Universidade do Minho

Sistemas Operativos

Trabalho Prático

Controlo e Monitorização de Processos e Comunicação

João Freitas A74814

June 13, 2020

Índice

Índice de Imagens

Acrónimos

Named Pipes Pipes nomeados

Named Pipe Pipe nomeado

Pipes Anónimos Pipes anónimos

Pipe Anónimo Pipe anónimo

File Descriptor Descrito de ficheiro

Read Only Ler

Write Only Escrever

char* Array de caracteres

Process IDs Process IDs

Process ID Process ID

Handler Handler

Chapter 1

Introdução

Neste relatório irei expor o processo de desenvolvimento do trabalho prático da Unidade Curricular de Sistemas Operativos. Este trabalho teve como objetivo o desenvolvimento de dois elementos distintos, um servidor que execute tarefas e um cliente que as forneça.
O cliente e servidor deverão comunicar através de Named Pipes, sendo que o servidor tem também de fornecer uma série de funcionalidades como terminar tarefas em execução ou listar tarefas.

Chapter 2

Desenvolvimento

Neste capitulo vou explicar como é que cada uma das funcionalidades foi desenvolvida, começando pela comunicação entre servidor e cliente.

2.1 Named Pipes

A comunicação entre servidor e cliente é feita através de Named Pipes que permite a um cliente envie comandos para o servidor executar e ao servidor responder.
Para tal foi criada a função **createNewFifo** que, para além de criar um Named Pipe, irá eliminar o antigo se necessário.

```c
int createNewFifo(const char *fifoName)
{
    struct stat stats;
    if (stat(fifoName, &stats) < 0)
    {
        if (errno != ENOENT) // ENOENT is ok, since we intend to delete the file anyways
        {
            perror("stat failed"); // any other error is a problem
            return (-1);
        }
    }
    else // stat succeeded, so the file exists
    {
        if (unlink(fifoName) < 0) // attempt to delete the file
        {
            perror("unlink failed"); // the most likely error is EBUSY, indicating that some other process is using the file
            return (-1);
        }
    }

    if (mkfifo(fifoName, 0666) < 0) // attempt to create a brand new FIFO
    {
        perror("mkfifo failed");
        return (-1);
    }

    return (0);
}
```

Figure 2.1: Criação de Named Pipe

De seguida o servidor iria abrir o File Descriptor em Read Only enquanto o cliente iria abrir o mesmo File Descriptor em Write Only e escrever a mensagem.

```
if ((fd = open(fifo, O_RDONLY)) == -1)
{
    write(1, "Something went wrong\n", 22);
    return EXIT_FAILURE;
}
// Read from FIFO
read(fd, arr2, sizeof(arr2));
```

Figure 2.2: Leitura de FIFO

```
if ((fd = open(fifo, O_WRONLY)) == -1)
{
    write(1, "Something went wrong\n", 22);
    return EXIT_FAILURE;
}

write(fd, arr1, strlen(arr1) + 1);
close(fd);
```

Figure 2.3: Escrita em FIFO

Por fim tanto o servidor como o cliente trocam de lado, ou seja, o servidor abre o File Descriptor em Write Only de forma a enviar a resposta e o cliente faz o mesmo para Read Only para poder ler a resposta.

2.2 Menu

Como indicado no enunciado o cliente irá ter dois modos de funcionamento, através de um **Menu** ou através da shell. Para saber se é necessário correr através do **Menu** basta verificar o número de argumentos dados. Se não for dado nenhum argumento então irá funcionar através de um **Menu**, se for dado pelo menos um argumento irá funcionar através da shell.

```
if (argc < 2)
{
    char escolha[2];
    int aux;
    char time[4];
    int tempo = 0;
    while (1)
    {
        printMenu();
        read(0, &escolha, 1);
        escolha[1] = '\0';
        aux = atoi(escolha);
        if (aux >= 0 && aux < 8)
            break;
    }
```

Figure 2.4:

```
void printMenu()
{
    write(1, "-------MENU-------\n", 20);
    write(1, "1#- Mudar tempo de inactividade\n", 33);
    write(1, "2#- Mudar tempo de execução\n", 31);
    write(1, "3#- Executar uma tarefa\n", 25);
    write(1, "4#- Listar tarefas em execução\n", 34);
    write(1, "5#- Terminar tarefas em execução\n", 36);
    write(1, "6#- Histórico\n", 16);
    write(1, "7#- Ajuda\n", 11);
    write(1, "0#- Sair\n", 10);
    write(1, "------------------\n", 20);
    write(1, "Escolha a sua opção\n", 23);
}
```

Figure 2.5: Função que imprime menu

De qualquer das formas no fim irei ter um char* que irá ser enviado para o servidor.

