Instituto Superior
de Engenharia
Politécnico de Coimbra
Instituto Politécnico de Coimbra
Instituto Superior de Engenharia de Coimbra
Departamento de Engenharia Informática e Sistemas

Licenciatura em Engenharia Informática

Relatório do trabalho prático de Sistemas Operativos 2

Rafael Filipe Rodrigues Pereira
2022150534

Sebastian Ernesto Gonçalves Vigas
2023155905

Coimbra, Junho de 2025

Índice

1 Introdução

O presente relatório descreve o desenvolvimento do trabalho prático de Sistemas Operativos 2, cujo principal objetivo é aplicar e consolidar os conhecimentos abordados ao longo do semestre nesta unidade curricular.

Este projeto consiste em criar um jogo para adivinhar palavras. Conta com um árbitro, que irá coordenar o jogo, e múltiplos jogadores, que vão receber letras e tentar adivinhar palavras a partir destas. Estes jogadores podem ser humanos ou *bots*.

Foram colocados em prática os principais conceitos aprendidos para realizar o projeto, sendo estes *named pipes*, eventos, memória partilhada, etc.

Neste relatório será analisado todo o trabalho feito ao longo do projeto, bem como os seus resultados, desafios e demais aspetos.

2 Estrutura do trabalho

A primeira decisão sobre o trabalho prático foi sobre a sua organização e estrutura, bem como a de cada programa deste. Como será possível ver nos capítulos e subcapítulos seguintes, a adoção de uma estrutura mais modular permitiu alcançar as soluções utilizadas com mais facilidade. Facilitou também a gestão dos recursos, nomeadamente *named pipes*, *mutexes* e *threads*, os quais requerem especial atenção, pois os programas devem encerrar corretamente, libertando todos os recursos em uso. Este aspeto destaca-se muito bem no programa árbitro, pois é o maior e mais complexo dos quatro programas do projeto.

