
# Sistemas Operativos 1

## Programação em C para UNIX

2020 - 2021

**TheForgotten**
**JOSEALM3IDA**

Licenciatura de Engenharia Informática
24 de janeiro de 2021

# Conteúdo

**1 Introdução** **2**

**2 Arquitetura Geral** **2**

**3 Estruturas de Dados** **3**

3.1 Struct Player . . . . . 3

3.2 Struct Wait . . . . . 4

**4 Árbitro** **4**

4.1 Tratamento de Clientes . . . . . 5

4.1.1 Entrada . . . . . 5

4.1.2 Saída . . . . . 6

4.1.3 Comunicação . . . . . 6

4.2 Campeonato . . . . . 6

4.2.1 Fase de Espera . . . . . 6

4.2.2 Fase de Jogo . . . . . 7

4.2.3 Fase Final . . . . . 7

4.3 Comandos ADMIN . . . . . 8

**5 Cliente** **8**

**6 Jogos** **9**

6.1 Unscrambler . . . . . 9

6.2 Obtenção dos Jogos Disponíveis . . . . . 9

**7 Makefile** **11**

**8 Conclusão** **11**

**9 Anexos** **12**

# 1 Introdução

O trabalho prático consiste na implementação de um sistema de gestão de campeonatos de jogos denominado CHAMPION. O sistema pretendido encarrega-se de fazer a ponte entre os jogadores e os jogos, mediando as mensagens trocadas entre ambos e gerindo o campeonato.

O trabalho prático foi concretizado em linguagem C, para plataforma UNIX (Linux), usando os mecanismos do sistema operativo abordados. No que respeita à manipulação de recursos do sistema foi dada prioridade ao uso de chamadas ao sistema operativo face ao uso de funções biblioteca.

# 2 Arquitetura Geral

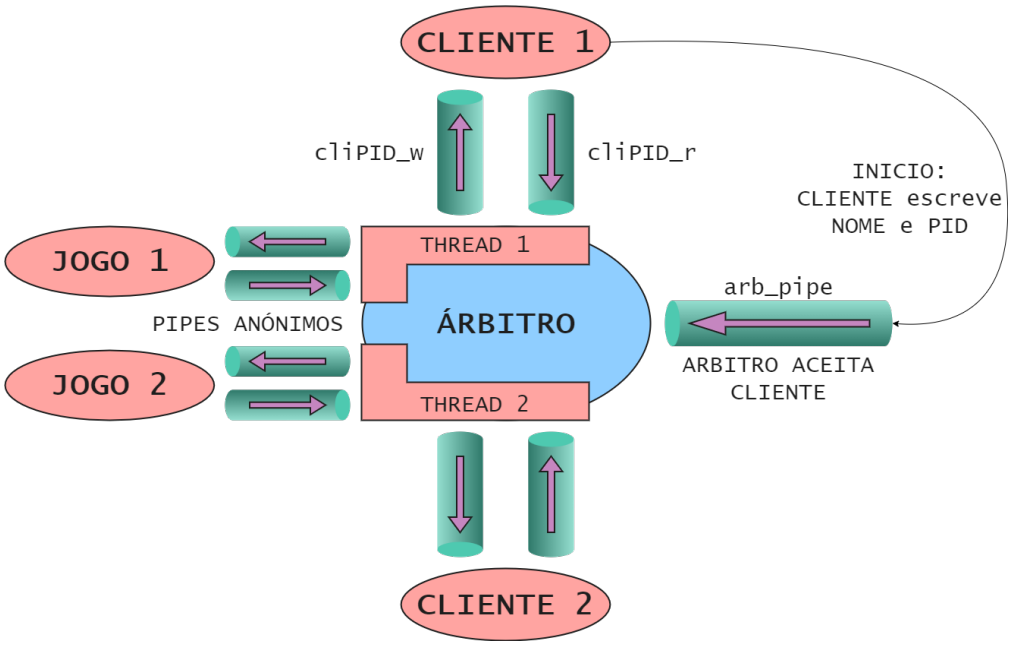


Figura 1: Esquema da arquitetura geral implementada

Sendo o árbitro responsável por equacionar o campeonato este aparece no centro do esquema. Na arquitetura implementada existem 2 pipes por cliente, 2 pipes por jogo e um pipe do árbitro.

