diversas funcionalidades implementadas. No que toca às funções, temos algumas de inicialização e outra para receber as informações configuráveis no servidor.

- void InicializaJogo() – Ainda não está totalmente configurada, mas será utilizada para inicializar todos os componentes do jogo;
- void IniciaInvaders(Input inp) – Os *Invaders* como componente fundamental do jogo, e como necessário para esta primeira meta, necessitam de uma função que os inicie;
- void ColocaInvaders(Input inp) – Como parte da inicialização, esta função é um algoritmo para distribuir os inimigos pelo mapa;
- int CreateThreadsInvaders() – Função utilizada para lançar as *Threads* relativas aos inimigos;
- Input RecebeInput() – Simplesmente recebe parâmetros que são introduzidos no servidor.

Apresentamos também a implementação de diversas *Threads*:

- DWORD WINAPI ThreadConsumidor(LPVOID param) – *Thread* que vai ser responsável por ler mensagens do buffer. Esta vai chamar a função TrataMensagem() e toda a lógica incluída na mesma;

As seguintes *Threads* são as responsáveis pelo controlo das naves inimigas. Temos uma *Thread* por tipo de nave. Ainda não definimos todos os detalhes dos Invaders extra, como tal inicializámos todos os dados relativas às mesmas a 0, e excluímo-las, para já, da implementação. Apesar de ainda não terem sido implementadas, a lógica já está definida (inclusive em comentários no código enviado em anexo com este relatório). Esta lógica será semelhante para todo o tipo de *Threads*, apenas mudando algumas características como por exemplo no algoritmo de movimento delas.

Indo ao encontro do descrito anteriormente, acabámos por adicionar algum movimento simples na *ThreadInvadersBase* de forma a mostrarmos o funcionamento de todos os mecanismos que implementámos para esta primeira meta:

- DWORD WINAPI ThreadInvadersBase(Input inp)
- DWORD WINAPI ThreadInvadersEsquivo(Input inp)
- DWORD WINAPI ThreadInvadersExtra(Input inp)

5.3. Funções e Threads Gateway

De momento a única funcionalidade que temos na *Gateway* é a criação das *Threads* que irão escrever no *buffer*.

- DWORD WINAPI ThreadProdutor(LPVOID param) – Simplesmente vai chamar a função EnviaMensagem() de forma a colocar mensagens no *buffer* circular.
- DWORD WINAPI ThreadAtualizacao(LPVOID param) – *Thread* que vai chamar a função RecebeAtualização(); de forma que um determinado invader receba a atualização de coordenadas no mapa.