Licenciatura em Engenharia Informática

Sistemas Operativos 19/20

Programação em C para UNIX – Meta 1

Sistema MSGDIST

Trabalho Realizado por:

- Pedro Rodrigues 21280225
- Vítor Fabião 21271030

Índice

Introdução

Este trabalho prático de sistemas operativos consiste na implementação de um sistema de gestão e redistribuição de mensagens denominado **MSGDIST**. As mensagens são simples mensagens de texto e o sistema encarrega-se de aceitá-las, armazená-las e redistribui-las a quem estiver interessado nelas.

Este trabalho prático é realizado em C, para UNIX (LINUX), usando os mecanismos do sistema abordados nas aulas teóricas e práticas. O trabalho foca-se no uso correto dos mecanismos e recursos de sistema.

Ao longo desta **Meta 1** iremos abordar vários mecanismos, assim como os *pipes* anónimos, as variáveis de ambiente, etc, onde tudo isto será bem explicado ao longo do relatório.

Meta 1

Para a primeira meta foram requeridos 5 pontos que serão abordados nas próximas páginas, cada um dos pontos foi cumprido com sucesso e serão todos devidamente explicados para que não surjam duvidas. Após uma detalhada leitura e análise ao enunciado começamos por idealizar de um modo mais abstrato o funcionamento do sistema de gestão **MSGDIST.**

Primeiro ponto:

- Planear e definir as estruturas de dados responsáveis por gerir as definições de funcionamento no gestor e no cliente. Definir os vários *header files* com constantes simbólicas que registem os valores por omissão comuns e específicos do cliente e servidor bem como as estruturas de dados relevantes.

Neste primeiro ponto, teremos que já , em avanço, idealizar todos os dados que à partida vão ser necessários guardar e para isso usaremos estruturas visto que a comunicação entre gestor/clientes e gestor/verificador será feita com recurso a pipes (mais à frente serão mais bem aprofundados).

Numa analise/desenvolvimento inicial consideramos três estruturas das quais serão as mais relevantes para a gestão e armazenamento dos dados do sistema. Para as estruturas ocorreu-nos usar listas ligadas devido ao baixo uso de recursos do sistema e a sua simplicidade no armazenamento de dados. A primeira estrutura em consideração, foi a estrutura que irá armazenar todas as mensagens enviadas por clientes.

```
typedef struct Mensagem mensagem, *pmensagem;
struct Mensagem{
    ptopico topico; // Ponteiro da estrututa
    char corpo[1000],titulo[101]; // Guarda o
    int duracao; //guarda a duraçao da mensag
    pmensagem next_msg; //Ponteiro para a pro
};
```

Figura 1- Estrutura Mensagem

Esta estrutura de nome “Mensagem”, inicia o seu armazenamento de dados com um ponteiro para um outro tipo de estrutura de seu nome “Topico” (de seguida já será apresentada). De seguida são criados dois vetores de caracteres em que um com uma capacidade de 1000 caracteres irá guardar o corpo da mensagem e o outro com um tamanho para 101 caracteres irá guardar o título da mensagem. Perto do final da estrutura seguimo-nos com a criação de uma variável do tipo inteiro que guardará a duração da mensagem no sistema. Para finalizar, a estrutura terá um ponteiro que vai apontar para a próxima mensagem.

A estrutura seguinte, será a estrutura que vai servir para armazenar as principais relevantes informações de cada cliente.

```
typedef struct Pid pid,*ppid;
struct Pid{
    int pid;//Guarda o numero
    char username[10];// Guard
    ppid next_pid; //Ponteiro
};
```

Figura 2 – Estrutura Pid

Esta estrutura de nome “Pid”, começa com a criação de uma variável do tipo inteiro que vai guardar a informação do *process id (pid)* do cliente em questão. Para além de guardar o seu *pid*, vai igualmente guardar o *username* do cliente criando assim um vetor do tipo *char* com uma capacidade de 10 caracteres. Para finalizar, esta estrutura terá também um ponteiro para o próximo cliente.

