UNIVERSIDADE DO MINHO

DEPARTAMENTO DE INFORMÁTICA

SISTEMAS OPERATIVOS

Trabalho Prático

Controlo e Monitorização de Processos e Comunicação

Grupo 6

Autores:

Diogo Barros (A89555)

Ivo Vilas Boas (A89492)

Joel Gama (A82202)

15 de Junho de 2020

Conteúdo

Introdução

O presente relatório surge no âmbito da Unidade Curricular de Sistemas Operativos integrada no 2º ano do Mestrado integrado em Engenharia Informática.

Este trabalho representa a avaliação da componente prática da UC. Neste trabalho é pedido que seja implementado um programa de "Controlo e Monitorização de Processos e Comunicação".

O trabalho é composto por três partes. Funcionalidades mínima, funcionalidade adicional e cliente e servidor. Nas funcionalidades mínimas é esperado que sejam implementadas: tempo de execução, tempo de inatividade, executar tarefa, listar tarefas em execução, terminar uma dada tarefa em execução, histórico de tarefas terminadas e por fim ajuda para o utilizador.

Na funcionalidade adicional era esperado que fosse possível verificar o output de uma dada tarefa.

Por fim, o cliente e servidor é a forma de comunicação do programa.

Arquitetura Geral

Como forma de início do trabalho, o grupo decidiu basear a implementação numa comunicação através de *Pipes* com nome. Os principais participantes da comunicação são os $argus$ e $argusd$ (Cliente e Servidor, respetivamente).

A comunicação é feita da seguinte forma, o $argus$ é responsável pela receção dos comandos, que logo de seguida são enviados ao servidor através do *Pipe* "Cliente", então o $argusd$ lê a informação recebida e procede à tomada de decisão. Posteriormente o $argusd$ envia informações sobre o comando executado ao cliente através do *Pipe* "Bus"e por fim o $argus$ vai imprimir essas informações para o terminal do cliente.

O objetivo da aplicação é fornecer ao utilizador um ambiente em que várias tarefas possam ser submetidas de modo a que, caso sejam um pouco mais demoradas, possam executar em paralelo. Desta forma existe controlo perante a execução das tarefas para não serem excedidos certos limites de tempo. A aplicação permite também obter informações sobre todas as tarefas que executou, em especial o modo como terminaram. Permite saber quais as tarefas em execução e caso deseje, terminar alguma que encontre em execução fornecendo o seu respetivo *id*.

