Instituto Superior de Engenharia de Coimbra

Unidade Curricular de Sistemas Operativos

Turma P3 - Ano Lectivo 2017/2018

Trabalho Prático - Meta 2

Bomberman

António Daniel Neves Faneca nº21250021

Ricardo Jorge Pinho Marques nº21250193

Índice

1.Introdução

Para a Unidade Curricular de **Sistemas Operativos** foi proposto aos alunos a realização do jogo **Bomberman** em modo multijogador com uma interface em modo-texto em que todos os jogadores estão ligados à mesma máquina UNIX.

Este projeto tem como guia um enunciado onde estão explícitos todos os requisitos necessários para a obtenção da nota máxima, assim como um breve sumário das regras do jogo.

Este relatório consiste somente na abordagem do grupo à segunda meta, onde agora nos é proposto a implementação de uma arquitectura e estratégia de ***named pipes***. As alterações do programa em relação à primeira meta também vão estar em foco, como por exemplo as nossas estruturas de dados.

A validação e os testes realizados para esta meta já incluem a presença de um servidor e de um ou mais clientes.

2. Estruturas de Dados

As estruturas de dados deste projecto estão definidas em dois ficheiros (*servidor.h* e *cliente.h*).

2.1. Labirinto

Para esta meta ainda não é requerido a estrutura de dados Labirinto, mas encontra-se num estado primitivo no ficheiro *servidor.h* um rascunho desse mecanismo.

2.2. Jogador

No decorrer da composição deste trabalho, o grupo chegou a um consenso de colocar quatro membros nesta estrutura de dados:

- **username** - Array de caracteres onde vai ser guardado o username do jogador;
- **password** - Array de caracteres onde vai ser guardada a password do jogador;
- **online** - Variável inteira, que varia entre 0 e 1, que define se o jogador está ou não online.
- **pontuação** - Variável inteira que vai guardar a pontuação do jogador;
- **nomeDoFicheiro -** Array de caracteres onde vai ser guardado o nome do ficheiro temporário ao qual o jogador pertence;

2.3. Mensagem

Todos os clientes e o servidor irão necessitar de uma estrutura mensagem.
Irá ser necessária na estrutura mensagem uma representação de quais os elementos de comunicação entre cliente e servidor.

- **op1, op2, op3, op4 -** Array de caracteres que têm por base guardar palavras-chave essenciais para o funcionamento de certas funções;
- **resposta -** Array de caracteres que vai conter a resposta dada pelo recipiente ao remetente, após todo o processo do envio de uma mensagem;
- **endereço -** Array de caracteres que vai conter o endereço - nome do ficheiro ao remetente a que a mensagem está associada (ex: CPCliente <PID>).

2.4. dados_pipes

O funcionamento das threads é definido por uma estrutura de dados que tem informação sobre:

- **fd** - Inteiro onde vai ser guardado o descritor do named pipe que vai ser lido;
- **qual** - Array de caracteres com o nome do named pipe para efeitos informativos;

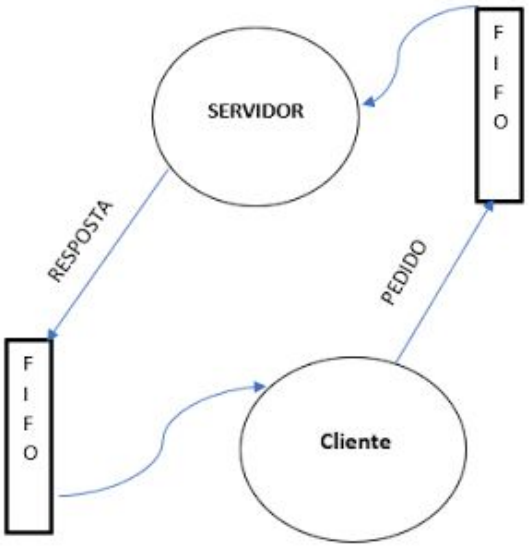
3. Arquitetura e estratégia de named pipes

A estratégia abordada pelo nosso grupo para a realização deste ponto fulcral passa por criar os named pipes que operam através das suas funções habituais (open, close, read, write, unlink, etc.).

