IPV – Instituto Politécnico de Viseu

ESTGV – Escola Superior de Tecnologia e Gestão de Viseu

Departamento de Informática

Relatório do Trabalho Prático 3

Licenciatura em Engenharia Informática

Realizado em

Sistemas Operativos

por

Guilherme Pedrinho – 25215

Francisco Marques – 20888

Orientadores

Entidade: Sistemas Operativos

ESTGV: Carlos Quental

Viseu, 2024

IPV – Instituto Politécnico de Viseu

ESTGV – Escola Superior de Tecnologia e Gestão de Viseu

Departamento de Informática

Relatório do Trabalho Prático 3

Licenciatura em Engenharia Informática

Realizado em

Sistemas Operativos

por

Guilherme Pedrinho – 25215

Francisco Marques – 20888

Orientadores

Entidade: Sistemas Operativos

ESTGV: Carlos Quental

Viseu, 2024

Resumo

A comunicação entre processos (IPC) é um mecanismo que permite que processos que executam em paralelo possam trocar dados e coordenar as suas ações. Neste trabalho, exploramos diferentes tipos de IPC para obtermos uma comunicação entre processos funcional

Palavras-chave: comunicação, IPC

Índice

1. Introdução

O terceiro trabalho na cadeira de Sistemas Operativos visa uma maior familiarização com a programação de mecanismos de sincronização e comunicação entre processos utilizados em sistemas operativos, no ambiente Unix/Linux lecionados nas aulas.

O objetivo deste trabalho de grupo é a simulação de uma comunicação servidor/cliente(s), em que os servidores têm acesso a uma base de dados (criada pelo utilizador) composta por livros com um código identificador e, os clientes, pedem, com o código do livro, informações sobre o mesmo.

Para este trabalho foram utilizados dois métodos de IPC (inter-process comunication) sendo estes, *shared memory* e *messages queues*.

2. Metodologia

Neste capítulo vamos analisar a abordagem selecionada para o desenvolvimento deste projeto.

2.1 Análise do problema

Como já foi referido, este trabalho teve como objetivo encontrar uma solução eficiente para a comunicação de vários processos clientes, com um processo servidor.

O servidor continha uma base de dados (ficheiro *txt*) com livros armazenados. Estes livros são compostos por um código único identificador, nome do autor, nome do livro e o preço do livro. O servidor, ao ser iniciado, cria uma fila de mensagens e uma memória partilhada para possibilitar a comunicação com os clientes.

O cliente, ao ser iniciado, é inserido na lista de clientes ativos no servidor, e acede aos IPC criados pelo servidor.

2.2 Planeamento

Após a análise inicial do problema, o grupo começou a discutir ideias de como implementar uma solução. Durante esta fase, o grupo explorou várias opções, nomeadamente a escolha dos métodos de comunicação que seriam utilizados.

Após uma análise das vantagens e desvantagens de cada opção, e com a consideração da sugestão do docente, o grupo decidiu utilizar filas de mensagens para os pedidos e respostas do servidor com os clientes, e memória partilhada para partilhar a lista de clientes ativos no servidor com todos os processos.

2.3 Ferramentas

Durante este trabalho foram utilizadas as seguintes ferramentas:

2.3.1 Linguagem C

Este trabalho foi implementado com a linguagem programação "C" sendo esta a linguagem utilizada no ambiente de desenvolvimento das aulas.

2.3.2 VScode

Para o desenvolvimento foi utilizado o IDE VScode

2.3.2 Ubuntu

Para a testagem e implementação do código, foi utilizada a máquina virtual com o sistema operativo Ubuntu Linux.

3. Implementação

Neste capítulo vamos analisar a estrutura do código implementado neste projeto, bem como analisar as funcionalidades mais importantes.