Ao iniciar, o árbitro cria o seu pipe. Caso um cliente queira participar do campeonato, a informação sobre o seu jogador é inicialmente enviada por este pipe.

Em seguida, é criada uma thread para este jogador. Esta vai ficar encarregue de verificar se o jogador é válido (ou seja, se o nome está disponível), avisá-lo caso não seja, confirmar a sua entrada e gerir a restante comunicação.

A partir do momento que existam pelo menos dois jogadores, o árbitro começa uma contagem decrescente para o início do campeonato. No caso de sairem jogadores de modo a ficarem menos de dois, a contagem é interrompida e o árbitro volta ao estado de espera por jogadores.

Tendo terminado o tempo de espera, o campeonato começa. Neste momento é iniciado o cronómetro do campeonato e é atribuído aleatoriamente, pelas respetivas threads, um jogo a cada

jogador. Para além disso é também montado o mecanismo de comunicação entre árbitro e jogo usando pipes anónimos. Durante toda a duração do campeonato podem se ligar novos jogadores.

No fim do campeonato é enviado aos jogadores a sua pontuação e o nome e pontuação do melhor jogador daquele campeonato. Neste momento os clientes são questionados se pretendem juntar-se a um novo campeonato e, no caso de esta ser a sua vontade, este tem que passar pelo processo de fornecer dados e se conectar ao árbitro do zero.

Este modelo, usando threads, não constava do trabalho na meta 2, mas consideramos que tinha mais vantagens perante o que estava previamente implementado. As grandes vantagens são o facto de o código ser mais modular e, no caso de dar problemas com um cliente, apenas este é afetado e não todo o campeonato.

# 3 Estruturas de Dados

## 3.1 Struct Player

```
struct player {
   pthread_t th;
   char nome[STR_SIZE];  // Nome do jogador
   char jogo[STR_SIZE];  // Nome do jogo
   pid_t PID;            // PID do processo do cliente
   int score;            // Pontuacao
   int fd[2];            // Pipes de leitura (fd[0]) e escrita (fd[1]), da
       perspetiva do servidor
   bool hasComms;        // Se a comunicacao entre jogador-jogo esta ativa
   bool sair;            // Se a thread do jogador deve sair (true para sair, false
       para continuar)
   int stateCampeonato;  // 0 -> à espera de clientes; 1 -> campeonato começou,
       cliente sem jogo atribuido; 2 -> jogo a executar; 3 -> fim do campeonato

   int jogoIO[2];        // Pipes de leitura (jogoIO[0]) e escrita (jogoIO[1]), da
       perspetiva do servidor
   pid_t jogoPID;        // PID do processo do jogo

   char info[BUFF_SIZE]; // Mensagem com informação sobre o que aconteceu, a ser
       enviada ao cliente

   pthread_mutex_t* lock; // Lock de exclusão mútua
   jogador* prox;        // Ligacao a lista ligada
};
```

Código 1: Struct player

Esta é a estrutura associada aos jogadores. É usada como argumento pelas threads e é guardada numa lista ligada.

O lock é partilhado por todos os jogadores.

### 3.2 Struct Wait

```
struct wait {
   pthread_t threadCron; // ID da própria thread
   int duracao;          // Duração do cronómetro
   bool isFinished;      // Se já terminou a contagem ou não
   pthread_mutex_t* lock; // Lock de exclusão mútua
};
```

Código 2: Struct wait

Esta é a estrutura associada ao cronómetro. Sempre que uma thread cronómetro é aberta é passado como argumento uma estrutura deste tipo. Por causa da thread associada funcionar com recurso a sinais esta é sempre guardada num ponteiro global.

