Universidade do Minho
Escola de Engenharia
Mestrado Integrado em Engenharia de Telecomunicações e Informática
Sistemas Operativos

Trabalho Prático

Argus: Controlo e Monitorização de Processos e Comunicação

Bruno Xavier Brás dos Santos A72122

Guimarães
09 de Janeiro de 2020

Resumo

"There is no reason for any individual to have a computer at home"[1]

No âmbito da Unidade Curricular de Sistemas Operativos, do curso de MIETI (Mestrado Integrado em Telecomunicações e Informática), foi atribuído um projeto enquadrado na monitorização de execução e de comunicação entre processos.

Foi disponibilizado um enunciado com as características e as funcionalidades dos programas Cliente e Servidor a desenvolver.

Juntamente com os programas a desenvolver, foi requerido um pequeno relatório de, no máximo, seis páginas. O presente relatório encontra-se estruturado da seguinte forma:

No inicio coloquei um pequeno resumo do presente relatório.

No primeiro capítulo encontra-se, de uma forma resumida, a estrutura do programa Cliente desenvolvido.

O segundo capitulo descreve como foi desenvolvido, e as principais funções, do programa Servidor.

No terceiro e último capítulo, foi colocada a Conclusão.

Para o desenvolvimento do código fonte, utilizei um sistema de controle de versões. O sistema de controle de versões utilizado foi o GitHub, cujo repositório do projeto é o seguinte: "`https://github.com/BrunoXBSantos/Sistemas-Operativos-Argus.git`"

Para a elaboração do presente relatório, optamos por utilizar o software LaTeX. Sendo este software uma excelente forma de produzir trabalhos científicos, relatórios e textos académicos com excelente qualidade.

[1] https://www.theguardian.com/technology/2011/feb/09/ken-olsen-obituary

Capítulo 1

Programa Cliente

O presente capitulo, *Programa Cliente*, tem como finalidade a apresentação do programa responsável pela iteração com o cliente.

Para a comunicação entre o programa cliente e o programa servidor ser bidireccional, implementei dois FIFOs (pipes com nomes). No programa cliente, o FIFO responsável pela escrita no servidor é designado por fd1, e o responsável pela leitura do servidor é fd2.

Na função *int main()*, inicio na linha 29, é verificado o número de argumentos com que o programa cliente iniciou. Se iniciou apenas com um (nome do programa, *./argus*), é inicializada a interface textual interpretada, em que o cliente pode fazer vários pedidos ao servidor sem sair do programa. Para esse fim, é executada a função *iniciaInterpretador()*.

De uma forma resumida, a função mencionada anteriormente, aguarda pela inserção de um comando pelo utilizador. Depois de inserido o comando, colocado na string *fraseLida*, é feito a verificação se a primeira palavra corresponde a uma das funcionalidades suportadas pelo servidor. Se for uma das funcionalidades suportadas, envia, através do FIFO *fd1* para o servidor a frase inserida pelo utilizador.

A titulo de exemplo, se a primeira palavra coincidir com *executar*, é enviado para o servidor a respectiva frase e, de seguida, aguarda uma resposta do servidor a indicar o número de tarefa atribuída, exibindo-a no output do programa Cliente, ficando de seguida a aguardar pela inserção de um novo comando. Num novo comando, se a primeira palavra corresponder a *tempo-inatividade*, é enviado para o servidor a respetiva frase. O procedimento é o mesmo para as restantes funcionalidades suportadas.

Depois de executada uma funcionalidade com sucesso, o programa Cliente aguarda pela nova inserção de um comando. A qualquer momento, se a frase inserida corresponder a *SAIR*, o programa Cliente é encerrado com sucesso, continuando o servidor a correr.