Devido à semântica de bloqueio dos FIFO's e à ordem pela qual as funções têm de ser efetuadas, o trabalho está desenvolvido de forma a que não haja qualquer tipo de bloqueio e tudo se desenvolva naturalmente na relação cliente/servidor.

A nossa arquitectura passa por um simples pedido/resposta em que sempre que há um pedido por parte do cliente ele tem que se dirigir ao ficheiro FIFO do servidor e sempre que há uma resposta por parte do servidor, é sempre na direção do ficheiro FIFO que se faz o primeiro contacto ao cliente.

Em suma, o servidor tem de se encontrar sempre disponível para a recepção de pedidos dos vários clientes, para isso a sua identificação tem de ser conhecida à partida pelos clientes para o efetuamento de pedidos.



4. Funcionalidades realizadas

Neste ponto vão ser discutidas as novas implementações no trabalho, tendo em conta os requisitos da meta anterior e da atual meta 2.

As funcionalidades anteriores podem ser consultadas no relatório da meta 1.

4.1 Criação de um Servidor

Para que exista uma saudável interação entre processos (clientes e servidores), criamos um servidor que tem como base o processamento de comandos e o processamento de pedidos de clientes.

Previamente à criação de um servidor verificamos se existe já um servidor a correr. Se não, é criado um ficheiro FIFO que vai ser a essência da comunicação entre servidor/cliente.

A função open() irá abrir as portas para a comunicação, iniciado assim o servidor.

```
/* VERIFICAR SE EXISTE "CP" DO SERVIDOR -- APENAS UM!!!*/
if(access("CPservidor", F_OK)==0){
    printf("[SERVIDOR] Ja existe um servidor!\n");
    exit(1);
}

if(mkfifo("CPservidor", 0600) < 0){ // CRIAR "CP" DO SERVIDOR
    perror("[SERVIDOR] Erro na criação do FIFO...");
    exit(1);
}

fd_servidor = open("CPservidor", O_RDWR);  /* ABRIR "CP" DO SERVIDOR - MINHA (open - O_RDONLY) */
printf("[SERVIDOR] Servidor Iniciado!\n");
```

4.2 Desligar um servidor

A inserção do comando “shutdown” provoca o fecho do servidor através da função *close* e a remoção do ficheiro FIFO pelo função *unlink.*
Como é pedido no enunciado o sinal SIGUSR1 tem de ser indicado para encerrar o servidor.
A referência ao sinal SIGUSR1 irá ser referida mais tarde no relatório.

```
else if (strcmp(cmd[0], "shutdown") == 0) {
    if (tamCMD == 1){
        free(mv);
        printf("[SERVIDOR] SERVIDOR DESLIGADO\n");
        close(fd_servidor);     // FECHAR "CP" DO SERVIDOR
        unlink("CPservidor");   // REMOVER "CP" DO SERVIDOR


        raise(SIGUSR1);

    }
```

4.3 Criação de um cliente

Colocamos a verificação da existência de um servidor que possa hospedar o cliente. Em caso positivo é passado para a estrutura mensagem o endereço a que este cliente está associado a partir da função **getpid** que obtém um número de identificação designado para esse processo.
Posteriormente abrimos o servidor e passamos-lhe a mensagem através de um named pipe de que um novo cliente está pronto para se conectar.

```
sprintf(msg.endereco, "CPcliente %d", getpid());    // CRIAR "CP" DO CLIENTE
mkfifo(msg.endereco, 0600);                         //0600 READ && WRITE
fd_servidor = open("CPservidor", O_WRONLY);         // ABRIR "CP" DO SERVIDOR (open - O_WRONLY)

strcpy(msg.op1, "novo");
sprintf(msg.op2, "O Cliente %s acabou de se connectar", msg.endereco);

write(fd_servidor, &msg, sizeof(msg));      // ENVIAR PEDIDO PARA "CP" DO SERVIDOR (write)
fd_cliente = open(msg.endereco, O_RDONLY);  // ABRIR "CP" DO CLIENTE (open - O_RDONLY)
read(fd_cliente, &msg, sizeof(msg));        // RECEBER RESPOSTA NA "CP" DO CLIENTE (read)
close(fd_cliente);                          // FECHAR "CP" DO CLIENTE - MINHA (close)
```

A função *processaPedidos()* vai desbloquear o named pipe de imediato com uma função *read()* que vai procurar qual o pedido feito pelo cliente.

