INSTITUTO SUPERIOR DE ENGENHARIA DE COIMBRA

INSTITUTO POLITÉCNICO DE COIMBRA

Licenciatura em Engenharia Informática 2º Ano – 1º Semestre 2021/2022

"SOBay"

Rafael Couto Nº 2019142454

Filipa Pimentel Nº 2018011698

COIMBRA
07 de janeiro de 2023

Índice

Introdução

A elaboração deste trabalho prático visa consolidar conhecimentos em sistemas operativos UNIX.

O objetivo é implementar uma plataforma de gestão para um sistema de leilões, "*SOBay*". Esta plataforma permitirá comunicação entre clientes e servidor, que gere itens à venda, verifica preços e determina quem adquire os itens.

Esta plataforma estará distribuída entre três programas essenciais:

- *Frontend:* programa responsável por utilizadores, que permitirá compra de itens, venda de itens e gestão de saldo. Cada utilizador será representado por um programa *frontend*.
- *Promotores:* programas responsáveis pelo lançamento de promoções. Apenas comunicam com o *backend*. No início da aplicação são lançados dois: um *promotor_oficial* e um *black_friday*.
- *Backend:* programa responsável pelo sistema, que permite toda a comunicação com clientes e promotores. Existe apenas uma instância deste programa a correr.

Estruturas de dados

- StructThreadCredentials

A estrutura de dados *"StructThreadCredentials"* é responsável por estabelecer a ponte de ligação entre todas as outras estruturas essenciais ao bom funcionamento do sistema. O principal objetivo da criação desta estrutura é servir de elo entre o programa principal e funções a correr em *thread* que necessitem de mais que uma estrutura. Existe apenas uma única estrutura.

```
typedef struct structThreadCredentials{
    struct backend *backend;
    struct user *user;
    struct item *item;
    struct promotor *promotor;
}StructThreadCredentials;
```

Figura 1- Estrutura StructThreadCredentials

- Backend

A estrutura de dados *"Backend"* é responsável por guardar os valores armazenados das variáveis ambientes e acima de tudo por armazenar o ponteiro para o *mutex*, consequentemente permitindo o bom funcionamento das *threads*. Existe apenas uma única estrutura.

```
typedef struct backend{
    int connectedClients;
    pthread_mutex_t *mutex;
    int maxPromoters;
    int maxItems;
    int maxUsers;
}Backend;
```

Figura 2- Estrutura Backend

- User

A estrutura de dados *"User"* é responsável por guardar todo os dados necessários para cada utilizador do programa.

Trata-se de uma lista ligada que é criada sempre que o programa *Backend* é iniciado e dado que o foco deste trabalho não são conceitos de programação C e poupança de recursos é alocada memória para o número máximo de utilizadores.

```
typedef struct user{
    char username[20];
    char password[20];
    int balance;
    int PID;
    int loggedIn;
    int heartbeating;
    struct user *next;
}User;
```

Figura 3- Estrutura User

- Comms

A estrutura *"Comms"* é responsável por armazenar todo o tipo de dados necessários para comunicações entre o programa *Backend* e o programa *Frontend*.

```
typedef struct comms{
    char username[20];
    char message[100];
    char argument[100];
    int balance;
    int buyID;
    int PID;
}Comms;
```

Figura 4- Estrutura Comms

- Item

A estrutura *"Item"* é responsável por armazenar todo o tipo de dados de um item do nosso programa. Fui desenvolvido numa perspetiva funcional e como tal usa a mesma lógica de criação da estrutura *"User"* e é alocado espaço para o número máximo de itens possíveis de gerir.

```
typedef struct item{
    int id;
    char name[20];
    char category[20];
    int basePrice;
    int buyNowPrice;
    int duration;
    int highestBid;
    char sellingUser[20];
    char highestBidder[20];
    int bought;
    struct item *next;
}Item;
```

Figura 5- Estrutura Item

Promotor

A estrutura *"Promotor"* é responsável por armazenar todo o tipo de dados de um promotor. Dado que todos os promotores são lançados em *thread*, guarda-se também na estrutura o seu *threadID* de modo que seja mais fácil posteriormente matar a *thread*, se necessário. De igual modo, funciona seguindo a mesma metodologia da estrutura *"User"* e *"Item"*, alocando memória para o total de promotores possíveis logo no início do programa *Backend*.

```
typedef struct promotor{
    char path[100];
    char category[20];
    int discount;
    int duration;
    int active;
    int threadID;
    int PID;
    struct promotor *next;
}Promotor;
```

Figura 6- Estrutura Promotor

Variáveis de Ambiente

```
export HEARTBEAT=30
export MAX_USERS=20
export MAX_PROMOTORS=10
export MAX_ITEMS=30
export FPROMOTERS="txt/fpromoters.txt"
export FUSERS="txt/fusers.txt"
export FITEMS="txt/fitems.txt"
```

Figura 7 – Variáveis Ambiente

As variáveis ambiente definidas tratam constantes necessárias ao bom funcionamento do programa *Backend* que são verificadas e executadas no início do programa.

Armazenam os valores limite para a criação das estruturas e também a diretório para os ficheiros usados para carregar informações para as estruturas.