Uma outra forma de o Cliente enviar pedidos ao servidor, é colocando directamente na linha de comandos, para além do nome do programa, o comando a ser executado pelo servidor juntamente com uma flag. Na função *int main()* é verificado o número de argumentos (argc), e como neste caso é diferente de um, é colocado na string *fraseEnviar* o comando inserido. De seguida é executada a função *enviarComando(fraseEnviar)*. Esta função verifica qual é a flag inserida e, envia para o servidor o respetivo comando. A título de exemplo, se a flag é *-r*, responsável pela receção do histórico de tarefas terminadas, é enviado para o servidor o respetivo comando e de seguida executa a função *hitorico()*.

Na função *historico()* é criada uma string auxiliar *temp* e uma variável *flag historico* é inicializada com o valor 0. No ciclo *while*, e enquanto a respetiva variável é igual a zero, o programa Cliente recebe dados do servidor e imprime no seu standart output. No fim da transmissão, a variável supramencionada é colocada com o valor 1 e a função *historico()* termina.

Executada uma funcionalidade, o programa Cliente é terminado com sucesso.

Capítulo 2

Programa Servidor

O presente capitulo, *Programa Servidor*, tem como finalidade a apresentação do programa responsável pelo processamento das tarefas e dos comandos inseridos pelo cliente e enviados para o servidor.

O servidor é responsável por criar os dois FIFOs usados para a comunicação bidireccional. Depois de criar os FIFO's, o descritor do FIFO responsável pela leitura de comandos do Cliente é atribuído a *fd fifo1* e o responsável pela escrita é atribuído a *fd fifo2*.

Para controlar o tempo de execução, tempo de inatividade e a execução eficaz de tarefas, foram declaradas variáveis globais e respetivas flags.

Na função *main()*, no arranque do programa, este verifica se existe alguma informação na extremidade de leitura do FIFO responsável por receber dados do cliente. Quando existe alguma informação proveniente do cliente na extremidade de leitura do FIFO, a informação é lida para a string *comandosRecebidos()*. De seguida, é executada a função *comunicacao()*.

- void comunicacao(char comandosRecebidos[MAX CARACTERES COMANDO])

A função acima destacada, tem como finalidade controlar o tipo de funcionalidade a ser executada pelo servidor.

O argumento recebido corresponde a um vetor de caracteres (string) com o comando inserido pelo cliente. De seguida é verificada qual é a primeira palavra do comando. A primeira palavra corresponde à execução de uma determinada funcionalidade. Depois de retirada a primeira palavra e colocada na variável *primeiraPalavra*, é feito a comparação com as possíveis funcionalidades que podem ser executadas.

Se a primeira-palavra corresponder a *tempo_inatividade* ou a *-i*, é colocada na variável global *tempo_inatividade* o número correspondente à segunda palavra existente no comando. Se a primeira palavra corresponder a *tempo_execucao* ou a *-m*, do mesmo modo, é redefinido o tempo de execução. Se a primeira palavra corresponder a *listar* ou a *-l* é executada a função *listar()*, responsável por enviar, através do FIFO *fd_fifo1*, a lista de tarefas em execução para o cliente. Se a primeira palavra corresponder a *historico* ou a *-r* é enviado para o cliente a lista de tarefas concluídas, através da função *historico()*. Por fim, se a primeira palavra corresponder a *executar* ou a *-e*, e como um dos objetivos do servidor é executar tarefas concorrentemente, é criado um processo novo. O processo novo, filho, é o responsável pela execução da tarefa. Para tal, e apenas o processo filho, executa a função *executarTarefa()*. O processo pai, por outro lado, coloca na lista de tarefas em execução, a respetiva tarefa a ser executada pelo seu filho. De seguida, e no fim de executar qualquer funcionalidade, o processo pai retorna novamente à função *main()* e aguarda a receção de um novo comando.

- void executarTarefa(char *comandosRecebidos, int numeracaoTarefas);

A função *executarTarefa()*, como mencionado anteriormente, é executada apenas pelos processos filho do processo cujo PID é o que executa a função *main()*. A presente função recebe uma string com a tarefa a ser executada e a numeração da respetiva tarefa, atribuída pelo processo Pai.