2.3 Ajuda -h

Esta funcionalidade foi a mais simples de implementar, visto que não envolve a comunicação entre cliente e servidor. Basicamente, se a opção escolhida pelo utilizador for **-h** então irá imprimir as opções.

```
if (strcmp(option, "-h") == 0)
{
    write(1, "tempo-inatividade segs--> '-i n'\n", 34);
    write(1, "tempo-execucao segs--> '-m n'\n", 31);
    write(1, "executar comando--> '-e \"comando\"'\n", 36);
    write(1, "comandos em execução--> '-l'\n", 32);
    write(1, "terminar tarefa--> '-t n'\n", 27);
    write(1, "historico--> '-r'\n", 19);
    write(1, "ajuda--> '-h'\n", 15);
    exit(0);
}
```

Figure 2.6: Menu de Ajuda

2.4 Leitura dos inputs

Para fazer **parse** ao **Input** do utilizador foram implementadas as seguintes funções.

```
char *leArgumentos(char *arr2)
{
    char *args = malloc(sizeof(char) * 200);
    int j = 0;
    for (int i = 3; arr2[i] != '\0'; i++, j++)
    {
        args[j] = arr2[i];
    }
    args[j] = '\0';
    return args;
}
/**
 * Lê os comandos do array input
 */
char *leComando(char *arr2)
{
    char *comando = malloc(sizeof(char) * 2);
    comando[0] = '-';
    comando[1] = arr2[1];
    return comando;
}
```

Figure 2.7: Parse do **Input**

Estas função irão devolver um char* para o comando e um char* para o argumento, se este existir.

2.5 Tempos -i|-m

Estas funcionalidades centram-se em definir o tempo máximo de inatividade de comunicação em Pipes Anónimos **-i** e tempo máximo de execução de uma tarefa **-m**.
Para tal, foram criadas duas variáveis globais, **inatividade** e **exec**, que irão conter o tempo em segundos para cada uma dessas funcionalidades.

```
int inatividade = 10;
int exec = 20;
```

Figure 2.8: Variáveis globais

Se o comando for **-i** ou **-m** essas variaveis globais irão ser alteradas.

```
read(fd, arr2, sizeof(arr2));
char *comando = leComando(arr2);

if (strcmp(comando, "-i") == 0)
{
    char *args = leArgumentos(arr2);
    tempoInatividade(args);
    strcpy(resposta, "Inatividade=");
    strcat(resposta, args);
    strcat(resposta, "\n");
    free(args);
}
else if (strcmp(comando, "-m") == 0)
{
    char *args = leArgumentos(arr2);
    tempoExec(args);
    strcpy(resposta, "Execução=");
    strcat(resposta, args);
    strcat(resposta, "\n");
    free(args);
}
```

Figure 2.9: Mudança dos tempos

2.6 Tarefas

Tendo em conta as seguintes funcionalidades, achei que todas as tarefas iniciadas pelo servidor deviam ser guardadas numa estrutura, sendo estas posteriormente guardadas num ficheiro. Nessa estrutura tenho 4 campos.

