



Sistemas Operativos

Trabalho Prático – MedicAlso

Docente: João Durães

Hugo Gabriel Carvalho Ferreira nº 2020128305 - LEI

Miguel Ângelo Rodrigues Ferreira nº 2020107016 - LEI

Coimbra, 17 de janeiro de 2022

Índice

Introdução	2
Estruturas necessária para o trabalho.....	2
Principais funções.....	3
Estratégias de Implementação	3
Makefile.....	4
Variáveis de Ambiente	5
Conclusão	5

Introdução

O trabalho pratico consiste na implementação de um sistema baseado num hospital(MedicAlso), o sistema tem como programa principal o balcão que trata de fazer a gestão de todo o hospital. Temos ainda o nosso programa utente, que é onde o utente interage com o balcão e mais tarde eventualmente com um especialista.

Estruturas necessária para o trabalho

```
// struct para utentes e medicos
struct pessoa
{
    char pNome[TAM_MAX];
    pid_t pid;
    int estado; // 0 para nao, 1 para sim
    char msg[TAM_MAX];
    char sintomas[TAM_MAX];
    char especialidade[TAM_MAX];
    int numConsulta;
    int prioridade;
    int tipoPessoa; // 1 para utente, 2 para medico
    int tempo;
} typedef Pessoa;

struct balcao
{
    int freq;
    int numUtentes;
    int numMedicos;
    Pessoa *utentes;
    Pessoa *especialistas;
} typedef Balcao;
```

Consideramos estas duas estruturas para a implementação do nosso trabalho prático, uma estrutura que guarda todas as informações relevantes sobre a pessoa(seja ela utente ou médico), e temos ainda uma estrutura “balcão” que guarda dois arrays dinâmicos(um de utentes e outro de médicos).

Principais funções

```
void reset(Pessoa *aux);
void removerPessoa(Balcao *aux, int pid, int tipoPessoa, int maxMedicos, int maxUtentes);
void adicionaNovaPessoa(Balcao *aux, Pessoa pessoa, int maxMedicos, int maxUtentes);
void reset(Pessoa *aux);
void removerPessoa(Balcao *aux, int pid, int tipoPessoa, int maxMedicos, int maxUtentes);
void resetTempo(Balcao *aux, int pid, int maxMedicos);
void *aumentarTempo(void *dados);
void atribuiConsulta(Balcao *aux, int maxMedicos, int maxUtentes);
void comUtentes(Balcao *aux, int maxMedicos, int maxUtentes);
void comEspecialistas(Balcao *aux, int maxMedicos, int maxUtentes);
bool delUtX(Balcao* aux, char *nome, int maxMedicos, int maxUtentes);
void *listaListas(void* dados);
void encerrar(Balcao *aux, int maxMedicos, int maxClientes);
int max(int a, int b);
```

Estas foram as funções que usámos para a implementação do nosso trabalho prático, com estas funções conseguimos fazer todo o tipo de gestão necessária no balcão sem qualquer problema.

Estratégias de Implementação

Para resolver os vários problemas propostos pelo trabalho prático decidimos desenvolver diferentes estratégias.

Para questão da comunicação, entre o balcão e o classificador criámos um unnamed pipe em que enviamos os sintomas do nosso utente, e ele devolve a especialidade e prioridade em que ele se insere. Em tudo o resto foi a base de fifos com associados ao pid, pela parte do utente e do médico. Já o balcão tem dois fifos que todos os programas sabem o nome, no entanto o sinal_fifo apenas serve para o médico enviar o “sinal de vida”.

Para o handle dos file descriptors, tanto no utente, no médico e no balcão para estarem preparados para “tudo” desenvolvemos mecanismos select.

Para guardar os nossos médicos e utentes decidimos criar dois arrays dinâmicos, um de médicos e outro de utentes, que apenas o programa balcão tem acesso.

Temos ainda duas threads no balcão, uma que nos mostra as listas de espera com uma frequência definida pelo comando freq "N" e outra que serve para estar constantemente a aumentar a variável tempo do médico.

Por último mas não menos importante, para implementar o "sinal de vida" criámos um fifo no balcão apenas para o médico mandar por lá, o médico manda um write a cada 20 segundos por uma thread implementada no médico. Caso o balcão não receba o sinal e o tempo do médico já esteja a 20 segundos o médico é removido do array e já não tem qualquer tipo de ligação com o programa.

Para não implementarmos um fifo para o utente comunicar com o balcão, e outro para o médico comunicar com o balcão, decidimos criar um valor tipoPessoa que define o tipo de pessoa que está a ser recebida, 1 para utente, 2 para médico, e desta forma conseguimos gerenciar tudo o que o balcão tem de fazer tanto para os utentes como para os médicos.

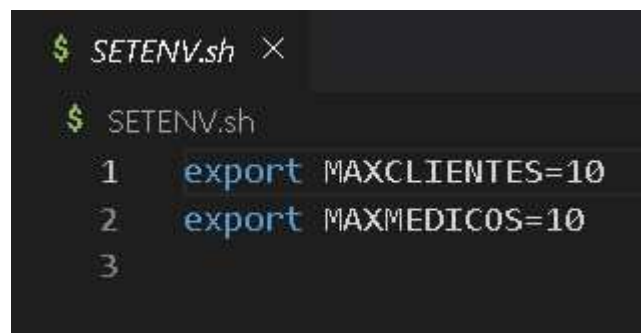
Makefile

```
M Makefile
1  default: all;
2
3  all: balcao.o utente.o medico.o
4
5  balcao.o: balcao.c
6      gcc -pthread balcao.c -o balcao
7
8  utente.o: utente.c
9      gcc utente.c -o utente
10
11 medico.o: medico.c
12     gcc -pthread medico.c -o medico
```

O nosso Makefile compila todos os programas necessários para o projeto.

Variáveis de Ambiente

Para as variáveis de ambiente criámos um “ficheiro” ao qual damos export antes de correr o programa `balcão`, para sabermos o número máximo de médicos e de utentes com que conseguimos lidar ao mesmo tempo.



```
$ SETENV.sh X
$ SETENV.sh
1 export MAXCLIENTES=10
2 export MAXMEDICOS=10
3
```

Conclusão

No geral gostámos muito de implementar este trabalho prático, adquirimos vários conhecimentos que ainda não tínhamos uma vez que é um tipo de programação com um “propósito diferente”, pensamos que cumprimos todos os requisitos pelo enunciado e que realizamos um bom projeto.