A tarefa a ser executada é encadeada por pipes anónimos. Depois das declarações das variáveis locais à função, é executada duas funções auxiliares, *separarComandos* e *separarComandos2*, responsáveis por colocar na variável *char *comandos2[MAX-PIPES-COMANDO+1][MAX-PALAVRAS-PIPE]* a lista de comandos a serem executados. A presente variável é um vetor de um vetor de strings e, em cada posição deste vetor, que é um vetor de strings, contem as palavras existentes em cada pipe anónimo.

Para controlar o tempo de execução e o tempo de inatividade, e se não ocorreram erros na execução de cada pipe anónimo, é feito a ativação de sinais através da syscall *signal()*. Os sinais ativados correspondem às flags SIGALRM, SIGCHLD e SIGUSR1, cujos *handle* são, respetivamente, *alarme*, *handle_filhoSecundTerminou* e *handle_sigTerminarTarefa*.

Como é necessário redirecionar o standard output e input na execução dos sucessivos pipes anónimos, foi necessário criar *n_pipes*. O valor de *n_pipes* é calculado através da função *separarComandos2* e correspondem ao numero de pipes, ", que o comando enviado para o servidor tem. De seguida é iniciada a contagem do tempo, através de alarm(1) e iniciada a execução da tarefa.

Sempre que um segundo é passado, o *handle* associao ao sinal *SIGALRM* é executado e é verificado se já se encontra atingido o tempo de execução ou o tempo de inatividade.

A execução da tarefa é realizada através do seguinte "for": *for(i=0;i¡n_pipes+2 && !flag_tempo_execucao && !flag_tempo_ inatividade && !flag_ tarefa_terminada && !flag_erroExecucaoTarefa ;i++)*. Em cada iteração do ciclo "for", é criado um processo filho que executa um determinado comando associado a um pipe. O processo pai, através da ativação do sinal SIGCHLD previamente, espera pela conclusão do processo filho. No *handle* associado a este sinal, e através das macros *WEXITSTATUS* e *WIFEXITED* é verificado se o filho foi corretamente executado. Se não o for, a variável global *flag_erroExecucaoTarefa* é colocada a 1 e a execução da tarefa termina.

O processo responsável pela tarefa executa o ciclo até *n_pipes + 2*. Na ultima iteração do ciclo, o standard input da execução do processo filho anterior é redirecionado para o descritor fd[i-1], com a ajuda da intrução*dup2(fd[i-1][0],0);*, com o objetivo de imprimir o resultado da tarefa no standard output do servidor.

Sempre que o tempo de inatividade ou o tempo de execução é detetado, flag correspondente é ativada, o processo responsável pela execução da tarefa, através do sinal SIGKILL e da syscall *kill*, termina possíveis processos em execução.

De seguida, com a tarefa concluída, o alarme é desactivado, *alarm(0)*, e de acordo com o estado de conclusão da tarefa, esta é inserida no ficheiro de tarefas concluídas e o filho responsável pela tarefa é terminado, *exit(0)*.

Sempre que o processo principal, processo cujo PID inicia a função *main()* cria um filho, o único motivo para que tal acontece, é na execução de uma tarefa, o processo pai guarda numa lista o PID responsável da tarefa, o nome da tarefa e a numeração atribuída. No processo pai também é ativado o sinal SIGCHLD. Assim, quando um filho responsável pela execução de uma tarefa é terminado, o pai recebe uma sinal SIGCLHD e é executada a função *handler handle_filhoPrincipalTerminou()*.

- void handle_filhoPrincipalTerminou(int sig);

Esta função, através da syscall *wait* retira o processo já terminado da tabela de processos, processo zumbi. De seguida, através do valor retornado por *wait* recupera o pid do processo filho responsável pela tarefa e, com a ajuda da função auxiliar *removerTarefaListaExecucao()* remove a tarefa da lista de tarefas em execução. Isto é possível porque, previamente, na lista de tarefas em execução, associado à numeração da tarefa, foi guardado o pid do processo filho responsável pela tarefa.

