
# Trabalho Prático Sistemas Operativos 2

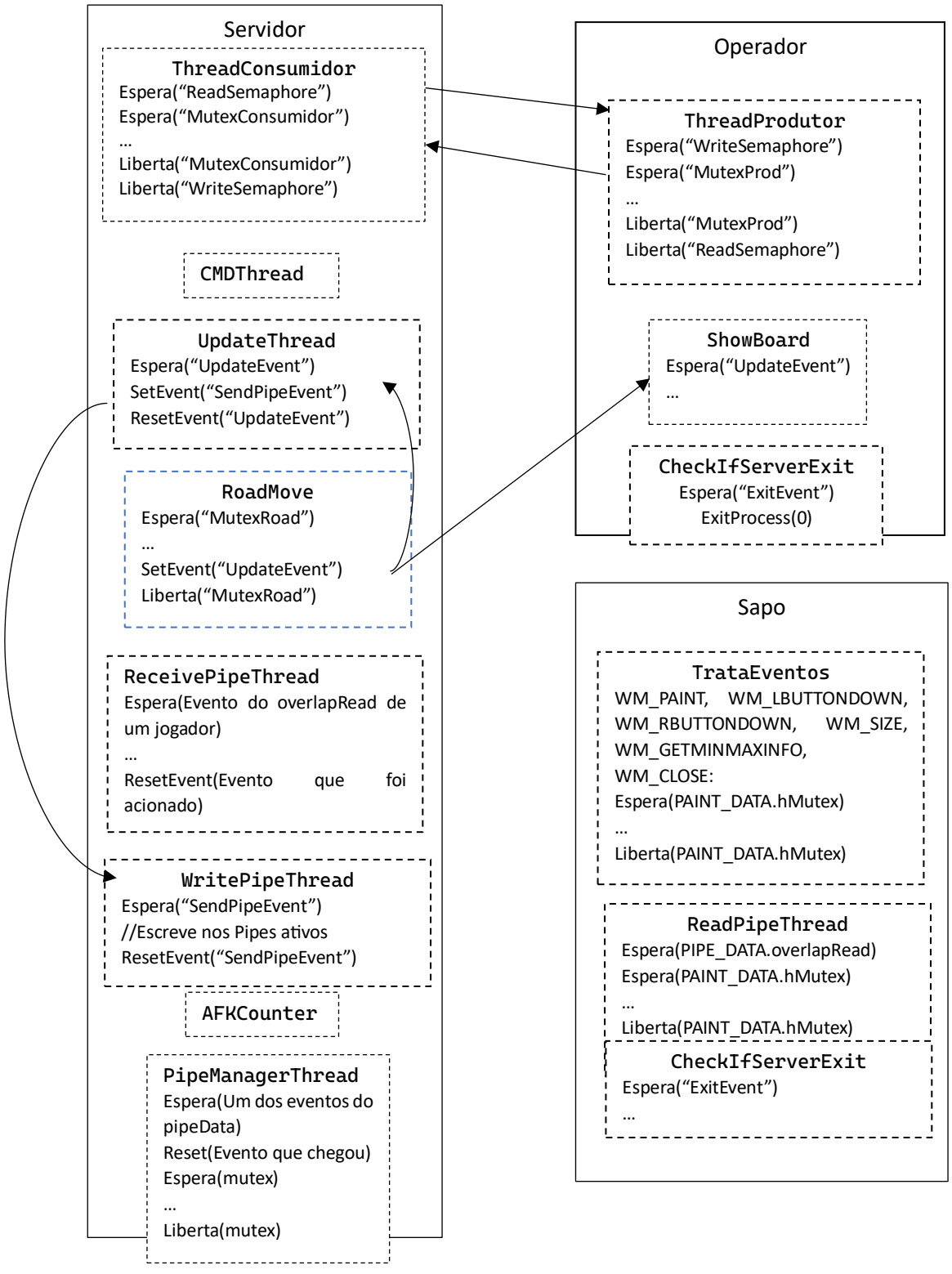
## Meta 2

Daniela Correia – 2021143404

Tiago Cardoso – 2021138999

# Índice

# Diagrama/Esquema



| Eventos | |
|---|---|
| “UpdateEvent” | Avisa a thread UpdateThread que há alterações a fazer no tabuleiro, por exemplo os carros moveram-se, logo o conjunto de caracteres que representa o jogo tem de ser atualizado.<br><br>Avisa o operador que há uma alteração no jogo e que deve ser mostrado o novo estado do jogo. |
| “ExitEvent” | Avisa o operador e o jogador que o servidor foi desligado. |
| **Semáforos** | |
| “ReadSemaphore” | Serve para avisar o servidor que á uma posição para ler no buffer circular |
| “Write Semaphore” | Serve para avisar o operador que pode escrever noutra posição do buffer circular |
| **Mutex** | |
| “MutexConsumidor” | Serve para haver sincronismo no buffer circular no lado do servidor. |
| “MutexProd” | Serve para os operadores não escreverem em posições sobrepostas. |
| “MutexRoad” | Mantém o sincronismo no movimento entre as estradas durante o seu movimento |
| “Server” | Permite que corra um e um só servidor |
| PAINT_DATA.hMutex | Permite um sincronismo na ação de preencher a tela. |

| Threads | | |
|---|---|---|
| Servidor | Operador | Sapo |
| **ThreadConsumidor:** Lê e interpreta os comandos que os operadores escrevem, contendo todo o sincronismo de leitura do buffer circular dado nas aulas. | **ThreadProdutor:** Permite que o operador escreva comandos que irão constituir uma posição do buffer circular. Esta thread contém todo o sincronismo de escrita do buffer circular dado nas aulas. | **ReadPipeThread:** Fica a espera que receba atualizações por parte do servidor e ao receber preenche o ecrã com o estado do jogo atualizado. |
| **CMDThread:** Aqui são inseridos e executados comandos escritos no servidor, sendo efetuadas as devidas alterações no jogo. | **ShowBoard:** É esperado que o evento "UpdateEvent" seja assinalado para depois ser mostrado o novo estado do jogo, no final é feito o ResetEvent do "UpdateEvent". | **CheckIfServerExit:** Fica a espera que o evento "ExitEvent" seja assinalado e com isso o jogador encerra e é avisado que o servidor o desconectou. |