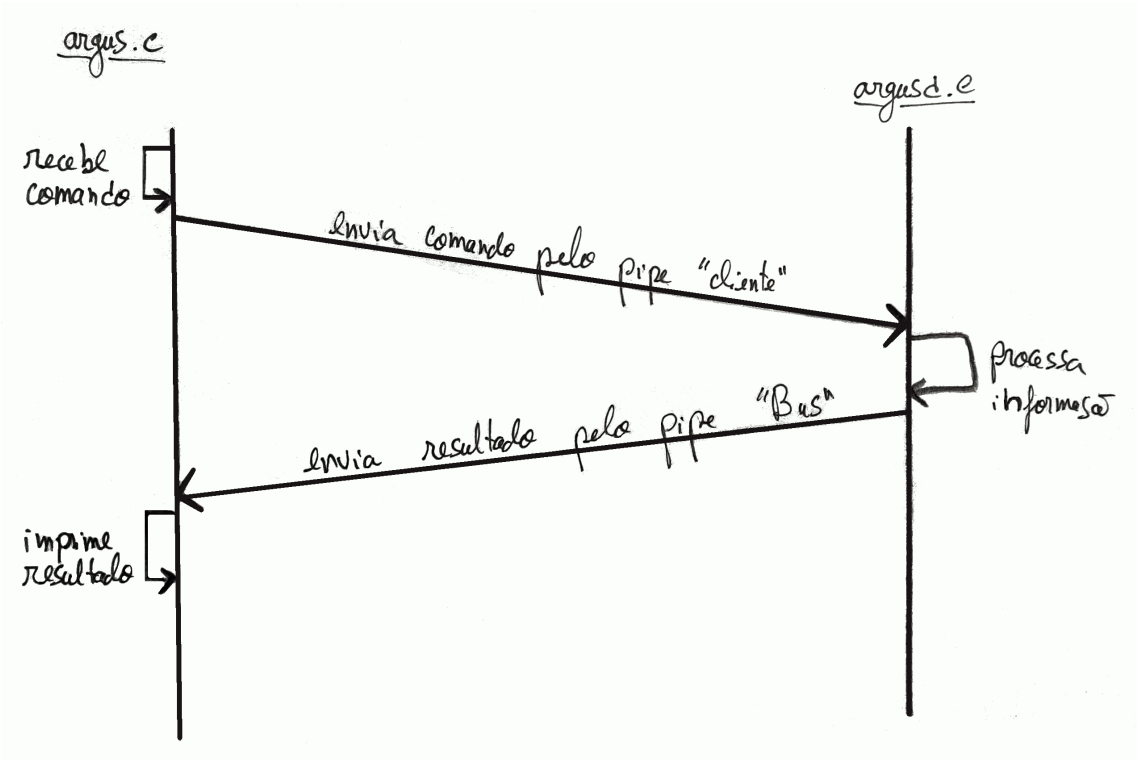

Figura 2.1: Esquema da arquitetura de comunicação.

Implementações

Neste capítulo será apresentado a forma como o grupo abordou as funcionalidades mínimas pedidas no enunciado.

3.1 Executar tarefa

Um dos requisitos do enunciado era a execução de tarefas. Tendo isso em conta o grupo desenvolveu a função:

$$int\ executar(char * command, int\ tempo\_execucao, int\ indice\_tarefa)$$

Esta retorna um inteiro apenas como verificação e recebe quatro argumentos, uma *string* que representa o comando a executar, e três inteiros que representam, respetivamente, o tempo de execução máximo e o índice da tarefa que se vai executar.

Para o funcionamento da função em si são utilizados *pipes* anónimos e uma *string* com os comandos a executar. Uma vez que a string de comandos é recebida sem separação de cada comando o algoritmo inicia-se com a separação dos comandos pelos *pipes* ("|") presentes no comando. Neste processo também é feita a contagem dos comandos, valor que é utilizado para criar o número correto de *pipes* anónimos necessários. O passo seguinte é criar e inicializar os *pipes* anónimos referidos anteriormente. Tendo já os *pipes* anónimos criados e os comandos separados é possível iniciar o processo de execução. Através de um ciclo *while*, enquanto a lista de comandos não for vazia, é criado um processo filho (através da syscall $fork()$) para executar o respetivo comando. Em cada iteração do ciclo são feitas as ligações dos *inputs* e *outputs* de cada execução através dos *pipes* anónimos criados anteriormente. Desta forma é possível fazer os redirecionamentos necessários para produzir o resultado esperado.

Porém, existem duas particularidades no ciclo definido. Em cada iteração são feitas duas verificações. Uma para verificar se o comando em questão não é o último e outra para verificar se o comando não é o primeiro. No primeiro ponto, a verificação é feita pois caso seja o último comando não queremos redirecionar o *output* da execução para o próximo *pipe*, mas sim para o ficheiro *output.txt* (redirecionamento do *stdout*). E no segundo caso a verificação é feita pois no primeiro comando não queremos utilizar o valor que se encontra no *pipe* anterior pois esse ainda nem existe.

Por fim, no final do ciclo *while* são fechados todos os *pipes* e é feito *wait* para todos os processos filhos como forma de evitar processos zombies.