3.1 Estrutura do código

Este trabalho foi dividido em quatro ficheiros:

1. Basedados.h: Ficheiro *header* que contêm as funções declaradas para a utilização nos ficheiros cliente.c e server.c. Contêm também, todas as estruturas de dados e bibliotecas necessárias para o funcionamento do programa;
2. Basedados.c: Implementação das funções utilizadas nos ficheiros cliente.c e server.c ;
3. Cliente.c: Programa responsável pela implementação do servidor. Inicializa ambos os mecanismos de IPC e a lista de livros na base de dados. Por fim, recebe e responde aos pedidos de livros dos clientes;
4. Server.c: Programa responsável pela implementação do cliente. Pede *input* do utilizador para mandar pedidos ao servidor e demonstra o resultado desse pedido na consola;

3.1.1 Basedados.h

O ficheiro "basedados.h" começa com a inclusão de todas as bibliotecas necessárias para o funcionamento deste programa e a definição de algumas constantes.

```
#include <sys/ipc.h>
#include <sys/types.h>
#include <sys/msg.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <signal.h>
#include <unistd.h>
#include <sys/shm.h>
#include <sys/wait.h>

#define exit_on_error(s,m) if (s < 0) { perror(m); exit(1); }
#define PERMISSOES 0666 /* permissões para outros utilizadores */
#define MAXMSG 250 /* assumir que chega */
#define MAXUSERS 15
#define CHAVE_SHM 23
```

Figura 1- Bibliotecas & Constantes

Para além das bibliotecas mais *standart*, inclui-se bibliotecas para a utilização de mecanismos IPC sendo estas; *sys/ipc.h* (para a criação e manipulação de mecanismos de comunicação),*sys/msg.h (*para utilização de fila de mensagens), *sys/shm.h (*para utilização de memória partilhada) e *signal.h* (para utilização de sinais).

Após as bibliotecas, definimos as constantes a serem utilizadas nos programas:

- *Exit_on_error:* Constante fornecida pelo docente para terminar o programa caso um *int* retorne um valor negativo (querendo dizer que houve algum erro com os mecanismos IPC);
- *PERMISSOES*: Constante para determinar as permissões dos mecanismos IPC
- *MAXMSG*: Limite máximo para os arrays de char;
- *MAXUSERS*: Limite máximo de users que podem estar ativos no servidor;
- *CHAVE_SHM*: Chave a ser utilizada na criação e manipulação da memória partilhada;

Após as bibliotecas e constantes, define-se todas as estruturas de dados a serem utilizadas no programa.

```
typedef struct livro{
    int cod_livro;
    char autor[MAXMSG];
    char titulo[MAXMSG];
    float preco;
}LIVRO, *ptrLivro;

typedef struct nodeLivro{
    ptrLivro data;
    struct nodeLivro *next;
}NODELIVRO, *ptrNodeLivro;

typedef struct listaLivro{
    int NEL; //número de elementos na lista;
    ptrNodeLivro inic;
}LISTALIVRO, *ptrListLivro;

typedef struct cliente{
    long pid_cliente;
}CLIENTE;

typedef struct s_msg{
    long para; /* 1º campo: long obrigatório = PID destinatário*/
    int de; /* PID remetente*/
    char texto[MAXMSG];
    LIVRO livroPretendido;
}S_MSG;

typedef struct s_shm{
    CLIENTE onServ[MAXUSERS];
}S_SHM;
```

Figura 2- Structs

Na figura 2, podemos ver todas as estruturas que foram criadas:

- Livro: Estrutura para guardar informação sobre cada livro;
- nodeLivro: Estrutura para ser utilizada na lista ligada de livros. Contêm um ponteiro para o livro e outro ponteiro para o próximo elemento na lista;
- listaLivro: Estrutura da lista ligada de livros. Contêm o número de elementos na lista (NEL) e um ponteiro para o início da lista;
- cliente: Estrutura para guardar o PID dos processos que entram no servidor;
- s_msg: Estrutura para a comunicação entre processos pelo meio de fila de mensagens. Contêm o PID do processo destinatária, ou seja, para quem a mensagem se dirige (para), o PID do processo remetente, ou seja, o processo que envia a mensagem (de), um array de char para a mensagem (texto) e uma variável para enviar o livro pretendido por um cliente (livroPretendido);
- s_shm: Estrutura da memória partilhada que contém um array de clientes ativos no servidor (onServ). O array apenas aceita, até 15 utilizadores;