| **UpdateThread:** É esperado o evento "UpdateEvent", que é assinalado na thread RoadMove, para que o jogo seja atualizado. | **CheckIfServerExit:** É esperado que o evento "ExitEvent" seja assinalado e com isso o operador é encerrado. O evento referido é inicializado quando o servidor é executado. | |
| **RoadMove:** É esperado o mutex "MutexRoad" para sincronizar o movimento de cada estrada, e assinalar o evento "UpdateEvent", desta maneira avisa que houve alterações no jogo. No final é libertado o "MutexRoad". | | |

| | | |
|---|---|---|
| **AFKCounter:** Conta o número de segundos que cada jogador que esteja ativo não se movimenta e ao passar dos 10 segundos o jogador é colocado novamente na zona de partida. | | |
| **ReceivePipeThread:** Fica à espera que um dos eventos correspondentes à estrutura Game.pipeData.overlapRead.hEvent seja acionado e procede à leitura do pipe, executando qualquer que seja a informação que tenha sido recebida, e no fim faz o reset desse mesmo evento. | | |
| **WritePipeThread:** Fica a espera que o evento "SendPipeEvent" seja acionado e procede ao envio do estado do jogo atual a cada jogador que esteja ativo. Envia o que tem a enviar e no fim faz o reset desse evento. | | |
| **PipeManagerThread:** Fica à espera que um dos eventos do game.pipeData.hEvents seja acionado e caso tenha sido conectado um novo jogador faz o reset a esse evento e se o numero de bytes recebidos for igual a 0 espera-se pelo mutex game.pipeData.hMutex inicializa-se a informação do novo jogador e liberta-se esse mutex. Se o jogo acabar são desconectados todos os clientes. | | |

# Estruturas Memória Partilhada

**CELULA_BUFFFER** – Estrutura que representa o nosso buffer circular.

```c
//estrutura para o buffer circular
typedef struct CELULA_BUFFER {
    int id;
    TCHAR str[100];
}CELULA_BUFFER;
```

**BUFFER_CIRCULAR** – estrutura para auxílio ao uso do buffer circular.

```c
//representa a nossa memoria partilhada
typedef struct BUFFER_CIRCULAR {
    int nProdutores;
    int nConsumidores;
    int posE; //proxima posicao de escrita
    int posL; //proxima posicao de leitura
    CELULA_BUFFER buffer[TAM_BUF]; //buffer circular em si (array de estruturas)
}BUFFER_CIRCULAR;
```