4.4 Kick de um cliente

Um comando ao qual temos de dar especial atenção é ao comando “Kick” que quando inserido no terminal do servidor tem o especial propósito de encerrar o jogo para um determinado cliente.

```
void sinalizaKick(){
    pid_t pid;

    printf("\nClientes Ativos: \n");
    int conta = contaPlayers();
    int i = 0;


    if (conta == 0) {
        printf("Nao existem Users a jogar\n");
    }

    for (i = 0; i < conta; i++) {
        if(strcmp(msg.op1, v[i].username)==0){
            v[i].online = 0;
            pid = getPid(v[i].nomeDoFicheiro);
            kill(pid, SIGUSR1);
        }
    }
    grava();
}
```

Nesta função vai existir um ciclo que vai percorrer o nosso vetor dinâmico com a informação de todos os jogadores presentes no ficheiro de texto, que procura o username que foi introduzido no servidor.

4.5 Desligar um cliente

Um jogador conectado ao escolher a opção de sair manda uma mensagem ao servidor da sua intenção. O processamento de pedidos do servidor encarrega-se de atualizar o vetor dinâmico com a informação dos jogadores de que este jogador já não mais se encontra a jogar.

```
}else if (strcmp(cmd[0], "sair") ==0) {
    strcpy(msg.op1, cmd[0]);

    write(fd_servidor, &msg, sizeof(msg));
    fd_cliente = open(msg.endereco, O_RDONLY);
    read(fd_cliente, &msg, sizeof(msg));
    close(fd_cliente);

    raise(SIGUSR2);
}
```

É ativado o sinal SIGUSR1 que ao levar o cliente para uma função de tratamento de sinais termina a sua sessão com a função exit(0), mas não antes de ser usada a função *unlink()* para remover o ficheiro FIFO através do seu endereço.

```
if(sinal == SIGUSR2){
    printf("Sessao Terminada. %d \n", sinal);
    unlink(msg.endereco); // REMOVER "CP" DO CLIENTE
    exit(0);
}
```

4.6 Login

Para um cliente se tornar num jogador ele terá que passar pela função *login()* que vai estar em ciclo infinito até o cliente acertar no username e na password de um jogador já existente no sistema, isto é no ficheiro de texto.

Sempre que existe uma tentativa de passar uma mensagem nova por um named pipe é enviado um pedido ao servidor para ele validar o login. Passando por verificar se o username e a password existem no sistema e depois verificar se já não existe ninguém com essas mesmas credenciais conectado. Caso se confirme, é chamada uma função *grava()* que faz essa alteração ao ficheiro de texto de imediato.

```
int validaLogin(char user[], char pass[], char end[]){
    int total = contaPlayers();

    for(int i = 0; i < total; i++){
        if(!strcmp(user, v[i].username)){
            if(!strcmp(pass, v[i].password)){
                if(v[i].online == 0){
                    v[i].online = 1;
                    strcpy(v[i].nomeDoFicheiro, end);
                    grava();
                    strcpy(msg.resposta,"Credenciais validas.");
                    return 1;
                }else{
                    strcpy(msg.resposta,"User já se encontra a jogar");
                    return 0;
                }
            }
            else
                strcpy(msg.resposta,"Credenciais invalidas.");
                return 0;
        }
    }
    strcpy(msg.resposta,"Credenciais invalidas.");
    return 0;
}
```

4.7 Threads

O cenário deste projecto é composto por um número limitado de clientes mas só de um servidor, por isso temos em mãos um caso de leitura de várias fontes em simultâneo no lado do servidor para além do seu teclado.