## 4 Árbitro

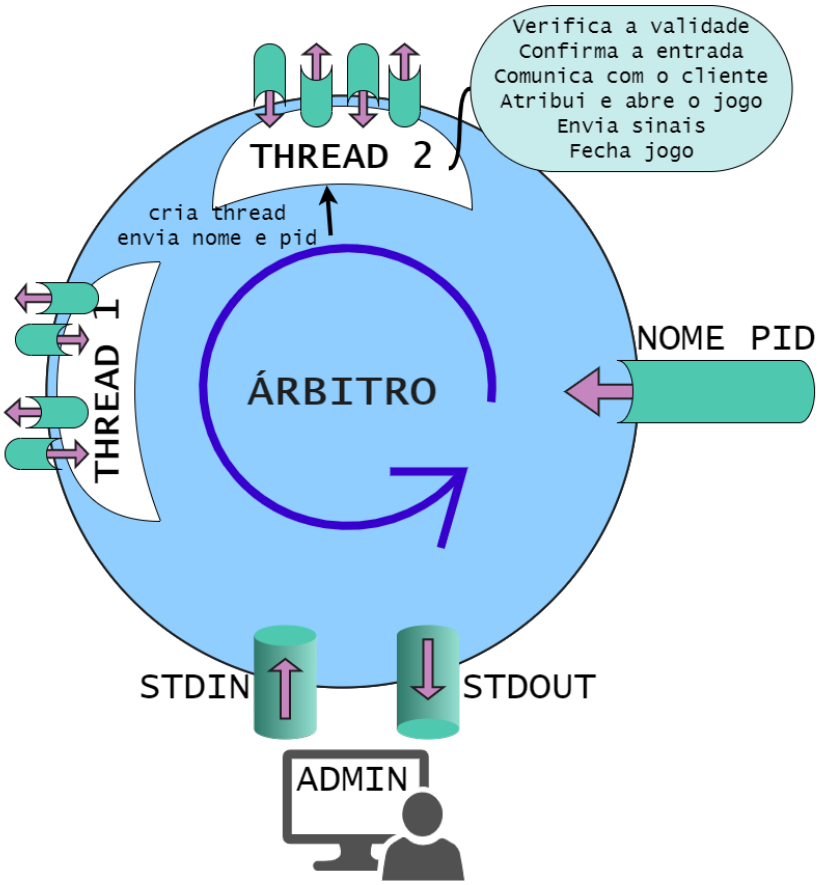


Figura 2: Esquema do árbitro

O árbitro é o cérebro do campeonato do sistema CHAMPION. Este tem sobre os seus ombros a responsabilidade de garantir toda a comunicação.

## 4.1 Tratamento de Clientes

### 4.1.1 Entrada

Um cliente que pretenda participar de um campeonato começa por enviar o nome do seu jogador e o seu PID (id de processo). Esta informação é enviada ao árbitro pelo pipe deste (arb_pipe).

De modo a receber a informação o árbitro encontra-se num while loop onde está constantemente a verificar se recebeu informação ou do STDIN ou do seu pipe. Assim, quando recebe algo no seu pipe executa o seguinte excerto de código:

```
if ((n = read(fd_arb, buff, sizeof(buff) - sizeof(char))) == -1)
   sair(EXIT_ERR_MSG, "ERRO: Falha a ler do meu pipe!\n");

buff[n] = '\0';

nome = strtok(buff, " ");
temp = strtok(NULL, " ");

if (isAbsoluteNumber(temp)) {
   pid = atoi(temp);

   novoJog = malloc(sizeof(jogador));
   if (novoJog == NULL)
      sair(EXIT_ERR_MSG, "ERRO: Impossivel de alocar memoria para a nova thread!\n");

   strcpy(novoJog->nome, nome);
   novoJog->PID = pid;
   novoJog->lock = &lock;
   novoJog->stateCampeonato = currState;

   pthread_create(&(novoJog->th), NULL, &threadJogador, novoJog);
}

continue;
```

Código 3: Entrada de um cliente

Neste excerto de código o árbitro lê a string do pipe, extrai o nome do jogador e o PID do cliente que se pretende juntar e se este PID for válido então aloca memória para uma nova estrutura jogador, preenche-a e cria uma nova thread, enviando esta como argumento.

Esta nova thread terá a responsabilidade de verificar se este jogador é válido (ou seja, se o seu nome é único) e, caso não seja, avisa o cliente do sucedido. Em caso de ser válido, confirma a comunicação e junta-o à lista ligada de jogadores que pretendem / estão a participar do campeonato.

A thread faz também uso da função pthread_sigmask de modo a não apanhar sinais SIGINT direcionados ao árbitro.

#### 4.1.2 Saída

O cliente pode ser removido do campeonato de diversas maneiras: pode desistir, o árbitro pode retirá-lo, o árbitro pode fechar, o campeonato pode acabar ou pode ocorrer um erro.