**SHARED_BOARD** – estrutura que contém o mapa de jogo assim como as suas dimensões.

```c
typedef struct SHARED_BOARD {
    DWORD dwWidth;
    DWORD dwHeight;

    TCHAR board[MAX_ROADS + 4][MAX_WIDTH];
}SHARED_BOARD;
```

**SHARED_MEMORY** e **SHARED_DATA** – estruturas para apoio ao uso de memória partilhada e para o uso de sincronização.

```c
typedef struct SHARED_MEMORY {
    BUFFER_CIRCULAR bufferCircular;
    SHARED_BOARD sharedBoard;
}SHARED_MEMORY;

//estrutura de apoio
typedef struct SHARED_DATA {
    SHARED_MEMORY* memPar; //ponteiro para a memoria partilhada
    HANDLE hSemEscrita; //handle para o semaforo que controla as escritas (controla quantas posicoes estao vazias)
    HANDLE hSemLeitura; //handle para o semaforo que controla as leituras (controla quantas posicoes estao preenchidas)
    HANDLE hMutex;
    int terminar; // 1 para sair, 0 em caso contrario
    int id;
}SHARED_DATA;
```

## Estruturas de funcionamento do jogo

**OBJECT** – estrutura que representa um objeto (sapo, carro, obstáculo).

```
//diração a seguir
typedef enum WAY { UP, DOWN, LEFT, RIGHT, STOP }WAY;
typedef struct OBJECT {
    DWORD dwX, dwY;
    TCHAR c;
}OBJECT;
```

**ROAD** – estrutura que representa uma faixa de rodagem que pode conter obstáculos (*objects*) e carros (*cars*) assim como tem um sentido (*way*) e variáveis como o número de carros que nela circulam e etc...

```
//dados de uma estrada
typedef struct ROAD {
    DWORD dwNumOfCars;
    DWORD dwNumOfObjects;
    DWORD dwSpaceBetween;
    DWORD dwSpeed;
    DWORD dwTimeStoped;
    BOOL bChanged;
    HANDLE hMutex;
    HANDLE hThread;

    WAY lastWay;
    WAY way;

    OBJECT cars[MAX_CARS_PER_ROAD];
    OBJECT objects[MAX_CARS_PER_ROAD];
}ROAD;
```

**GAME** – estrutura principal que serve de auxílio para todo o correr do jogo

```
//dados do jogo
typedef struct GAME {
    DWORD dwLevel;
    DWORD dwShutDown;
    DWORD dwInitSpeed, dwInitNumOfRoads;
    BOOL  bPaused;

    HANDLE hKey;

    SHARED_DATA sharedData;
    PIPE_DATA pipeData;
    ROAD roads[MAX_ROADS];
}GAME;
```

## Estruturas de comunicação named pipe

**PIPE_DATA** – estrutura com informação sobre o número de clientes conectados e com eventos e mutex para sincronização da comunicação.

```
//dados do pipe
typedef struct PIPE_DATA {
    PLAYER_DATA playerData[MAX_PLAYERS];
    HANDLE hEvents[MAX_PLAYERS];
    HANDLE hMutex;
    DWORD dwNumClients;
}PIPE_DATA;
```

**PLAYER_DATA** – estrutura com informação sobre o estado de cada cliente e com os meios necessários para cumprir com a comunicação.

```
//dados dos jogadores
typedef struct PLAYER_DATA {
    HANDLE hPipe;
    OVERLAPPED overlapRead,overlapWrite;
    BOOL active;
    OBJECT obj;
    DWORD dwPoints;
    DWORD dwNEndLevel;//numero de vezes que chegou ao fim
    DWORD dwAFKseg;// numero de segundos away from keyboard 'afk'
    GAME_TYPE gameType;
    BOOL bWaiting;
}PLAYER_DATA;
```

**PIPE_GAME_DATA** – estrutura a enviar para cada pipe, sendo atualizada dependendo do cliente com quem estamos a lidar.

```
//dados do jogo a enviar pelo pipe aos clientes ativos
typedef struct PIPE_GAME_DATA {
    SHARED_BOARD sharedBoard;
    DWORD dwLevel;
    DWORD dwPlayer1Points, dwPlayer2Points;
    DWORD dwX, dwY;
    DWORD dwNEndLevel;//numero de vezes que chegou ao fim
    GAME_TYPE gameType;
    BOOL bWaiting;
}PIPE_GAME_DATA;
```