Sinais

Em relação a implementações de sinais, foi dado uso dos sinais essencialmente para execução de funções. Relativamente ao *Backend*, foi configurado o *SIGINT* para execução da função *ctrlSignal()* que envia um sinal *SIGUSR1* para todos os *Frontend*, de modo a que sejam avisados que o *Backend* fechou e que serão igualmente interrompidos de desempenharem mais ações.

```
void crtlCSignal()
{
    for (int i = 0; i < frontendPIDArrayIndex; i++)
    {
        kill(frontendPIDArray[i], SIGUSR1);
    }
    quit(NULL);
}
```

Figura 8 – Função ctrlSignal()

Em ambos os casos executam funções de *quit()* que encerram os seus programas.

Funções como, *cancelPromotor()* e *quit()* dão uso aos sinais para matar *promotores* e *users*, respetivamente.

```
if (strcmp(prt->path, path) == 0)
{
    kill(prt->PID, SIGUSR1);
    waitpid(prt->PID, NULL, 0);
    printf("\n\t[+] Promotor thread killed '%s'\n\n", prt->path);
    prt->active = 0;
    notFound = 1;
    break;
}
```

Figura 9 – Excerto da função cancelPromotor()

Relativamente ao *Frontend*, são igualmente chamadas as funções *receiveSignal()* e *quit()*, ativadas através da receção de sinais *SIGINT* e *SIGUSR1*, respetivamente.

Threads

As threads são a grande base deste trabalho, tendo sido implementadas threads para todo o tipo de funções que necessitavam correr em background.

Dada a maior compreensão relativa aos conteúdos programáticos de threads, optou-se por fazer uso deste recurso no seu esplendor.

No *Backend*, permite que funções como *frontendComms()*, *promotorComms()*, *verifyCredentials()*, *itemActions()*, *verifyUserAlive()* e *removeUserNotAlive*. Todas estas funções incorporam ciclos infinitos necessários às verificações recorrentes do programa.

De igual modo, o *Frontend*, corre em thread as funções, *frontendCommadReader()*, *receiveMessages()*, *threadAlive()* e novamente, portenciam ciclos infinitos de verificações a realizar.

```c
void *receiveMessages(void *user_ptr)
{
    User *user = (User *)user_ptr;
    Comms comms;
    // Assure that all frontends receive the message at the same time
    sprintf(FRONTEND_FINAL_FIFO, FRONTEND_FIFO, user->PID);
    int comms_fd = open(FRONTEND_FINAL_FIFO, O_RDONLY);
    if (comms_fd == -1)
    {
        printf("\n\t[!] Error while opening pipe '%s' [func: receiveMessages]\n", FRONTEND_FINAL_FIFO);
        quit();
    }

    while (1)
    {
        int size = read(comms_fd, &comms, sizeof(comms));
        if (size == -1)
        {
            printf("\n\t[!] Error while reading from pipe '%s' [func: receiveMessages]\n", FRONTEND_FINAL_FIFO);
            quit();
        }
        if (size > 0)
        {
            if (strcmp(comms.message, "ItemTimedOut") == 0)
            {
                printf("\n\n\t[~] Item '%s' with ID '%d' timed out\n", comms.username, comms.buyID);
            }
            else if (strcmp(comms.message, "sold") == 0)
            {

            }
        }
    }
    pthread_exit((void *)NULL);
}
```

Figura 10 – Exemplo de função usada em Thread

Decisões de implementação

Alguns aspetos de implementação são passíveis de explicação e, portanto, relativamente à utilização de *threads* invés de *selects* deve-se única e exclusivamente ao facto de, a nosso ver, serem de menor complexidade de compreensão e implementação.

Para a questão de "informar todos os clientes que o *Backend* saiu", foi escolhido o uso de sinais por ser mais direto, "envia sinal, executa função para sair".

Em relação ao envio e receção de *Heartbeat*, recorre-se ao envio constante do *PID* do *Frontend*. Se este não enviar o hearbeat no tempo delimitado, o *Backend* coloca o user como *"logged out"* através da variável ambiente definida.

Como já mencionado acima neste documento, visto que o intuito do trabalho e da unidade curricular não é o uso eficiente de recursos, mas sim de mecanismos de sincronização com sinais e aplicações modelo cliente servidor, como esta. De modo a ter fácil acessibilidade a todos os elementos que compõem este programa, decidiu-se alocar a memória para as listas ligadas para o máximo de estruturas que estas podem albergar, e, portanto, não se olha à poupança de memória, mas sim ao bom funcionamento do programa.

Conclusão

A termo de conclusão, este trabalho é de tamanha importância para a compreensão de sistemas operativos na sua generalidade e em como tudo se desenrola por de trás do que é representado num ecrã.

Ainda de referir que o trabalho se torna muito mais fácil desde o momento em que tornam o tema interessante e sem aspetos desnecessários e ou sem aplicação de matérias lecionadas.

Em termos de conceitos usados, as *threads* foi de facto um aspecto valorizado e reconhecida a sua importância, bem como os *mutexes* que permitem bloquear conteúdos que possam ser acessados por duas ou mais *threads* em simultâneo. Igualmente os sinais são uma metodologia muito interesse em ambiente Unix de avisar um processo e ou até desempenhar alguma ação predefinida.

Os pipes também foi um conceito de excelente compreensão após a implementação no trabalho, tanto o seu funcionamento síncrono como assíncrono.

Por fim, salientar a importância desta unidade curricular no programa da Licenciatura em Engenharia Informática, que potencia o melhor conhecimento dos sistemas operativos.