Por último, mas não menos relevante a estrutura que anteriormente já foi mencionada que vai guardar todos os tópicos.

```
typedef struct Topico topico, *ptopico;
struct Topico{
    char topico[40];//Nome do topico
    ptopico next_topic; //Ponteiro para
};
```

Figura 3 – Estrutura Topico

Esta estrutura intitulada de "Topico", tem apenas um vetor do tipo *char* com capacidade para 40 caracteres que vai guardar o nome do tópico e contará também com um ponteiro para o próximo tópico.

Depois da análise da estrutura iremos então falar dos *header files* e a sua relevância para o sistema. Considerámos que para uma melhor gestão e organização teríamos de facto criar três *header files* dos quais são um para o gestor, um para o cliente e outro para as estruturas que estes vão partilhar.

Para o gestor.h (*header file* do gestor.c) existe a inicialização de 2 funções, das quais vão ser requeridas várias vezes, o *shutdown* e o *help*. A função *help* é uma função trivial que quando chamada apresentará simplesmente um painel de instruções com os comandos que o administrador poderá usar. Já a função shutdown tem uma simples instrução, o *exit (0)* que tem só como função sair imediatamente do processo gestor.

Este gestor.h tem um *include* do *header file "estrutura.h"* que tem lá as estruturas já anteriormente referidas.

O *header file "estrutura.h"* para além das estruturas que já foram apresentadas tem também as bibliotecas que irão ser necessárias para funções, sinais, *prints*, leituras, etc.

O *header file "clientes.h"* ainda não tem muita informação relevante, incluindo assim só o *header file "estrutura.h".*

Segundo ponto:

- Desenvolver a lógica de leitura das variáveis de ambiente do gestor e do cliente, refletindo-se nas estruturas de dados mencionadas no ponto anterior. Sugestão: usar as funções `getopt()`, `getsubopt()` e `getenv()`.

Relativamente ao segundo ponto da **Meta 1** a implementação do código em *gestor.c* foi feita do modo mais simples e leve, tendo usado funções do próprio compilador como o **getenv()** e o **getopt()**. Para este ponto tivemos que recorrer a algumas bibliotecas assim como *unistd.h*, *ctype.h*, *stdlib.h* e *stdio.h* visto que usaremos algumas funções destes *headers*.

No gestor está presente o código que vai permitir buscar informação relativamente as variáveis de ambiente **MAXNOT**, **WORDSNOT** e **MAXMSG**.

```
int value_maxnot = atoi(getenv("MAXNOT")); //Vai à procura da variável de ambiente com o nome MAXNO

printf("Value of the environment variable <MAXNOT>: %d\n",value_maxnot);

char *value_wordsnot = getenv("WORDSNOT");
if(value_wordsnot == NULL){//Vai à procura da variável de ambiente com o nome WORDSNOT, verifica se
printf("Environment variable <WORDSNOT> does not exist! So it will start with a default file\n");
}
if(value_wordsnot){
printf("Value of the environment variable <WORDSNOT>: %s\n",value_wordsnot);
}

int value_maxmsg = atoi(getenv("MAXMSG"));//Vai à procura da variável de ambiente com o nome MAXMSG
printf("Value of the environment variable <MAXMSG>: %d\n",value_maxmsg);
```

Figura 4 – Código do ponto 2 da Meta 1 (gestor)

Esta secção do código inicia-se com a declaração de uma variável do tipo inteiro que vai guardar o valor numérico da quantidade de palavras que são proibidas nas mensagens (variável **MAXNOT**). Para a obtenção deste numero, usaremos inicialmente a função **getenv()** que vai enviar como argumento a *string* "**MAXNOT**" para comparar com as variáveis de ambiente previamente declaradas e és gay devolver o valor dela. Quando devolver o valor da *string* "**MAXNOT**", a função **atoi()** vai converter para valor numérico que posteriormente ficará armazenada na variável *value_maxnot*.

Segue-se com a criação de um *array* de caracteres que visa guardar o nome do ficheiro no qual estão contidas todas as palavras proibidas que uma mensagem não poderá conter. Para isso usamos novamente a função **getenv()** que irá enviar a *string* "**WORDSNOT**" e então será devolvida a *string* com o nome do ficheiro que vai ficar guardado na variável *value_wordsnot*.