**GAME_TYPE** – enumeração do tipo de jogo que o cliente se encontra.

```
//tipo de jogo a correr
typedef enum GAME_TYPE {
    SINGLE_PLAYER,MULTI_PLAYER,NONE
}GAME_TYPE;
```

# Funcionalidades

## Servidor

| ID | Descrição funcionalidade / requisito | Estado |
|---|---|---|
| 1 | O programa servidor é lançado pelo utilizador. Deve assegurar-se que ainda não estava a correr (se estiver, a nova instância termina, alertando o utilizador desse fato). | Implementado |
| 2 | O número de faixas de rodagem e a velocidade inicial dos carros são passados através da linha de comandos ou encontram-se definidos no Registry. | Implementado |
| 3 | Determina de forma aleatória a posição dos sapos. Devem estar localizados na área de partida e não podem estar sobrepostos. | Implementado |
| 4 | Recebe comandos do operador e desencadeia as ações necessárias. | Implementado |
| 5 | Aceita os jogadores que se ligam através do programa sapo. | Implementado |
| 6 | Recebe por parte dos clientes (programa sapo) os movimentos que pretendem efetuar e mantém atualizada toda a informação do jogo. | Implementado |
| 7 | Interface segundo o paradigma de consola, seguindo a lógica de comandos (não são menus). A interface deve permitir receber os seguintes comandos:<br>• Suspender e retomar o jogo.<br>• Reiniciar o jogo.<br>• Encerrar todo o sistema (todas as aplicações são notificadas). | Implementado |
| 8 | Se um jogador não efetuar qualquer movimento durante 10 segundos, o seu sapo volta para a zona de partida. Ao atingir a área de chegada, um jogador passa automaticamente para o próximo nível.<br>A cada novo nível, a velocidade e o número de carros aumentam. | Implementado |

## Operador

| ID | Descrição funcionalidade / requisito | Estado |
|---|---|---|
| 1 | O programa operador é lançado explicitamente pelo utilizador. Podem existir várias instâncias do operador em execução. | Implementado |
| 2 | Implementação dos comandos pedidos, parar os carros, alternar sentido da faixa, etc.. | Implementado |
| 3 | Comunicação com servidor através de memória partilhada | Implementado |
| 4 | Mostrar a informação do estado do jogo em tempo real (em modo de texto). | Implementado |
| 5 | Desencadear ações que alteram o funcionamento do jogo e comunicá-las ao servidor | Implementado |

## Sapo

| ID | Descrição funcionalidade / requisito | Estado |
|---|---|---|
| 1 | Implementação de duas modalidades de jogo, individual e competição e os seus requisitos. | Implementado |
| 2 | Interface gráfica Win32 que apresenta o jogo e toda a informação. Esta informação estará permanentemente visível e será atualizada em tempo real. | Implementado |
| 3 | Tratando-se da modalidade de jogo individual, o jogo começa de imediato. Na modalidade de competição, terá de aguardar a chegada de um adversário para se dar início ao jogo. | Implementado |
| 4 | No decurso do jogo o utilizador poderá utilizar as teclas de direção ou clicar em cima da posição para onde pretende movimentar o sapo. | Implementado |
| 5 | Interage com o servidor e com o utilizador. Este programa apenas comunica com o servidor, não existindo comunicação direta com o operador. | Implementado |
| 6 | A interface gráfica da aplicação sapo não deve cintilar. Deve-se utilizar a técnica de double buffering, abordada nas aulas | Implementado |
| 7 | Clique com o botão esquerdo do rato nas posições contíguas à localização do sapo, movem-no para essa posição. | Implementado |
| 8 | Clique com o botão direito do rato em cima do sapo permite reposicionar o sapo na zona de partida. | Implementado |
| 9 | Ao passar com o rato por cima do sapo mostra a quantidade de vezes que atravessou com sucesso a estrada. | Implementado |
| 10 | Utilização das teclas de direção para movimentar o sapo. | Implementado |
| 11 | O programa começa por solicitar ao utilizador o nome e o tipo de jogo que pretende. | Não Implementado |
| 12 | O menu principal da janela deve permitir alternar entre 2 conjuntos de bitmaps utilizados para representar os vários elementos do jogo (carros, sapos, etc.). | Não Implementado |