No caso do cliente desistir, a thread do árbitro responsável por este cliente é avisada da situação através dos pipes e procede à remoção do cliente da lista ligada dos participantes e, consequentemente, do campeonato.

No caso do árbitro retirar o jogador (usando o comando kplayer) a função principal avisa à thread que tem de sair através da alteração da flag sair presente na estrutura jogador.

No caso do árbitro fechar a situação é semelhante à anterior, mudando apenas a informação enviada ao cliente.

No caso do campeonato acabar a situação é semelhante às anteriores, mudando apenas a informação enviada ao cliente. Neste caso é também enviado um sinal SIGUSR1.

No caso de ocorrer um erro a thread é responsável por limpar toda a informação deste cliente e avisá-lo da situação ocorrida.

#### 4.1.3 Comunicação

Todas as mensagens enviadas pelo cliente são tratadas pelo árbitro.

Se a mensagem enviada pelo cliente começar com o caracter # esta é interpretada pelo árbitro pois é predefinido que todas as mensagens começadas por este caracter são comandos e não interações com o jogo. Caso contrário a mensagem é simplesmente reencaminhada para o jogo.

No caso das comunicações do cliente com o jogo terem sido suspendidas (usando o comando splayer), o árbitro não reencaminhará as mensagens deste para o jogo e, ao invés disso, irá avisá-lo que este tem as comunicações suspendidas, podendo este apenas enviar comandos. Se o árbitro desejar retomar as comunicações do jogador (usando o comando rplayer) o cliente é informado de que as comunicações já estão normais e as informações voltam a ser reencaminhadas.

### 4.2 Campeonato

#### 4.2.1 Fase de Espera

Esta é a fase pré-campeonato. Nesta fase o árbitro espera pela chegada de jogadores.

Quando existirem pelo menos dois jogadores, o árbitro inicia uma contagem regressiva para o começo do campeonato. Se, em algum momento, o número de jogadores se tornar inferior a dois, a contagem termina não iniciando o campeonato.

Na nossa implementação de cronómetros no árbitro o tempo é contado através duma thread com um simples sleep() cujo código é o seguinte:

```
void* threadEsperar(void *arg) {
   sigset_t set;

   sigemptyset(&set);
   sigaddset(&set, SIGINT); // PROIBIR A THREAD DE APANHAR SIGINTS
```

```
    if (pthread_sigmask(SIG_BLOCK, &set, NULL) != 0) {
       fprintf(stderr, "ERRO: Não foi possível colocar o SIGMASK numa thread.\n");

       pthread_mutex_lock(argEspera->lock);
       argEspera->isFinished = true;
       argEspera->threadCron = 0;
       pthread_mutex_unlock(argEspera->lock);

       pthread_exit(NULL);
    }

    sleep(argEspera->duracao);

    pthread_mutex_lock(argEspera->lock);
    argEspera->isFinished = true;
    argEspera->threadCron = 0;
    pthread_mutex_unlock(argEspera->lock);
}
```

Código 4: Thread cronómetro

Assim, quando existirem pelo menos dois jogadores e a contagem tiver terminado, as threads são avisadas, atribuem um jogo ao cliente e o campeonato começa.

### 4.2.2 Fase de Jogo

Nesta fase, a comunicação jogador-jogo torna-se possível (o jogador pode finalmente jogar). A comunicação jogador-jogo é realizada indiretamente, ou seja, o jogador comunica com o árbitro pelo seu respetivo pipe e este é responsável por reenchaminhar esta informação para o pipe anónimo do jogo. Depois, recebe informação do jogo por outro pipe anónimo e reencaminha-a para o cliente, ou seja, não faz qualque tratamento ou formatação desta informação.

A única altura em que o árbitro interpreta a informação recebida é quando esta começa por um # como referido anteriormente.

Este processo é repetido até ao final do campeonato.

### 4.2.3 Fase Final

Esta é a fase em que o campeonato termina. Aqui as threads acabam por sair devido ao facto da variável currState, que faz parte do jogador, ter o seu valor alterado para 3 (ou seja, fim de campeonato). Assim, a função principal do árbitro fica apenas responsável por recolher o nome e o score do jogador com melhor pontuação e fornecer esta informação aos restantes (colocando uma string com o formato "nome score"na variável info de todas as estruturas jogador).