Concluísse então esta parte com a declaração de uma variável do tipo inteiro que vai guardar o valor numérico da quantidade de palavras máxima que uma mensagem que o gestor pode ter (variável **MAXMSG**). Para a obtenção deste numero, usaremos inicialmente a função **getenv()** que vai enviar como argumento a *string* "**MAXMSG**" para comparar com as variáveis de ambiente previamente declaradas e devolver o valor dela. Quando devolver o valor da *string* "**MAXMSG**", a função **atoi()** vai converter para valor numérico que posteriormente ficará armazenada na variável *value_maxmsg.*

Na parte do *cliente.c* o objetivo será então o cliente tentar logar-se colocando pela linha de comandos o que pretende, neste caso ***./clientes -u username-do-cliente***. Assim o cliente já esta a colocar a sua informação relativamente ao seu username na estrutura já referida de seu nome "Pid".

```c
int main(int argc,char **argv){
    int c;
    opterr = 0; //desativa as mensagens de erro da função getopt
    pid *cliente;

while((c = getopt(argc,argv,"u:"))!=-1){
    switch(c){
        case 'u':
        cliente->username = optarg;
        printf("Welcome, you're on %s\n", cliente->username);
        break;
        case'?':
        if (optopt == 'u')  // Esqueceu um argumento
            fprintf (stderr, "Opção '-%c' requer um username.\n", optopt);
        else if (isprint (optopt))
            fprintf(stderr, "Opção '-%c' desconhecida.\n", optopt);
        else
            fprintf(stderr, "Caractere '\\x%x' de opção desconhecido.\n", optopt );
        exit (1);
        default:
        printf("User not available");
        abort();
    }
}
return 0;
}
```

Figura 5 – Código do ponto 2 da Meta 1 (cliente)

Para este caso iremos usar a função **getopt()** que vai buscar os comandos que o utilizador vai colocar na linha de comandos. Se o utilizador colocar ./cliente -u com o seu *username* então ele irá ser aceite e o seu *username* vai ficar guardado numa estrutura do tipo *Pid* que foi declarado previamente em *estrutura.h*.

Caso o cliente não coloque o seu *username* aparecerá uma mensagem a indicar que é necessário colocar o seu *username*. Se o cliente não colocar a opção *-u* então será enviada uma mensagem de erro ao utilizador indicando que o caracter inserido é desconhecido.

Caso contrário o *username* do cliente não irá estar disponível, mas isso são assuntos para serem tratados na próxima meta.

Terceiro ponto:

- Iniciar o desenvolvimento da leitura de comandos de administração do gestor implementando a leitura e validação dos comandos e respetivos parâmetros e a implementação completa do comando `shutdown`.

Neste ponto relativamente à autenticação dos comandos decidimos usar a função **help()** por nós criada que simplesmente apresenta um conjunto de *printf's* dos quais contêm a informação toda relativamente a cada comando.

```
void help(){
    printf("Commands:\n");
    printf("\tTurn on/off the menssage filter-->%8s \n", "filter on / filter off");
    printf("\tUsers list-->%8s \n","users");
    printf("\tTopics list-->%8s\n","topic");
    printf("\tMenssages list-->%8s\n","msg");
    printf("\tMenssage list by topic-->%8s\n"," topic topic-in-question");
    printf("\tErase menssage-->%8s\n"," del menssage-in-question");
    printf("\tKick user-->%8s\n"," kick username-in-question");
    printf("\tShutdown the system-->%8s\n","shutdown");
    printf("\tErase topics without messages-->%8s\n\n","prune");
}
```

Figura 6 – Função help

Após esta função ser chamada, é declarado um vetor do tipo *char* com uma capacidade de 20 caracteres que vai guardar o comando inserido pelo administrador. O próximo passo é entrar num ciclo que aguarda pelo comando que o administrador inserir, sendo guardado no vetor previamente criado. Se a comparação entre o comando e a *string* (*String* esta que contem a palavra respetiva ao comando esperado) for igual a 0 (der match) então o comando é aceite e procede ao pedido, caso contrario o comando não é reconhecido e aguarda que o administrador coloque o correto. O administrador ao colocar o comando *shutdown* vai estar a chamar a função **shutdown()** como já foi no primeiro ponto apresentada, irá simplesmente terminar o programa com um exit(0).