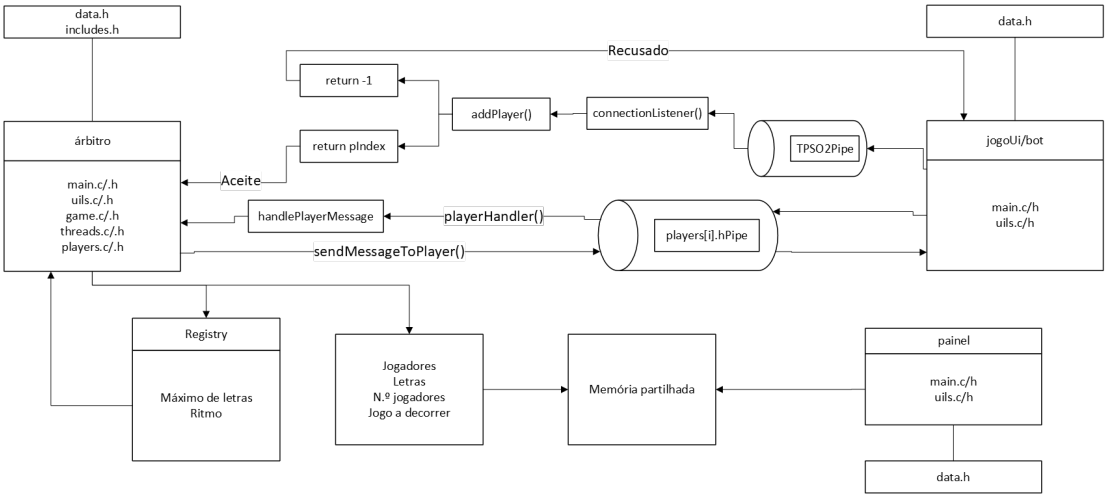

Figura 1: Estrutura base do projeto

A figura anterior (*Figura 1*) mostra a estrutura do projeto e como o mesmo funciona. Quando o árbitro inicia, é criada uma *thread connectionListener()*. Essa *thread* cria uma instância da *pipe "TPSO2Pipe"* e fica à espera de conexão. Quando um jogador se tenta conectar, a *thread* chama a função *addPlayer()* para fazer as devidas verificações, como verificar se o jogo está cheio ou se o nome de utilizador enviado já existe, e procede em adicionar ou recusar o jogador. Caso seja recusado, recebe uma mensagem com o motivo, a *thread* desconecta-se da *named pipe*, fecha a *handle* e o ciclo volta ao início, criando uma nova instância e ficando novamente à escuta. Caso seja aceite, as informações do jogador são guardadas na sua estrutura, sendo também armazenada a *handle* da instância atual da *pipe* e é adicionado ao *array* de jogadores. A função *addPlayer()* verifica também, depois de adicionar um jogador, se o jogo deve começar ou não. Se existirem 2 jogadores ou mais, o jogo começa, sendo todos os jogadores notificados de tal. No decorrer do jogo a memória partilhada também vai sendo atualizada de modo a manter o painel com os dados mais recentes do jogo.

2.1 Requisitos implementados

ID	Funcionalidade / requisito	Estado
1	Comunicação por *named pipes*	implementado
2	Sincronicação (*mutexes*)	implementado
3	Utilização de *threads*	implementado
4	Memória partilhada	implementado
5	Utilização do *Registry*	implementado
6	Interface gráfica WIN32	implementado

Tabela 1: Requisitos implementados

Todas as principais matérias abordadas na unidade curricular e necessárias para o trabalho prático foram implementadas no projeto.

3 Implementação

3.1 Árbitro

O programa árbitro é composto pelos seguintes ficheiros *.c* e respetivos *headers*:

- **main.c** - ficheiro principal do programa árbitro.
- **game.c** - lógica do jogo e manipulação do *registry* (a *shared memory* é manipulada diretamente onde necessário).
- **players.c** - manipulação dos jogadores (humanos e *bots*).
- **threads.c** - comunicação do jogo (threads para aceitar/rejeitar jogadores e processar pedidos destes).
- **utils.c** - funções centrais para o programa *arbitro*.

3.1.1 main.c

O ficheiro *main* do programa árbitro contém o mínimo de código possível, chamando apenas determinadas funções para a inicialização do programa. Conta com um ciclo *while* para processar comandos de administrador.

3.1.2 game.c

Neste ficheiro estão todas as funções que tratam da lógica do jogo e do *registry*. No que toca à lógica do jogo, tem uma função para começar o jogo, verificar se as letras enviadas por um utilizador formam uma palavra do dicionário, entre outras. Quanto ao *registry* apenas tem as funções básicas para criar, escrever e ler. A memória partilhada é maioritariamente manipulada nas funções que gerem as letras e não foi necessário criar funções adicionais.

3.1.3 players.c

Um dos ficheiros mais importantes do programa. Contém todas as funções para lidar com os jogadores, isto é, comunicar e gerir os mesmos. As funções mais importantes e centrais são a *addPlayer()* e a *handlePlayerMessage()*. A função *addPlayer()* trata de guardar todos os dados necessários do jogador e criar uma *thread* para tratar da comunicação com o mesmo. A função *handlePlayerMessage()* serve apenas para processar pedidos por parte do jogador. Outra função também a destacar é a função *broadcastMessage()*, que envia uma mensagem para todos os jogadores.

3.1.4 threads.c

Aqui encontram-se as funções centrais da comunicação. Contém apenas duas funções para criar as *threads* que vão processar a comunicação entre o árbitro e os jogadores. A função *connectionListener()* serve para ficar à escuta de novos jogadores e chamar a função *addPlayer()* quando há uma nova conexão. A função *playerHandler()* serve para a *thread* individual de cada conexão. Vai servir para processar as mensagens do jogador ao qual está associada.

3.1.5 utils.c

Neste ficheiro temos as funções essenciais do programa árbitro, como a função *handleCommand()* para processar os comandos do árbitro, *listPlayers()* para mostrar os jogadores atuais, *initializeManager()* para lançar as *threads*, criar os eventos, etc. Uma função também a destacar é *sendMessageToPlayer()*. Sempre que é preciso enviar uma mensagem para um jogador, evita-se escrever uns bons pedaços de código, sendo até possível evitar criar variáveis do tipo da estrutura das mensagens e preencher a estrutura diretamente na chamada da função.