Sendo este o final do campeonato é enviado um sinal SIGUSR1 pelas threads aos seus respetivos clientes tal como foi mencionado anteriormente.

Por fim, o árbitro retorna à fase de espera a não ser que seja explicitamente fechado.

### 4.3 Comandos ADMIN

O árbitro e, deste modo, o campeonato, são controlados pelo ADMIN. Este envia, a partir da linha de comandos, ordens que devem ser executadas pelo árbitro. Estas podem ser as seguintes:

- **players:** mostra na consola os jogadores que estão no momento registados e se o campeonato já estiver a decorrer mostra o jogo que lhes está atribuído;
- **games:** mostra na consola os jogos disponíveis para serem atribuídos;
- **kplayer:** remove o jogador player do campeonato e informa o cliente;
- **splayer:** suspende a comunicação jogador-jogo do jogador player (este sempre que tentar enviar algo ao jogo é avisado de que tem a comunicação suspendida);
- **rplayer:** retoma a comunicação jogador-jogo do jogador player;
- **end:** termina o campeonato imediatamente, forçando o árbitro a entrar na fase final;
- **exit:** encerra o árbitro e consequentemente o campeonato.

## 5 Cliente

O cliente é o responsável por permitir a um jogador participar num campeonato do sistema CHAMPION.

Numa fase inicial este começa por verificar se o árbitro existe verificando se consegue ligar-se ao pipe do árbitro. Se efetivamente o árbitro não existir este fecha. Se existir, abre dois pipes: um para receber informação do árbitro e outro para enviá-la, cujos nomes seguem os formatos cliPID_r (para escrita) e cliPID_w (para leitura), onde PID será substituído pelo seu respetivo ID de processo.

Se o árbitro estiver então aberto e a funcionar, o cliente pede ao jogador o seu nome. Nesta parte está num while loop à espera de receber confirmação do árbitro se o jogador pode participar ou não.

Em seguida, este entra noutro while loop para poder receber informação tanto do pipe do árbitro como do STDIN (ou seja, input do jogador) através dum select. Toda a informação recebida do STDIN é imediatamente redirecionada para o seu pipe de escrita para ser então tratado pelo árbitro.

Aquando a receção dum sinal SIGUSR1, que por defeito significa que o campeonato em que este participava terminou, este questiona o jogador se pretende participar num novo campeonato. No caso do jogador desejar participar de novo o cliente então passa por todo o processo de recolha de dados e confirmação dos mesmos com o árbitro de novo, tal como a abertura dos pipes.

O cliente pode enviar comandos para o árbitro. Para isto, este tem apenas que colocar um # no início da informação a enviar. O cliente não efetua qualquer tratamento destes comandos, sendo a informação recebida de total responsabilidade do árbitro.

# 6 Jogos

## 6.1 Unscrambler

Este é o jogo que propomos. O objetivo dele é muito simples: o jogador recebe uma palavra com as letras trocadas e tem que descobrir qual é a palavra.

Por cada palavra acertada o jogador acumula um ponto, não havendo penalizações para quantas vezes erra.

Apesar de ter uma premissa simples, este jogo é bastante complicado.

## 6.2 Obtenção dos Jogos Disponíveis

A estratégia usada para obter os jogos disponíves consiste em duas partes. Na primeira parte é verificado se o diretório fornecido existe usando o seguinte código:

```
bool existeDir(char* dir) {
   struct dirent *de;

   // opendir RETORNA UM PONTEIRO DO TIPO DIR
   DIR *dr = opendir(dir);

   if (dr == NULL) // opendir DEVOLVE NULL SE NÃO FOI POSSÍVEL ABRIR O DIRETÓRIO
   return false; // DIRETÓRIO NÃO EXISTE

   return true; // DIRETÓRIO EXISTE
}
```

Código 5: Verificar se existe diretório

Usando conteúdos não lecionados, esta função começa por declarar um ponteiro para uma estrutura dirent. Em seguida, tenta abrir o diretório e se falhar na abertura do mesmo significa que este não existe e, por isso, devolve false. Caso contrário, devolve true.