Relativamente à interação com o administrador, o sistema de implementação de código vai contar com vetores dinâmicos, a função getline e a função strtok. Na figura 7 vamos poder observar a estruturação do código para o desenvolvimento da leitura de comandos.

Começamos então por alocar memória para um array de caracteres que vai acabar por conter o comando que vai ser inserido pelo administrador, declaramos também um ponteiro para um vetor declarado como token que vai ser usado como guarda palavras em cada posição.

Pedimos então, dentro de um ciclo while, ao administrador o comando que ele deseja inserir e usamos a função getline que vai guardar o comando, e se for o caso, igualmente o segundo argumento que poderá ser ou um tópico ou um utilizador, isto tudo como string. Seguidamente usamos a função strtok para separar os dois argumentos. Inicialmente o strtok vai separar o primeiro argumento, que está separado por um espaço do segundo argumento, e colocar na primeira posição do vetor token. De seguida, o strtok deixou um ponteiro para a segunda palavra e é apartir dai que vamos inserir o resto das palavras para o resto das posições do vetor. Feito isto, só falta as comparações para verificar se está tudo em conformidade com o que é suposto. Para isso, usamos a função strcmp para comparar uma string com outra, neste caso, vamos comparar a string da primeira posição do vetor (primeiro argumento, comando pedido) com um conjunto de strings, se coincidir então o programa executa o que o administrador quer, senão uma mensagem de comando não reconhecido será apresentada. Existem 3 casos em que é efetivamente necessário uma segunda verificação, o del, o topic e o kick, então aqui vai ser comparado a segunda posição do array com outra string ainda a definir na próxima meta.

```c
char *command;
size_t command_size = 20;
command = (char *) malloc(command_size * sizeof(char));
size_t characters;
int arguments = 2;
char *token[arguments];
if(command == NULL){
    perror("Unable to allocate command");
    exit(1);
}
while(1){
    printf("---->");//Espera o utilizador colocar o comando pretendido
    getline(&command, &command_size,stdin);
    arguments = 0;
    token[arguments] = strtok(command," ");
    while ( (token[++arguments] = strtok(NULL," ")) != NULL);

        if(strcmp("filter",token[0])==0){ //compara o comando com a palavra que
            if(strcmp("on\n",token[1])==0)
                printf("Filter is on\n");
            else if(strcmp("off\n",token[1])==0)
                printf("Filter is off\n");
            else
                printf("Command filter invalid\n");
        //Se for igual ao comando entao o comando é aceite
        }else
        if(strcmp(token[0],"users\n")==0){
            printf("Accepted command\n");
        }else
        if(strcmp(token[0],"topics\n")==0){
            printf("Accepted command\n");
        }else
        if(strcmp(token[0],"msg\n")==0){
            printf("Accepted command\n");
        }else
        if(strcmp(token[0],"topic")==0){
            printf("This is the topic you choose <%s>\n",token[1]);
        }else
        if(strcmp(token[0],"del")==0){
            printf("This is the message you choose to delete <%s>\n",token[1]);
        }else
        if(strcmp(token[0],"kick")==0){
            printf("This is the user you choose to kick <%s>\n",token[1]);
        }else
        if(strcmp(token[0],"shutdown\n")==0){
            shutdown(); // se o comando for shutdown entao chama a funcao shutd
        }else
        if(strcmp(token[0],"prune\n")==0){
            printf("Accepted command");
        }
        else
            printf("Command <%s> not recognized\n",token[0]);//Se o comando for
}
```

Figura 7 – Código do desenvolvimento de leitura de comandos

Quarto ponto:

- Preparar a ligação entre gestor e verificador de forma a permitir testar a funcionalidade do filtro com algumas palavras enviadas a partir do gestor.