O programa servidor tem duas listas de tarefas implementado. Uma responsável por guardar as tarefas que encontram-se em execução e outra por guardar o histórico de tarefas já realizadas.

Para as tarefas em execução, foi optado pela estrutura de dados de uma lista ligada. Cada nó da lista armazena três campos de informação, a numeração da tarefa em execução, a string identificativa do comando responsável pela tarefa e o pid do processo filho que a executa.

Por sua vez, sempre que uma tarefa é executada, é guardado num ficheiro as tarefas concluídas.

Decidi utilizar um ficheiro para armazenar as tarefas concluídas, visto que, uma vez que já não se encontram em execução, não necessitava de ter as tarefas concluídas permanentemente em memória.

Uma das funcionalidade requeridas, é terminar uma tarefa em execução. Para eliminar uma tarefa, o cliente envia o número da tarefa a terminar para o servidor. Este, através da lista de tarefas em execução, procura o PID do processo responsável pela execução da tarefa. A função que encontra o PID de uma determinada tarefa é a seguinte:

- int getPidTarefaExecucao(int numeroTarefaTerminar);

De seguida, é enviado o sinal *SIGUSR1* para o processo representativo da tarefa. O processo filho recebe o sinal e executa o *handler* correspondente. O prototipo do *handler* executado é o seguinte:

- void handle_sigTerminarTarefa(int sig);

Esta função coloca o valor da variável global *flag_tarefa_terminada* a 1. Deste modo, o processo filho interrompe a sua execução e envia um sinal SIGKILL aos demais processos que naquele momento estão a executar os comandos encadeados por pipes. O processo filho também é terminado, e de seguido é escrito no ficheiro de tarefas concluídas que a tarefa com a respetiva numeração foi cancelada e, na lista de tarefas em execução, a identificação da tarefa é removida através da função, cujo prototipo é o seguinte:

- int removerTarefaListaExecucao(int pidTarefa);

Uma das funcionalidade requisitadas pelo servidor, é enviar a lista de tarefas já concluídas para o cliente. O protótipo da função responsável por essa funcionalidade é o seguinte:

- void historico();

Esta função, abre o ficheiro em que são guardadas as tarefas apenas para leitura, *O_RDONLY*. De seguida, e com a ajuda da função *int readln(int fildes, char *buf, int nbyte)*, é lida uma linha do ficheiro e colocada na variável *temp*. O conteúdo da variável, e através da syscall *write()*, é enviado para o descritor fd_fifo2, responsável por enviar dados através do FIFO para o cliente. Enquanto não é detetado o EOF no ficheiro, o processo repete-se.

Capítulo 3

Conclusão

Através da realização do presente projeto, foi possível compreender e aplicar os conhecimentos adquiridos nas aulas práticas e teóricas.

O projeto foi realizado, essencialmente, fora das aulas teóricas e práticas. Sendo realizado na parte final do semestre, foi possível aplicar com mais clarividência todos os conceitos abordados ao longo do semestre.

Os objetivos propostos foram, na generalidade, alcançados. Foram concedidos dois programas, Cliente e Servidor, que comunicam através de pipes com nomes, FIFOs. É possível ao cliente enviar tarefas para o servidor executar, concorrentemente, bem como listar as tarefas em execução, listar as tarefas concluídas, terminar uma dada tarefa em execução e, definir vários parâmetros, como por exemplo, o tempo máximo de execução e o tempo de inatividade entre pipes anónimos.

A parte em que senti mais dificuldade, foi na elaboração do tempo de inatividade. No início o conceito referido não foi totalmente compreendido. Depois de alguns testes e procura de informação, o referido ponto foi feito com sucesso.

Em conjunto com o presente relatório, numa pasta separada, é enviado o código fonte em linguagem C dos programas Clientes e servidores desenvolvidos.