No final do ficheiro, declaramos todas as funções implementadas pelo ficheiro basedados.c para poderem ser utilizadas pelos ficheiros cliente.c e server.c.

```
ptrListLivro criarListaLivros();
ptrNodeLivro criarLivro();
int addLivroInic(ptrNodeLivro livro, ptrListLivro lista);
void showLista(ptrListLivro lista);
int carregarBaseDados(ptrListLivro lista);
void lerLivro(ptrNodeLivro livro);
int mygetline(char line[], int max);
int gravarBiblioteca(ptrListLivro lista);
ptrNodeLivro procurarLivro(int cod, ptrListLivro lista);
void showLivro(LIVRO livro);
void showPtrLivro(ptrNodeLivro livro);
void showOnServ(CLIENTE arr[]);
void inicServ(CLIENTE arr[]);
int addToServ(CLIENTE arr[], CLIENTE cliente);
int remFromServ(CLIENTE arr[], long pid);
void finishServer(CLIENTE arr[]);
int isEmpty();
void destruirLivro(ptrNodeLivro livro);
void destruirLista(ptrListLivro lista);
#endif
```

Figura 3- Funções

3.1.2 Basedados.c

O ficheiro basedados.c contêm todas as funções que são uteis a todos ficheiros. Todas são necessárias para o projeto funcionar corretamente, contundo, estas vão ser as funções abordadas neste relatório:

- finishServer();
- isEmpty();
- addToServ();

A primeira função, finishServ(), encarrega-se de terminar todos os clientes ativos no servidor.

```
//Termina todos os processos ativos no servidor
void finishServer(CLIENTE arr[]){
    for(int i = 0; i < MAXUSERS; i ++){
        if(arr[i].pid_cliente != 0){
            printf("Cliente [%ld] terminou\n", arr[i].pid_cliente);
            kill(arr[i].pid_cliente, SIGTERM);
        }
    }
}
```

Figura 4- finsihServ()

A função leva um array de clientes como argumento. Dentro da função, esse array é iterado. A função verifica se o pid do processo não é “0” , ou seja, existe um cliente, e envia um sinal para esse processo para terminar com SIGTERM.

A função faz a iteração 15 vezes, pois é o número máximo de utilizadores que podem estar no servidor a qualquer instante.

A seguir, existe a função *isEmpty*(). Esta função serve para determinar se existe uma base de dados ou não.

```
//Função para saber se a BD está vazia
int isEmpty(){
    int value = 0;

    /** return values:
     * 0 -> file does not exist
     * 1 -> file is empty
     * 2 -> file is not empty
    */
    FILE *f = fopen("livros.txt", "r");

    if(!f)  return 0;

    fseek(f, 0L, SEEK_END);

    if(ftell(f) == 0)
        value = 1;
    else
        value = 2;

    fclose(f);
    return value;
}
```

Figura 5- isEmpty()

A função começa por declarar uma variável auxiliar *value*. Em comentário, vemos que a função pode retomar 3 valores:

- 0: O ficheiro não existe. Neste caso a função repara que a ficheiro a tentar ser aberto não existe e retorna logo o valor de 0, fechando a função;
- 1: O ficheiro existe, mas está vazio. A função dá o valor de 1 à variável *value;*
- 2: O ficheiro existe e contêm já uma base de dados. A função dá o valor de 2 à variável *value;*

Se o ficheiro existir, usa-se a função *fseek()* para inserir o ponteiro no final do ficheiro. No final, a função fecha o ficheiro e devolve a variável *value*.

Por fim, *addToServ*() adiciona um cliente à lista de clientes no servidor.

```
//Adiciona um cliente ao servidor
int addToServ(CLIENTE arr[], CLIENTE cliente){
    for(int i = 0; i < MAXUSERS; i ++){
        if(arr[i].pid_cliente == 0){
            arr[i].pid_cliente = cliente.pid_cliente;
            return 1;
        }
    }
    //Se não for possível adicionar cliente
    return 0;
}
```

Figura 6- addToServ

A função leva 2 argumentos, um array de clientes e um cliente.