3.2 JogoUi

Este programa é para os jogadores. Contém uma estrutura muito simples, contendo apenas um *main* e um *utils*. Nada mais vai fazer do que enviar e receber mensagens do árbitro, tendo apenas uma thread para processar essas mensagens em paralelo. Para facilitar o processamento de mensagens, é utilizado um *switch case* com os tipos para cada mensagem (*3.5.1 - Tipos de mensagem*).

3.3 *Bot*

O programa *bot* nada mais é do que uma cópia do programa jogoUi com algumas alterações para enviar letras aleatórias. É executado a partir de um comando do árbitro, sendo toda a lógica semelhante ao jogoUi. Apesar disso, cumpre o seu dever de criar uma competição artificial e até é possível escolher um nível de dificuldade. Quando o *bot* é iniciado recebe uma mensagem do tipo *DICTIONARY* com todas as palavras no dicionário do jogo. Quanto maior for o nível de dificuldade, mais probabilidade tem de mandar uma palavra correta em vez de um *array* aleatório com as letras disponíveis. O código do *bot* é mais desorganizado, pois foi feito mais à pressa na meta final do trabalho prático, contendo até, apenas, um único ficheiro.

3.4 Painel

O programa painel nada mais faz do que apresentar dados do jogo com suporte de um ambiente gráfico. Apresenta as letras disponiveis, os jogadores e as suas informações.

3.5 data.h

O ficheiro data.h é comum a todos os programas (árbitro, *bot*, jogoUi e painel). Este ficheiro serve para definir as estruturas comuns a todos os programas, tais como:

- **Player** - Esta estrutura contém todos os aspetos de um jogador, como a sua pontuação, se é humano ou um *bot*, o seu nome de utilizador e as respetivas *handles* para os mecanismos utilizados pelos programas.
- **Message** - Esta estrutura contém as informações enviadas numa mensagem, sendo estas o nome de utilizador (raramente utilizado), o seu conteúdo e o seu tipo. O campo "tipo"é explicado posteriormente (*3.5.1 - Tipos de mensagem*).
- **SharedData** - Esta estrutura contém tudo o que vai ser guardado em memória partilhada.

Neste ficheiro estão também definidos os eventos que podem ocorrer durante um jogo (jogo começou, árbitro encerrou, ...), a localização da memória partilhada, o número máximo de jogadores, o *buffer* do conteúdo da mensagem, os tipos de mensagem, entre outros.

3.5.1 Tipos de mensagem

Este mecanismo é utilizado para facilitar o processamento de mensagens entre o árbitro e o jogador, como se fosse utilizado um envelope. Todos os tipos de mensagem estão definidos no ficheiro data.h. Alguns dos tipos de mensagem:

- **JOIN_MSG** - Mensagem de um jogador a pedir para se conectar ao jogo.
- **KICK_MSG** - Mensagem do árbtiro a avisar que o jogador em questão foi expulso do jogo.
- **ERROR_MSG** - Mensagem de erro enviada pelo árbitro. Ex.: *Username* já existe ou o jogo está cheio.
- **LIST_REQ** - Pedido (de um jogador) da lista de jogadores ativos.

  ...

A utilização de tipos de mensagem facilita o processamento de mensagens de ambos os programas, pois evita a necessidade de analisar os conteúdos das mensagens de modo a determinar o que são e para que são.

4 Conclusão

A realização deste projeto permitiu aplicar de forma prática os principais conceitos abordados na unidade curricular. A implementação dos diversos programas requeridos para o projeto foi essencial para aprofundar e consolidar o conhecimento e a matéria abordada ao longo do semestre.

Embora o resultado final não tenha ficado muito distante do inicialmente esperado, reconhecemos que existem aspetos que poderiam ser melhorados. A estrutura do código poderia ser mais coesa e organizada, de forma a facilitar a sua leitura e manutenção. Além disso, algumas das soluções encontradas para resolver os desafios que surgiram ao longo do desenvolvimento poderiam ter sido mais elegantes e eficientes, refletindo uma abordagem mais limpa e robusta. A escassez de tempo nesta parte final do semestre dificultou a melhoria e a organização mais cuidada do código, impedindo a adoção de soluções mais limpas e estruturadas.

Apesar disso, estamos satisfeitos com o resultado alcançado, pois todas as funcionalidades principais foram implementadas com sucesso e o sistema funciona de forma relativamente estável e dentro do previsto.