- **int Tarefa**–> Inteiro que identifica uma tarefa
- **char* comando**–> Comando da tarefa
- **char* estado**–> Estado atual da tarefa
- **int pid[20]**–> Lista de Process IDs de uma tarefa

```
typedef struct Tarefa
{
    int tarefa;
    char *comando;
    char *estado;
    int pid[20];
} Tarefa;
```

Figure 2.10: Estrutura tarefa

2.7 Listar Tarefas -r|-l

Para listar as tarefas usei as seguintes funções que percorrem o ficheiro *tarefas.bin* e sempre que encontram um estado relevante para a funcionalidade em questão devolvem um char* contendo a informação necessária para essa mesma.

```
char *leHistorico()
{
    int fd = open("tarefas.bin", O_RDONLY, 0644);
    lseek(fd, 0, SEEK_SET);
    ssize_t readByte = 0;
    int byte = 1;
    char *buffer = malloc(sizeof(char) * 10000);
    Tarefa t;
    while (byte > 0)
    {
        byte = read(fd, &t, sizeof(struct Tarefa));
        readByte += byte;
        if (byte == 0)
        {
            break;
        }
        else
        {
            if (strcmp(t.estado, "concluida") == 0 || strcmp(t.estado, "max inatividade") == 0 ||
            strcmp(t.estado, "terminado") == 0 || strcmp(t.estado, "max execução") == 0)
            {
                char str[100];
                sprintf(str, "%d", t.tarefa);
                strcat(buffer, "#");
                strcat(buffer, str);
                strcat(buffer, ", ");
                strcat(buffer, t.estado);
                strcat(buffer, ": ");
                strcat(buffer, t.comando);
                strcat(buffer, "\n");
            }
        }
    }
    close(fd);
    return buffer;
}
```

Figure 2.11: Historico das tarefas terminadas

```c
char *listagem()
{
    int fd = open("tarefas.bin", O_RDONLY, 0644);
    lseek(fd, 0, SEEK_SET);
    ssize_t readByte = 0;
    int byte = 1;
    char *buffer = malloc(sizeof(char) * 10000);
    Tarefa t;
    while (byte > 0)
    {
        byte = read(fd, &t, sizeof(struct Tarefa));
        readByte += byte;
        if (byte == 0)
        {
            break;
        }
        else
        {
            if (strcmp(t.estado, "executar") == 0)
            {
                char str[100];
                sprintf(str, "%d", t.tarefa);
                strcat(buffer, "#");
                strcat(buffer, str);
                strcat(buffer, ": ");
                strcat(buffer, t.comando);
                strcat(buffer, "\n");
            }
        }
    }
    close(fd);
    return buffer;
}
```

Figure 2.12: Listagem de tarefas em execução

2.8 Terminar Tarefas -t

Para terminar uma tarefa é necessário 3 passos.

- Encontrar a tarefa em questão
- Terminar todos os seus Process IDs
- Alterar o seu estado

2.8.1 Encontrar a tarefa

Para encontrar uma tarefa através do seu identificador é necessário percorrer o ficheiro *tarefas.bin* e verificar se o inteiro fornecido é igual ao identificador da tarefa lida.

```
Tarefa terminate(int i)
{
    int fd = open("tarefas.bin", O_RDWR, 0644);
    ssize_t readByte = 0;
    int byte = 1;
    Tarefa t;
    while (byte > 0)
    {
        byte = read(fd, &t, sizeof(struct Tarefa));
        readByte += byte;
        int num = t.tarefa;
        if (byte == 0)
        {
            break;
        }
        if (num == i)
        {
            break;
        }
    }
    close(fd);
    return t;
}
```

Figure 2.13: Função que devolve uma tarefa através do seu inteiro

2.8.2 Terminar todos os seus Process IDs

De seguida termino todos os Process IDs excepto o primeiro.

```
char *terminar(Tarefa t)
{
    char *res = malloc(sizeof(char) * 50);
    if (t.comando != NULL)
    {
        int i;
        for (int j = 1; t.pid[j] != -1; j++)
        {
            i += kill(t.pid[j], SIGTERM);
        }
        char str[100];
        sprintf(str, "%d", t.tarefa);
        strcpy(res, "Tarefa #");
        strcat(res, str);
        strcat(res, " Terminada\n");
    }
    else
    {
        strcpy(res, "Tarefa inválida\n");
    }
    return res;
}
```

Figure 2.14: Função que termina todos os Process IDs exceto o do pai

2.8.3 Alterar o seu estado

Por fim termino o pai, altero o estado da tarefa para **terminado** e escrevo essa nova tarefa no ficheiro *tarefas.bin*.

```
else if (strcmp(comando, "-t") == 0)
{
    char *args = leArgumentos(arr2);
    Tarefa t2 = terminate(atoi(args));
    resposta = terminar(t2);
    kill(t2.pid[0], SIGTERM);
    t2.estado = "terminado";
    alteraLog(t2, t2.tarefa);
    free(args);
}
```