A função itere pelo array dos clientes e verifica se o índex já está a ser ocupado por algum cliente. Se o PID for igual a 0, quer dizer que não existe cliente naquela posição. A função, insere então, o cliente nesse índex disponível.

Se o ciclo iterar pelo array todo e não houver espaço no servidor, a função devolve o falar de 0 e o cliente não é inserido.

3.1.3 Server.c

O ficheiro *server.c* contêm o código para o funcionamento do servidor no programa

```c
#include "basedados.h"

/**
 * 25215 -> Guilherme Pedrinho
 * 20888 -> Francisco Meneses
*/

struct s_msg msg;
struct s_shm *listServ;
ptrListLivro biblioteca = NULL;

//Converter o ponteiro do livro escolhido para struct
LIVRO convert(ptrNodeLivro l2){
    LIVRO l1;

    if(!l2){
        l1.cod_livro = -1;
    }else {
        strcpy(l1.autor, l2->data->autor);
        strcpy(l1.titulo, l2->data->titulo);
        l1.preco = l2->data->preco;
        l1.cod_livro = l2->data->cod_livro;
    }

    return l1;
}

//Handles Ctr C
void sig_handler(int sig_num){
    int status;
    //Aceder à fila
    int id_msg=msgget(100, 0);
    exit_on_error(id_msg, "Erro ao aceder à fila\n");


    //Terminar todos os processos ativos no servidor
    if(fork()) {
        //Espera pelo processo filho terminar
        wait(NULL);
        //Eliminar a fila
        id_msg = msgctl(id_msg, IPC_RMID, NULL);

        //Aceder mem. partilhada
        int id_shm = shmget(CHAVE_SHM, 0, 0);
        exit_on_error(id_msg, "Erro ao aceder à mem. partilhada\n");

        //Eliminar mem. partilhada
        id_shm = shmctl(id_shm, IPC_RMID, NULL);
    }else {
        //Função que termina os servidores ativos no servidor
        finishServer(listServ->onServ);

        destruirLista(biblioteca);
        exit(0);
    }
    printf("\nServidor terminou..\n");

    exit(0);
}
```

Figura 7- início server.c

O programa começa por incluir o ficheiro *basedados.h*. Depois, define-se as variáveis globais.

O programa define uma função para converter um ptrNodeLivro (ponteiro que aponta para um livro) para a estrutura de dados LIVRO para possibilitar a partilha do livro pretendido entre o servidor e os clientes.

De seguida, o programa define como lidar quando o utilizador termina o processo. Neste caso, o processo cria um processo filho para terminar todos os clientes no

servidor. O processo pai espera pela terminação do filho e procede a eliminar todos os mecanismos de IPC criados.

A função *main* começa por declarar as variáveis que vão ser necessárias para a execução do programa.

```
int main(int argc, char *argv[]){
    int id_msg, r, id_shm;
    long para;
    char op;
    ptrNodeLivro livroPretendido = NULL;
    signal(SIGINT, sig_handler);


    biblioteca = criarListaLivros();
    msg.de = getpid();

    //Criar uma lista de clientes ativos no Serv partilhada por todos os processos
    id_shm = shmget(CHAVE_SHM, sizeof(struct s_shm), IPC_CREAT | PERMISSOES);
    exit_on_error(id_shm, "Erro ao tentar criar/aceder a mem. partilhada");

    listServ = (struct s_shm *) shmat(id_shm, NULL, 0);

    //Inicializar array de utilizadores a zero's
    inicServ(listServ->onServ);

    //Cria fila
    id_msg=msgget(100, IPC_CREAT | IPC_EXCL | PERMISSOES);
```

Figura 8- inicialização de variáveis e criação de IPC

No final das variáveis estarem declaradas, o programa cria todos os mecanismos de IPC (fila de mensagens e memória partilhada) e inicializa array de clientes ativos (metendo todos os índices a zero).