Mesmo antes do agora nosso novo jogador poder começar a inserir o quer que seja este processo leva-o a juntar-se a uma thread que permite que os seus comandos possam ser processados ao mesmo tempo que outros. Este caso exige que o servidor esteja tanto à espera de uma mensagem de um cliente qualquer ou da inserção de um comando no no seu terminal através do teclado.

A função *processaPedidos()* é onde todos os pedidos de todos os clientes são processados.

```
if(pthread_create(& tpipea, NULL, &processaPedidos, tdados) != 0)
        printf("Erro a criar a thread...");
```

4.8 Sinais

Foi desenvolvida para esta meta uma função de tratamento de sinais que recebe o sinal e trata-o com a devida especificidade chamada *trata()*.

```
void trata(int sinal){
    if(sinal == SIGUSR1){
        printf("A terminar o programa. A informar clientes ativos... %d \n", sinal);
        free(v);
        exit(0);
    }


    if(sinal == SIGHUP || sinal == SIGINT){
        sinalizaFim();
        printf("Tentativa de forçar o fecho do terminal. A garantir uma saída segura...");
        printf("[SERVIDOR] SERVIDOR DESLIGADO\n");

        close(fd_servidor);     // FECHAR "CP" DO SERVIDOR
        unlink("CPservidor");
    }
}
```

4.8.1 SIGUSR1 & SIGUSR2

São sinais a ser usados pelo utilizador de forma personalizada. Era mandatório no nosso trabalho implementar o sinal SIGUSR1, para sinalizar o término do servidor. Assumiu-se então que quando o comando *shutdown* fosse inserido que o SIGUSR1 seria sinalizado e o algoritmo trataria de efetuar o devido tratamento do sinal.

4.8.2 SIGHUP & SIGINT

Idealmente, o servidor/cliente apenas seriam terminados recorrendo aos comandos personalizados que implementamos em ambos (“shutdown” e “sair” para o servidor e cliente, respectivamente).
No entanto, sabemos que, por variadas razões e em diferentes contextos, o programa pode vir a ser terminado abruptamente.
Se o terminal for desligado (SIGHUP) ou interrompido (SIGINT) manualmente pelo utilizador, implementamos mecanismos no algoritmo que permite ao sistema efetuar o tratamento de ambos os sinais e terminar o programa em segurança, assegurando a integridade dos dados e estado do programa.

4.9 Funcionalidades de menor destaque

4.9.1 Gravar

A funcionalidade de gravar o ficheiro de texto é algo de algum relevo no trabalho, para isso temos uma pequena função chamada *grava()* que coloca tudo em ordem dentro do ficheiro.

4.9.2 Obter o PID

Esta pequena função tem o objetivo de colocar o pid num número inteiro para uma melhor conexão entre os vários pontos do trabalho.

5. Verificação e validação

Para garantir que o nosso trabalho tinha um certo grau de qualidade foram feitos alguns testes durante a implementação dos vários requisitos que o enunciado sugere.

5.1 Comando *test*

Foi implementado um comando *test* que, como sugere o nome, é um pequeno teste de comunicação, em que o cliente manda uma mensagem ao servidor.

5.2 Teste ao comando *shutdown*

Foi verificado e validado que o nosso comando *shutdown* consegue de facto terminar o servidor assim como todos os jogadores envolvidos.

5.3 Verificação da integridade dos dados

Um teste que se repetia logo após o comando shutdown, era a verificação da integridade do ficheiro de texto, vendo se todos os clientes se encontravam lá e devidamente organizados.

5.4 Teste ao comando *kick*

Foi testado e verificado que o comando kick consegue de facto terminar o jogador em causa sem colocar em causa o servidor.

5.5 Mais que um jogador ligado

Ocasionalmente eram testados comandos com mais de um jogador conectado para ver se existiam repercussões a terceiros. Por exemplo, se um jogador é kickado o outro conectado não pode ter nada a haver com a situação.