Figure 2.15: Passos finais

```c
int alteraLog(Tarefa t, int i)
{
    int fd = open("tarefas.bin", O_RDWR, 0644);
    ssize_t readByte = 0;
    int byte = 1;
    Tarefa t2;
    while (byte > 0)
    {
        byte = read(fd, &t2, sizeof(struct Tarefa));
        readByte += byte;
        int num = t2.tarefa;
        if (byte == 0)
        {
            break;
        }
        if (num == i)
        {
            lseek(fd, -byte, SEEK_CUR);
            write(fd, &t, sizeof(struct Tarefa));
            break;
        }
    }
    close(fd);
    return 0;
}
```

Figure 2.16: Reescrever a tarefa no ficheiro

2.9 Executar Tarefas -e

Para executar uma tarefa é criado um Process ID filho que irá gerir a execução dessa tarefa, deixando então o Process ID pai receber e executar outros comandos dados pelos cliente. Esse Process ID filho irá também criar os seus próprios filhos para executar cada comando da tarefa.
Mais uma vez são necessários 3 passos para executar uma tarefa.

- Criar tarefa
- Executar Tarefa
- Alterar o seu estado

2.9.1 Criar tarefa

Para criar uma tarefa é necessário criar a estrutura.

```
else if (strcmp(comando, "-e") == 0)
{
    char *args = leArgumentos(arr2);
    pid_t pid = fork();
    Tarefa t;
    t.tarefa = tarefa;
    t.comando = args;
    t.estado = "executar";
    if (pid == 0)
    {
        t.pid[0] = getpid();
        for (int i = 1; i < 20; i++)
        {
            t.pid[i] = -1;
        }
        executar(args, t);
        free(args);
        exit(1);
    }
    else
    {
        char str[100];
        sprintf(str, "%d", t.tarefa);
        strcpy(resposta, "nova tarefa #");
        strcat(resposta, str);
        strcat(resposta, "\n");
        tarefa++;
    }
}
```

Figure 2.17: Criação da estrutura e filho

Como indica a imagem em cima, o primeiro Process ID de uma tarefa é o "pai" dessa mesma, sendo que é este processo que vai gerir toda a execução desta tarefa.

2.9.2 Executar Tarefa

Antes de começar a executar a tarefa é necessário dividir os argumentos dados, sendo que essa divisão é feita pelo caracter | que indica um Pipe Anónimo.

```
void executar(char *args, Tarefa t)
{
    char *cmdstrdup = strdup(args);
    char *saveptr;
    char *ptr;
    char *argv[20];
    int argc;

    if ((ptr = strtok_r(cmdstrdup, "|", &saveptr)) == NULL)
    {
        printf("%s\n", "no args given");
        return;
    }

    argv[argc = 0] = cmdstrdup;
    while (ptr != NULL)
    {
        ptr = strtok_r(NULL, "|", &saveptr);
        if (++argc >= 20 - 1) // -1 for room for NULL at the end
            break;
        argv[argc] = ptr;
    }
    alarm(exec);
    piping(argv, argc, t);
    t.estado = "concluida";
    alteraLog(t, t.tarefa);
    free(cmdstrdup);
}
```

Figure 2.18: Parse dos argumentos

Nota: Antes de executar a função piping temos ***alarm(exec)****, que vai ser explicado na próxima secção.*