Após a criação de todos os IPC, o programa verifica a existência de uma base de dados.

```
//Criação da BD
if(isEmpty() == 1|| isEmpty() == 0){ //Se o ficheiro se encontrar vazio
    printf("BD não existente. Criar? (s/n)\n");
    scanf(" %c", &op);
    if(op == 'n'){
        id_msg = msgctl(id_msg, IPC_RMID, NULL);
        exit(1);
    }
    do{
        fflush(stdout);
        ptrNodeLivro L = criarLivro();
        lerLivro(L);
        addLivroInic(L, biblioteca);
        printf("Adicionar livro? (s/n): ");
        scanf(" %c", &op);
    }while(op != 'n' && op != 'N');
    gravarBiblioteca(biblioteca);

}else if(isEmpty() == 2){ //Se o ficheiro tiver informação
    carregarBaseDados(biblioteca);
    showLista(biblioteca);
    printf("Editar BD existente? (s/n)\n");
    scanf(" %c", &op);
    if(op == 's'){
        do{
            fflush(stdout);
            ptrNodeLivro L = criarLivro();
            lerLivro(L);
            addLivroInic(L, biblioteca);
            printf("Adicionar livro? (s/n): ");
            scanf(" %c", &op);
        }while(op != 'n' && op != 'N');

        gravarBiblioteca(biblioteca);
    }
}
```

Figura 9- Verificação da existência de base de dados

Se o ficheiro não existir ou não tiver conteúdo, o programa pede ao utilizador para inserir os dados dos livros (criando o ficheiro se ele não existir), gravando depois no ficheiro “livros.txt”.

Se o ficheiro tiver conteúdo, o utilizador tem a opção de adicionar livros ou deixar a base de dados como está.

Por fim, o servidor entra num ciclo infinito com dois processos. Um para receber as mensagens dos clientes e outro para responder aos pedidos.

```
while (1) {
    //Aguarda mensagens na fila
    para=1;
    r=msgrcv(id_msg, (struct msgbuf *) &msg, sizeof(msg)-sizeof(long),para,0);
    exit_on_error(r, "Erro ao receber mensagem\n");

    if(strcmp("hello", msg.texto) == 0){ //Cliente entrou no servidor
        printf("Cliente [%d] iniciou\n", msg.de);
        //showOnServ(listServ->onServ);
    }else if(strcmp("goodbye", msg.texto) == 0){ //Cliente terminou o programa
        printf("Cliente [%d] terminou\n", msg.de);

    }else{ //Cliente pediu um código
        printf("Cliente [%d] solicitou código [%s]\n", msg.de, msg.texto);

        //Criar um processo filho para responder ao pedido
        if(fork()==0){
            livroPretendido = procurarLivro(atoi(msg.texto), biblioteca);
            //Converter o ponteiro para uma struct a ser enviada
            msg.livroPretendido = convert(livroPretendido);
            msg.para = msg.de;
            strcpy(msg.texto, "Here is your data");
            r=msgsnd(id_msg, (struct msgbuf *)&msg,sizeof(msg)-sizeof(long), 0);
            exit_on_error(r, "Erro ao enviar resposta");
            exit(0);
        }
    }
}
```

