Relatório

O presente software foi desenvolvido no contexto académico e deve ser utilizado sem qualquer garantia por conta e risco do utilizador.
Daniel Torres <a17442|at|alunos.ipca.pt>
http://www.github.com/nargotik

Parte 1

Em quase todos os exemplos do trabalho é utilizada a função stat para verificação se um ficheiro existe e que tipo de ficheiro se trata (se é directório ou ficheiro).
Para simplificar esta tarefa foram criadas algumas funções tais como **directory_exists**, **file_exists**.

mostra

Este exemplo resume-se basicamente à utilização da função read()
Para a utilização desta função é necessária a inclusão do header **unistd.h**

read(int *fd*, *void* ***buf***, *size_t count);**
A função é chamada com 3 parametros.

- **Parâmetro 1** é o file handler ou file descriptor
- **Parametro 2** é a variável que vai armazenar a informação (buffer)
- **Parametro 3** é o tamanho de informação pretendida (tamanho do buffer)

A cada utilização da função read a mesma retorna o numero de bytes lidos ou no caso de negativo ou 0 indica um erro ou o fim da leitura

Foi utilizado neste exemplo o write utilizando o **STDOUT_FILENO** de forma a mostrar que o stdout trata-se de um "file descriptor" com um valor que para simplificação de leitura pois por norma têm o valor de 1

```
write(STDOUT_FILENO,&buff, readed_bytes);
```

---

conta

Este exemplo baseia-se na analise do buffer, tendo sido optado manter um buffer de 1 byte de forma a que fosse possível contar o numero de vezes que o caracter **\n** está presente no ficheiro

```
#define NEW_LINE '\n'
...
while ((readed_bytes = read(fd,&buff,1))) {
   if (buff == NEW_LINE) linhas++;
```

Para escrever para o stdout for utilizado o printf devido a haver apresentação de valore, no entanto poderia ter sido utilizado o write de uma variavel de texto após a utilização do **sprintf** tal como é mostrado abaixo

```
sprintf(variavel,"O ficheiro %s têm %d linhas",argv[1] , linhas);
write(STDOUT_FILENO,variavel,sizeof(variavel));
```

---

acrescenta

Neste exemplo foram abertos 2 ficheiros o **fd1** e **fd2**
O **fd1** foi aberto para escrita e o cursor foi posicionado no final do ficheiro, aproveitando o posicionamento para verificação do tamanho do ficheiro.

```
// Abre o ficheiro para escrita
fd1 = open(argv[1], O_WRONLY);
// Coloca o posicionador no final do ficheiro e verifica o tamanho
size_t tamanhof1 = (size_t)lseek(fd1,0,SEEK_END);
```

O **fd2** foi aberto para leitura e o cursor foi posicionado no final do ficheiro, para verificar o tamanho do ficheiro e voltado a colocar no inicio do ficheiro.

```
// Abre o ficheiro 2 para leitura
fd2 = open(argv[2], O_RDONLY);
// Coloca o apontador no fim do ficheiro e vê o tamanho
size_t tamanhof2 = (size_t)lseek(fd2,0,SEEK_END);
// Recoloca o apontador no inicio
lseek(fd2,0,SEEK_SET);
```

A variável **tamanhof2** pode ser utilizada para verificação de que o numero de bytes escritos corresponda ao total de bytes do **fd2**

```
// Enquanto o numero de bytes lidos não fôr zero ...
while ((readed_bytes = read(fd2, &buff, sizeof(buff)))) {
     // Escreve no ficheiro fd1 o buffer de readed_bytes
     writed += write(fd1, &buff, readed_bytes);
}
```

---

lista

Exemplo que visa utilizar praticamente a função **opendir** e **readdir**, de notar que o handler do opendir() têm uma struct do tipo DIR.

```
while ((entrada = readdir(dir))) {
  switch (entrada->d_type) {
        case DT_REG: // DT_REG - A regular file.
            tipo[0] = 'F';
            break;
        case DT_DIR: // DT_DIR - A directory.
            tipo[0] = 'D';
            break;
        default:
            tipo[0] = 'U';
  }
  printf("[%c]\t %s \t\n",tipo[0], entrada->d_name);
}
```

foi também utilizado o **entrada->d_type** para mostrar ao utilizador se se trata de um ficheiro ou directório.