Em seguida passa à segunda parte, onde são lidos os ficheiros presentes nesse diretório e se verifica se o nome deles começa por "g_", pois apenas estes devem fazer parte dos jogos disponíveis. Para além disso, como de modo geral em UNIX apenas os ficheiros executáveis não possuem extensão, verificamos também se existe pelo menos um "."no nome do ficheiro. Se existir não o consideramos como executável. Isto é tudo realizado a partir do seguinte código:

```
while ((de = readdir(dr)) != NULL)
   if(de->d_type == DT_REG && de->d_name[0] == 'g' && de->d_name[1] == '_') {
      for(i = strlen(de->d_name) - 1; i >= 0; i--)
         if (de->d_name[i] == '.')
            break;

      if (i != -1)
         continue;

      temp = realloc(arr, sizeof(char*) * ((*tam) + 1));
```

```
        if (temp == NULL) {
            for(i = 0; i < *tam; i++)
                free(arr[i]);
            free(arr);
            return NULL;
        }

        arr = temp;

        arr[*tam] = malloc(sizeof(de->d_name));
        strcpy(arr[*tam], de->d_name);

        (*tam)++;
    }
```

Código 6: Obter jogos válidos

Este while loop é executado enquanto existirem ficheiros no diretório. É verificado que tipo de ficheiro está a ser lido (DT_REG de acordo com a documentação é um ficheiro "banal"ou regular). Em seguida realoca o array de strings fazendo todas as verificações necessárias para ter a certeza de que tudo correu bem.

Este código é executado na função char** lerDiretorio(char* dir, int* tam). Os executáveis válidos detetados têm o seu nome guardado num array de strings que é posteriormente devolvido ao árbitro.

# 7 Makefile

Não tendo sofrido alterações desde a meta 2, o nosso makefile apresenta as seguintes rules:

| **Rule** | **Dependencies** | **O que faz** |
|---|---|---|
| all | arbitro cliente jogos | Compila todos os programas associados ao trabalho |
| debug | arbitro-debug cliente-debug | Compila todos os programas (menos os jogos) associados ao trabalho para debugging |
| cliente | cliente.o utils.o | Compila o programa cliente |
| cliente.o | cliente.c utils.h | Compila o objeto cliente |
| cliente-debug | cliente.c utils-debug | Compila o programa cliente para debugging |
| arbitro | arbitro.o utils.o | Compila o programa arbitro |
| arbitro.o | arbitro.c arbitro.h utils.h | Compila o objeto arbitro |
| arbitro-debug | arbitro.c arbitro.h utils-debug | Compila o programa arbitro para debugging |
| utils.o | utils.c utils.h | Compila o objeto utils |
| utils-debug | utils.c utils.h | Compila o objeto utils para debugging |
| jogos | $(GAMES) | Responsável por pedir para compilar todos os jogos do subdiretório Games |
| g_% | g_%.o | Compila executáveis que comecem por "g_". Usado para compilar os executáveis dos jogos |
| g_%.o | g_%.c | Compila objetos que comecem por "g_". Usado para compilar os objetos dos jogos |
| clean | clean-obj clean-exe | Limpa todos os ficheiros gerados pelo makefile |
| clean-obj | N/A | Limpa todos os ficheiros .o gerados pelo makefile |
| clean-exe | N/A | Limpa todos os executáveis gerados pelo makefile |
| clean-pipes | N/A | Limpa todos os pipes que possam ter sido deixados por um erro de execução |

Apesar de não convencional, a rule clean-pipes foi adicionada para facilitar o processo de remoção dos pipes. Esta apenas foi útil durante o debugging dos programas propostos.

# 8 Conclusão

Ao longo deste trabalho, deparámo-nos e fomos resolvendo vários desafios que não esperávamos ter, tratando-se de uma excelente oportunidade para consolidação de matéria das aulas teóricas e práticas de Sistemas Operativos. Este permitiu-nos colocar em prática conceitos importantes sobre a arquitetura, criação e desenvolvimento de programas para sistemas UNIX, dando-nos uma visão para os vários detalhes dessas atividades.

# 9 Anexos

## Lista de Figuras


1 Esquema da arquitetura geral implementada . . . . . 2
2 Esquema do árbitro . . . . . 4


## Pedaços de Código


1 Struct player . . . . . 3
2 Struct wait . . . . . 4
3 Entrada de um cliente . . . . . 5
4 Thread cronómetro . . . . . 6
5 Verificar se existe diretório . . . . . 9
6 Obter jogos válidos . . . . . 9