Figura 10- pedidos e respostas

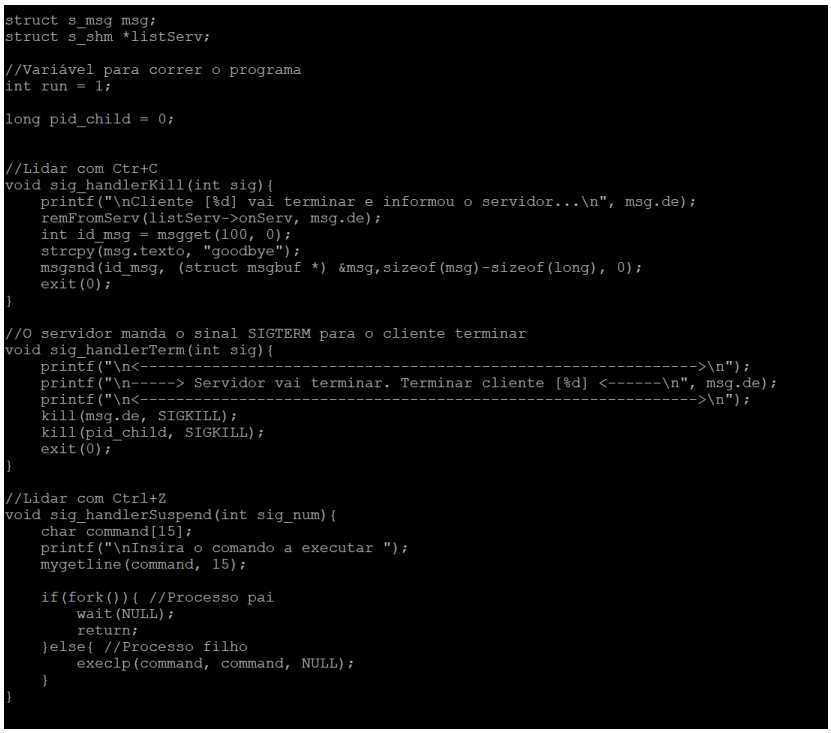
O processo pai encarrega-se de “ouvir” os pedidos dos clientes. Também verifica se as mensagens enviadas pelos clientes, são mensagens de auxílio para informar que o cliente entrou ou saiu da lista.

Quando um cliente solicita um código, o servidor cria um processo filho para ir procurar o livro pretendido e enviar resposta ao cliente, terminando logo de seguida.

O programa fica a correr até o utilizar enviar o sinal para o terminar

3.1.4 Client.c

O ficheiro *client.c* contêm o código para o funcionamento dos clientes.



```
struct s_msg msg;
struct s_shm *listServ;

//Variável para correr o programa
int run = 1;

long pid_child = 0;


//Lidar com Ctr+C
void sig_handlerKill(int sig){
    printf("\nCliente [%d] vai terminar e informou o servidor...\n", msg.de);
    remFromServ(listServ->onServ, msg.de);
    int id_msg = msgget(100, 0);
    strcpy(msg.texto, "goodbye");
    msgsnd(id_msg, (struct msgbuf *) &msg,sizeof(msg)-sizeof(long), 0);
    exit(0);
}

//O servidor manda o sinal SIGTERM para o cliente terminar
void sig_handlerTerm(int sig){
    printf("\n<------------------------------------------------------------->\n");
    printf("\n-----> Servidor vai terminar. Terminar cliente [%d] <------\n", msg.de);
    printf("\n<------------------------------------------------------------->\n");
    kill(msg.de, SIGKILL);
    kill(pid_child, SIGKILL);
    exit(0);
}

//Lidar com Ctrl+Z
void sig_handlerSuspend(int sig_num){
    char command[15];
    printf("\nInsira o comando a executar ");
    mygetline(command, 15);

    if(fork()){ //Processo pai
        wait(NULL);
        return;
    }else{ //Processo filho
        execlp(command, command, NULL);
    }
}
```

Figura 11-inicio client.c

Tal como no ficheiro *server.c*, o ficheiro *cliente.c* começa por declarar as variáveis globais e fazer o tratamento dos sinais.

O programa trata do sinal SIGTERM informando ao utilizador que vai terminar pela consola, removendo-se da lista de clientes no servidor e enviando uma mensagem ao programa do servidor a informar que vai terminar, terminando o processo no final.

Quando o servidor envia o sinal que vai terminar, o ficheiro *cliente.c* informa o utilizador que o servidor terminou, e termina também, “matando” o processo filho bem como o processo pai.