Este foi o ponto que mais trabalho deu visto que é a primeira vez que estamos a trabalhar com *pipes*. Mas com o esforço e trabalho percebemos como poderia funcionar e implementamos o código necessário para que existisse a comunicação entre o gestor e verificador.
Resumidamente, um *pipe* é um meio de comunicação entre dois processos e para existir essa

comunicação mútua terá de haver pelo menos 2 *pipes*, em que um envia a informação do gestor ao verificador e outro do verificador ao gestor.

Cada *pipe* só tem um sentido para a viagem de informação, dai existirem dois, e para isso, uma das extremidades serve para escrita e do outro lado serve para a sua leitura.

A figura 7 indica de um modo simplista como a comunicação é feita estando as portas principais e relevantes para os envios de informação. As portas com uma cruz são as portas que serão fechadas para que não haja um desperdício de recursos no sistema.

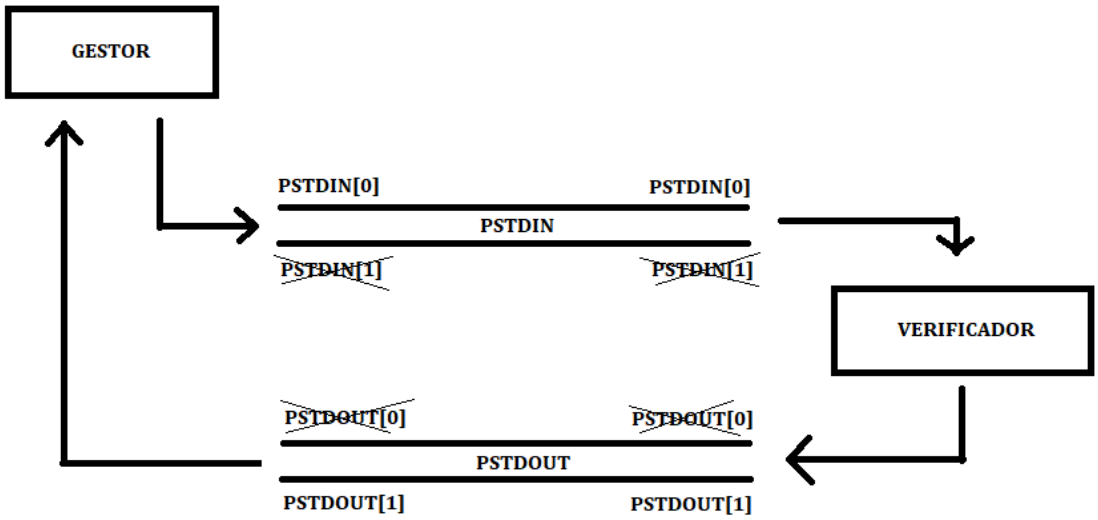

Figura 8 – Desenho geral do sistema MSGDIST entre gestor e verificador

Aprofundando melhor o comportamento que estes pipes vão necessitar para existir a sua comunicação do melhor modo e sem usar recursos desnecessários contamos agora com outro exemplo, figura 8, que demonstra o parte inicial quando são criados os pipes e o resultado de como o redireccionamento das saídas vai ter no final.