Parte 2

interpretador

Para evitar a utilização do **scanf** foi efetuada a leitura directa do STDIN através da função read.

```
// Leitura dos comandos através do read()
while((readed = read(STDIN_FILENO, &buffer, 1)) > 0) {
    if (buffer[0] == NEW_LINE) {
        // Se for uma newline sai fora do while read
        command[i] = '\0';
        break;
    } else {
        command[i] = buffer[0];
        i++;
    }
}
```

Após a leitura de um comando é realizada a separação do comando para um array de caracteres pelo espaço como podem ver no exemplo abaixo

```
// Separa o comando para um array de argumentos o args[0] 'e o comando
char ** args = separa(command);
```

Após esta separação temos na variável args[0] o comando a executar e em todas as outras os argumentos a passar para pode correr o comando com a função **execvp(args[0], args)**.

De salientar que esta função substitui o actual “runtime” pelo programa a ser executado que para continuar com a execução do programa interpretador é necessária a utilização de um fork().

No link https://www.geeksforgeeks.org/wait-system-call-c/ têm uma imagem que mostra o funcionamento de um processo exec e respectivo fork bastante bem.

```
if ((pid = fork()) < 0) {
    /* fork a child process */
    puts("Erro ao fazer fork");
    exit(1);
} else if (pid == 0) {
    // Executa o processo e termina com o result
    result = execvp(args[0], args);
    // Se chegar a este ponto algo falhou pois nao conseguiu correr o ficheiro args[0]
    puts("Erro ao executar a aplicação");

} else {
    // Processo pai
    do {
        // Ciclo de espera pelo processo filho ...
        int w = wait(&status);
        if (w == -1) {
            printf("Erro na espera do fim do processo %s",args[0]);
            exit(EXIT_FAILURE);
        }
        if (WIFEXITED(status)) {
            printf("Terminou o comando %s com código %d\n",args[0], WEXITSTATUS(status));
        } else if (WIFSIGNALED(status)) {
            printf("killed by signal %d\n", WTERMSIG(status));
        } else if (WIFSTOPPED(status)) {
            printf("stopped by signal %d\n", WSTOPSIG(status));
        } else if (WIFCONTINUED(status)) {
            printf("continued\n");
        }

    } while (!WIFEXITED(status) && !WIFSIGNALED(status));
    // Espera pelo terminos do processo
}
```

Parte 3

file-server / file-client

O funcionamento do protocolo TCP/IP utilizando as funções **accept**, **bind**, **socket**, **setsockopt**, **recv**, **send** em muito é similar ao funcionamento do **write** e **read** de ficheiros mas tendo como “file handle” o relativo ao socket.

O servidor e cliente foram testados em debian e pode ser testado também utilizando os comandos abaixo:

Emulação do funcionamento do cliente

```
echo GET /etc/hosts | nc 127.0.0.1 8080
```

Emulação do funcionamento do servidor

```
nc -l -p 8000
```

Aquando o pedido de um ficheiro ao servidor caso o mesmo não tenha o ficheiro não é enviado nenhum byte ao cliente sendo a conexão fechada.

Durante o desenvolvimento verifiquei que caso seja utilizado um cliente tipo **telnet** o próprio cliente TCP/IP envia bytes dummies no final de uma string pelo que para testar o funcionamento do servidor/cliente devem ser utilizados os protocolos TCP/IP sem adicionar "lixo" no final de cada comando.

O servidor aceita múltiplos clientes em simultâneo conseguindo servir todos tendo utilizando processos child (fork) a cada conexão nova.

Bibliografia / Referências

- http://man7.org/linux/man-pages/man2/read.2.html
- http://man7.org/linux/man-pages/man3/readdir.3.html
- https://www.gnu.org/software/libc/manual/html_node/Testing-File-Type.html
- https://linux.die.net/man/2/stat
- https://www.geeksforgeeks.org/wait-system-call-c/
- https://www.geeksforgeeks.org/socket-programming-cc/