Por fim, trata do sinal SIGTSTP. Quando CTR+ Z é carregado, o processo para e permite ao utilizador executar um comando na Shell, resumindo de seguida o programa normal.

```c
int main(int argc, char *argv[]){
    int chave, id_msg, r, codigo, id_shm;
    long para = 1;
    char *line=NULL;
    //Inicializa informação necessária
    msg.para = 1;
    msg.de = getpid();
    //Inicializa cliente para adicionar à lista no servidor
    CLIENTE client;
    client.pid_cliente = getpid();
    //Lidar com os sinais
    signal(SIGTERM, sig_handlerTerm);
    signal(SIGTSTP, sig_handlerSuspend);

    // Obtem id_msg da fila
    id_msg=msgget(100, 0);

    //Se o servidor não tiver sido iniciado
    if(id_msg == -1){
        printf("\n----->O servidor que tentou contactar não se encontra ativo<-----\n\n");
        exit(1);
    }

    //Aceder à lista de clientes ativos no serv
    id_shm = shmget(CHAVE_SHM, 0, 0);
    exit_on_error(id_shm, "Erro ao tentar aceder mem. partilhada\n");
    listServ = (struct s_shm *) shmat(id_shm, NULL, 0);

    //Adicionar o cliente à lista do servidor
    if(addToServ(listServ->onServ, client) == 0){
        printf("Servidor cheio! \n");
        exit(0);
    }

    //Mensagem de auxilio para indicar quando o processo acede pela 1a vez ao servidor
    strcpy(msg.texto, "hello");
    r=msgsnd(id_msg, (struct msgbuf *) &msg,sizeof(msg)-sizeof(long), 0);
    exit_on_error(r, "Erro ao mandar mensagem inicial");
```

Figura 12- declaração e inicialização de variaveis

Dentro da função *main*, o programa declara as variáveis necessessárias e inicializa o cliente para ser adicionado á lista de clientes.

O programa tenta aceder à fila de mensagem e à memória partilhada, e caso algum falhe, o programa é interrompido com um erro.

O cliente é adicionado à lista de clientes no servidor e envia uma mensagem ao servidor para informar que entrou.

```
printf("\n================\n");
printf("Para terminar prima as teclas Ctr + C\n");
printf("Para executar comandos primas as teclas Ctr + Z\n");
printf("================\n\n");

if(fork()){
    signal(SIGINT, sig_handlerKill);
    msg.para = 1;
    msg.de = getpid();
    printf("Insira os códigos dos produtos e prima ENTER\n");
    while(run){
        r=mygetline(msg.texto, MAXMSG);
        r=msgsnd(id_msg, (struct msgbuf *) &msg,sizeof(msg)-sizeof(long), 0);
        exit_on_error(r, "Impossivel mandar mensagem");
    }
}else{  //Processo para receber as mensagens
    signal(SIGTSTP, SIG_IGN);
    para = getppid();
    pid_child = getpid();
    while(1){
        r=msgrcv(id_msg, (struct msgbuf *) &msg, sizeof(msg)-sizeof(long),para,0);
        exit_on_error(r,"Erro na leitura de mensagem");
        printf("DATA: ");
        if(msg.livroPretendido.cod_livro == -1){
                printf("NO DATA AVAILABLE\n");
            }else{
            showLivro(msg.livroPretendido);
        }
        printf("Cliente [%d]\n", getppid());
    }
}
return 0;
```

Figura 13- Pedidos de livros

O programa cliente é compostopor dois porcessos. O processo pai que fica à espera de input do utilizador e envia as mensagens ao servidor, e o processo filho que recebe as mensagens do servidor.

No processo pai, o programa envia a mensagem para o tipo 1 (variável *para*) para o servidor receber a mensagem.

O processo filho recebe as mensagens enviadas pelo servidor do tipo do PID do processo pai. Após receber o livro, se o código do livro for -1, é demonstrado ao utilziador que o livro existente. Caso contrário, é mostrada a informação do livro ao utilizador.

Semelhante ao *server.c*, o programa só acaba com o sinal do utilizador.

4.Dificuldades

A principal dificuldade do trabalho foi a comunicação correta de todos os processos. Após muitas tentativas e erros, conseguimos implementar uma solução que satisfez todos os requerimentos impostos.

5. Conclusão

Após a realização deste trabalho, ficámos a entender melhor os mecanismos de IPC demonstrados e lecionados na cadeira de Sistemas Operativos

Este trabalho serviu-nos também para arranjar soluções criativas para a implementação de comunicação e sincronização de processos num projeto futuro.