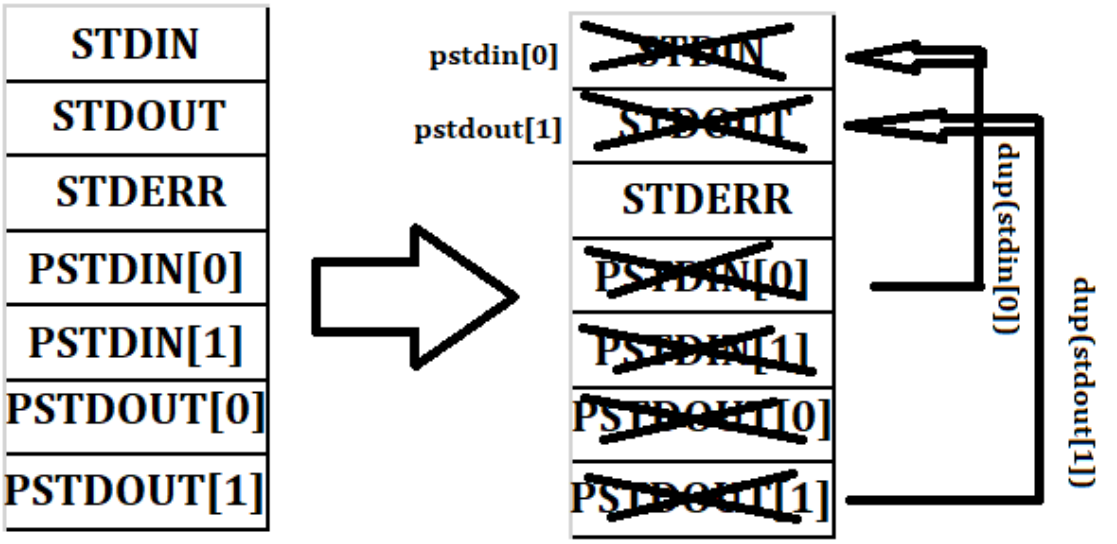

Figura 9 – Imagem de como o redireccionamento acontece

A figura 9 que vem em acompanhamento com a figura anterior vão ser ambas utilizadas para a explicação da ideologia do sistema em questão.

```
    int pstdin[2],pstdout[2]; //cria dois vetores de inteiros
pipe(pstdin); //cria o pipe responsavel por levar a info do pai para o filh
pipe(pstdout);//cria o pipe responsavel por levar a info do filho para o pa
int pid = fork(); //criação do filho e coloca o pid a 0
if (pid==0){//cria um novo processo duplicando o processo atual que o chama
    //input
    close(0); //Fecha stdin
    dup(pstdin[0]); //duplica o stdin do pipe que vai do pai para o filho e
    close(pstdin[0]); //depois da duplicaçao fechasse a posiçao original de
    close(pstdin[1]); // fechar igualmente o stdout do pipe que vai do pai

    //output do filho
    close(1); //Fecha o output
    dup(pstdout[1]); //duplica o stdout do pipe que vai do filho para o pai
    close(pstdout[1]); // depois da duplicaçao fechasse a posiçao original
    close(pstdout[0]); // fechar igualmente o stdout do pipe que vai do fil
    // IR BUSCAR A VARIAVEL DE AMBIENTE NOME DO FICHEIRO


    execl("verificador", "verificador", value_wordsnot, NULL); //substitui
    printf("ERROR in execl");//caso nao seja possivel a substituição entao

}
//pai
//ao inicio ele vai fechar aquilo que nao necessita para nao usar recursos
close(pstdin[0]); //fecha o stdin do pipe pai para filho
close(pstdout[1]); // fecha o stdout do pipe filho para pai
char mensagem[]="Esta cadeira de SO esta a ser muito facil ##MSGEND## \n";
char result[8]; // cria um vetor para o resultado que vier
write(pstdin[1], mensagem, sizeof(mensagem)); //envia a mensagem para o fil
close(pstdin[1]); //fecha a porta que utilizou ou seja o stdout do pipe do
int n=read(pstdout[0], result, sizeof(result)); //aguarda resposta e quando
close(pstdout[0]);//fecha a porta que utilizou ou seja o stdin do pipe do f
result[n-1]='\0'; //adiciona um /0 ao final da string
printf("O numero de palavras proibidas e: %s\n", result); // apresenta o re
```

Figura 10 – Código da comunicação entre gestor e verificador

Inicialmente criamos dois vetores de inteiros de tamanho 2, e logo de seguida os *pipes*. O *pipe pstdin* é o de ligação dos gestor ao verificador e o *pipe pstdout* é o de ligação entre o verificador e gestor (Como demonstra a figura x). Neste momento usamos a função **fork()** que tem como objectivo duplicar o processo que a chamou (pai criou um filho idêntico). O valor que a função retorna vai ser guardado na variável *pid* que à partida será igual a 0. Entramos então numa condição *if* em que se a variável *pid* estiver efetivamente igual a 0 então ele procede ao redireccionamento.

Este redireccionamento consiste em inicialmente fechar o *stdin* do processo filho que logo de seguida será ocupado pelo *stdin* do *pipe pstdin*, quando entrar a função **dup(pstdin[0])** que tem como objetivo criar uma cópia do file ***descriptor*** **(pstdin[0])**. Após este redireccionamento iremos fechar **o pstdin[0]** e o **pstdin[1]** para que não haja o uso desnecessário de *slots* permitindo assim espaço livre para caso seja necessário mais *pipes*.