Por fim temos a função *piping* que irá adicionar a tarefa ao ficheiro e executar o comando. Para executar o comando irá criar no mínimo um Process ID filho sendo que irão ser utilizados Pipes Anónimos para gerir os inputs e outputs de cada comando.

```c
void piping(char *argv[], int argc, Tarefa t)
{
    adicionaLog(t);
    int fd[2];
    pid_t pid;
    int fdd = 0; /* Backup */
    int i = 0;
    while (i < argc)
    {
        if (pipe(fd) == -1)
        {
            perror("Pipe failed");
            exit(1);
        }
        if ((pid = fork()) == -1) ···
        if (pid == 0)
        {
            if (i < argc - 1) ···
            else
            {
                int j;
                for (j = 0; t.pid[j] != -1; j++)
                    ;
                t.pid[j] = getpid();
                alteraLog(t, t.tarefa);
                dup2(fdd, 0);
                if (argv[i + 1] != NULL)
                {
                    dup2(fd[1], 1);
                    dup2(fd[1], 2);
                }
                close(fd[0]);
                execCom(argv[i]);
                perror("Comand Failed");
                exit(EXIT_FAILURE);
                i++;
            }
        }
        else
        {
            wait(NULL); /* Collect childs */
            close(fd[1]);
            fdd = fd[0];
            i++;
        }
    }
}
```

Figure 2.19: Execução de comandos

2.9.3 Alterar o seu estado

Por fim o estado da tarefa muda para **concluída** sendo que irá ser utilizado o mesmo processo para alterar a tarefa no ficheiro *tarefas.bin*.

2.10 Timeout Execução e Inatividade

Para a funcionalidade do tempo máximo de execução de uma tarefa é utilizado **SIGALRM** com um Handler e a variável tempo de execução **exec** previamente mencionada.

```
signal(SIGALRM, timeout);
```

Figure 2.20: Sinal

Este alarme está presente antes da função *piping* que faz com que a função *piping* tenha no máximo **exec** segundos para completar a sua execução.
Se não tiver acabado então o Handler irá acabar a execução e alterar o estado da tarefa em questão.

```
void timeout(int signum)
{
    int f = 0;
    int fd = open("tarefas.bin", O_RDWR, 0644);
    ssize_t readByte = 0;
    int byte = 1;
    int *num;
    Tarefa t;
    while (byte > 0)
    {
        byte = read(fd, &t, sizeof(struct Tarefa));
        readByte += byte;
        num = t.pid;
        if (byte == 0)
        {
            break;
        }
        for (int i = 0; num[i] != -1; i++)
        {
            if (num[i] == getpid())
            {
                f = 1;
            }
        }
        if (f == 1)
            break;
    }
    close(fd);
    if (f == 1)
    {
        t.estado = "max execução";
        alteraLog(t, t.tarefa);
        int i = 0;
        for (; num[i] != -1; i++)
            ;
        for (i--; i >= 0; i--)
        {
            kill(num[i], SIGTERM);
        }
    }
}
```

Figure 2.21: Handler

Para o timeout por inatividade é criado um processo que corre ao mesmo tempo que o processo que executa um comando. Tal como o timeout por execução é criado um handler e um sinal que obriga um certo processo a ser executado antes do timeout por inatividade ser atingido

```c
if (pid == 0)
{
    if (i < argc - 1)
    {
        signal(SIGALRM, timeout2);
        pid_t pid2;
        if ((pid2 = fork()) == -1)
        {
            perror("fork failed");
            exit(1);
        }
        if (pid2 == 0)
        {
            int j;
            for (j = 0; t.pid[j] != -1; j++)
                ;
            t.pid[j] = getpid();
            t.pid[j + 1] = getppid();
            alteraLog(t, t.tarefa);
            dup2(fdd, 0);
            if (argv[i + 1] != NULL)
            {
                dup2(fd[1], 1);
                dup2(fd[1], 2);
            }
            close(fd[0]);
            execCom(argv[i]);
            perror("Comand Failed");
            exit(EXIT_FAILURE);
            i++;
        }
        else
        {
            alarm(inatividade);
            waitpid(pid2, 0, WUNTRACED);
            exit(0);
        }
    }
```

Figure 2.22: Timeout Inatividade

Chapter 3

Resultados

Por fim obti os seguintes resultados.