De seguida trataremos então do *stdout* do filho e para isso fechamos desde inicio o seu *stdout* para que, como no caso anterior, quando o dup() criar a copia do file *descriptor*, essa cópia seja enviada para a *slot* que acabou de ficar livre. Depois eliminamos o original e o **pstdout[0]** visto que não vai ser utilizado.

Após a ligação entre o gestor e o seu filho estar operacional, chegou a hora de trocar esse processo idêntico do gestor pelo dito cujo verificador, através da função **execl("verificador","verificador",*value_wordsnot*, NULL).** Esta função recebe como primeiro argumento o caminho ate esse processo que queremos trocar, em segundo o argumento de 0, em terceiro o nome do ficheiro e em quarto o NULL.

Posto isto, temos então a ligação entre o gestor e o verificador e nada melhor que testar essa ligação enviando uma *string* pelo gestor e aguardar que o resultado que venha do outro lado (verificador) seja um inteiro com o número de palavras proibidas.

Já no gestor então começamos por fechar as saídas/entradas dos *pipes* que não serão relevantes para a comunicação. Para isso fechamos o **pstdin[0]** e o **pstdout[1]** e depois declaramos um vetor de caracteres que vai conter a mensagem que queremos enviar para verificação. Criamos também um vetor para receber a informação da verificação.

Posto isto, procedemos então ao envio da mensagem usando a função **write (pstdin[1]**, **mensagem**, **sizeof(mensagem))**, em que esta envia como argumentos, o local por onde a mensagem vai ser enviada **(pstdin[1])**, a mensagem em questão e o tamanho total dela. Após isto podemos fechar essa porta visto que não vai ser mais utilizada.

O processo fica em *stanby* à espera que a mensagem chegue através da função ***read*** **(pstdout[0], result, sizeof (result))**, que envia como argumentos o sitio por onde a informação vai ser recebida **(pstdou[0]),** e depois o vetor e o tamanho dele para que seja armazenada a informação que o verificador vai transmitir. Mais uma vez fechamos a porta porque não vai mais ser utilizada.

Por fim, colocamos um /0 no final da *string* que nos foi devolvida pelo verificador que é para reconhecer o final dela. Imprimimos o resultado da *string* para verificar se está tudo em ordem.

Quinto ponto:

- Desenvolver e entregar um *makefile* que possua os *targets* de compilação "all" (compilação de todos os programas), "cliente" (compilação do programa cliente), "gestor" (compilação do programa servidor), "verificador" (compilação do programa verificador) e "clean" (eliminação de todos os ficheiros temporários de apoio à compilação e dos executáveis).

Neste último ponto, a criação de um *makefile* é relativamente fácil em que iniciamos com o *all,* que vai conter o nome dos de todas as regras a serem executadas. Seguidamente, temos a parte em que tem o destino origem e a linha de comando que supostamente iriamos inserir manualmente. No destino origem vai haver um caminho desde o ficheiro principal ate à ultima estrutura. Para o clear basta inserir os comandos para remover os executáveis. Podemos ver isto tudo na próxima figura.

Conclusão

Interagir com um sistema operativo por vezes nem sempre é a tarefa mais fácil ou intuitiva, mas é com este tipo de trabalhos (faseados por metas) que um aluno consegue de alguma forma evoluir e aprender os mais diversos mecanismos disponíveis nestes sistemas para que um dia mais tarde possa fazer deles o uso mais correto e benéfico para si próprio.

Nesta **Meta 1**, aprendemos como dois processos conseguem comunicar entre si, através do uso de *pipes*, e igualmente o que são variáveis de ambiente. Consequentemente, conseguiu-se consolidar melhor a linguagem C, uma linguagem de programação que tem vindo a ser lecionada desde o primeiro ano deste curso.

Para concluir, esta primeira meta do trabalho serviu também para a primeira fase de estruturação do trabalho para que depois na ultima etapa a base onde iremos trabalhar já esteja moldada.