3.1 Ajuda -h

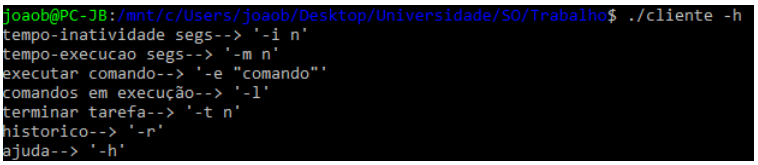

Figure 3.1: Resultado do comando -h

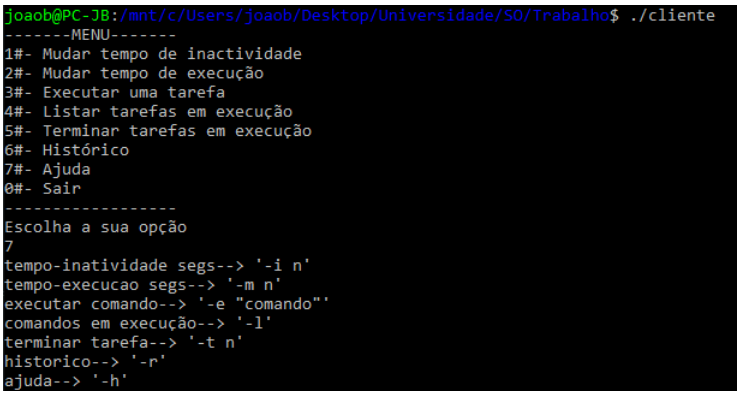

Figure 3.2: Ajuda através do menu

Por simplicidade o resto dos comandos vão ser corridos através da **shell**, apesar de que é possível fazer exatamente o mesmo através do menu.

3.2 Executar Tarefas -e

Como podemos ver é possível correr uma tarefa com um ou vários comandos encadeados.

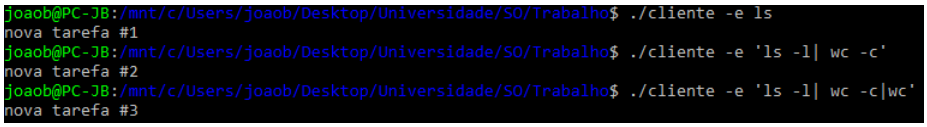

Figure 3.3: Execução de tarefas

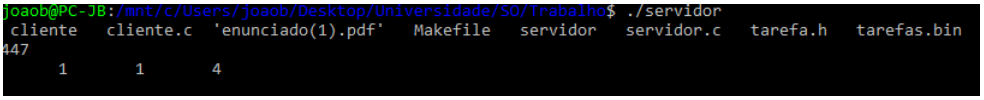

Figure 3.4: Resultado das tarefas

3.3 Resultado final

Por fim temos um exemplo onde o primeiro comando termina porque passou os 20 segundos máximos para executar uma tarefa, o segundo termina porque o cliente pediu para terminar e o terceiro que executou com sucesso.

Podemos também ver os resultados dos comandos **-l** e **-r**.

Figure 3.5: Comandos executados

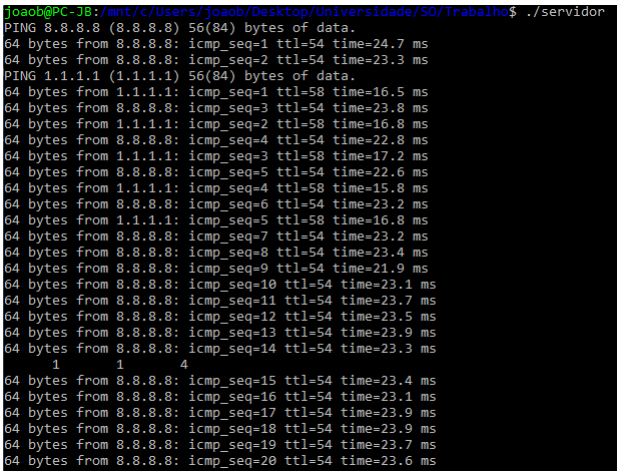

Figure 3.6: Outputs dos comandos

Chapter 4

Conclusão

Em suma, sinto que o trabalho foi bem sucedido visto que consegui implementar todas as funcionalidades básicas pedidas no enunciado, ficando apenas em falta a funcionalidade avançada. Acho que seria capaz de resolver o problema na sua totalidade, mas devido à carga de trabalhos deste semestre isso não foi possível. Na minha opinião, este trabalho é um bom complemento à unidade curricular pois permite que os alunos se familiarizem com os processos utilizados em Sistemas Operativos, tanto as coisas mais particulares, como as system calls e sinais, assim como o mais geral, planeamento e estruturação de projetos.