Em seguida apresentamos o pseudo código da funcionalidade de redirecionamento presente no ciclo da função $executar$ para uma melhor visualização do mesmo.

```
int fd[2*count];

cmds = split(command,"|");

while(cmds[index] != NULL) {
```

```
        if((pid = fork()) == 0) {

            // Não primeiro
            if(cmds[next] != NULL)
                dup2(fd[j + 1], 1)

            // Não último
            if(index != 0)
                dup2(fd[j-2],0)

            for(i = 0; i < 2 * count; i++)
                close(fd[i]);

            argv = split(cmds[index]," ");
            execvp(argv[0], argv);
        }
        next++;
        index++;
        j+=2;
    }
```

Por fim, uma explicação da arquitetura utilizada para o redirecionamento dos resultados entre os diferentes comandos.

No esquema seguinte é possível ver a arquitetura adotada pelo grupo para o redirecionamento dos resultados entre os diferentes comandos utilizando *pipes* anónimos. O processo inicia-se com um comando sendo o resultado deste enviado para o primeiro *pipe*. O segundo comando utiliza o *output* que foi escrito pelo primeiro comando no *pipe* como *input* e escreve o resultado no *pipe* seguinte. E assim sucessivamente até ao último comando, que escreve o resultado para o *stdout*.

```
cmd0      cmd1      cmd2      cmd3      ...     cmd_n-1
    pipe0     pipe1     pipe2     ... pipe_n-1
    [0,1]     [2,3]     [4,5]     ... [2n-2,2n-1]
```

(Assumindo que existem n comandos e *pipes* e $2n$ índices.)

Como nota final sobre a função $executar$. Nesta função são tratados mais pontos referentes aos requisitos do enunciado, mas serão explicados nos próximos pontos.

3.2 Terminar tarefa

Neste requisito é pretendido que esteja disponível uma forma para o cliente terminar uma tarefa dando o seu *id*.

Para a implementação, o grupo utiliza um ficheiro com os *pids* das tarefas associados ao *id* da tarefa. Dessa forma é possível matar os processos associados e utilizando o *id* é possível alterar o estado da execução no ficheiro de *logs*.

A forma de funcionamento do programa é bastante sistemática. O programa verifica se o estado atual das tarefas associadas ao *id* é zero (em execução), se sim o programa mata todos os procesos juntamente com o processo pai destes (nesta ordem), se o valor não for zero sai da função $terminar$ uma vez que a tarefa não se encontra em execução.

3.3 Tempo de Execução e Tempo de Inatividade

Dois dos requisitos presentes no enunciado eram o estabelecimento de um limite de tempo para a execução e de inatividade associados às tarefas. O primeiro diz respeito ao tempo limite para a execução de uma tarefa, enquanto o segundo representa o tempo que um *pipe* anónimo associado a uma tarefa pode ficar inativo (à espera de *input*, por exemplo).

Para solucionar o problema acima apresentado o grupo optou por guardar os valores introduzidos pelo $argus$ em variáveis globais do servidor. Estas variáveis são depois passadas como argumento à função $executar$ que faz o processo de verificação do tempo de execução de uma tarefa, pois é nela que as tarefas são executadas.

Em relação do tempo de execução a abordagem do grupo foi a seguinte:

No início da função $executar$ é feito um $signal(SIGALRM, timeout\_handler)$ que altera o comportamento do *SIGALRM*. Este faz com que todos os processos relativos à chamada do comando $executar$ terminem e altera o estado da execução da tarefa no ficheiro de *logs*. Por fim, depois de alterar o comportamento do *SIGALRM* é feito um $alarm(tempo\_execucao)$ caso o tempo de execução passado como argumento no $executar$ seja superior a zero.

Infelizmente, o grupo não conseguiu fazer a sua implementação do $tempo - inatividade$. Porém encontramos casos em que seria necessário ter a opção funcional, como é o caso da tarefa $'cat'$ em que ficava à espera de *input*.

3.4 Histórico e Listar

No enunciado era também pedido que fosse disponibilizada a possibilidade de visualização da informação relativa às tarefas que foram iniciadas pelo *cliente*. Porém é necessário que sejam separadas pelas que ainda se encontram em execução e as que já foram terminadas.

Para a implementação da funcionalidade a função definida foi a $listarTarefas$. São passados como argumento o valor do índice da próxima tarefa, o *pipe bus* e um inteiro para definir o tipo de listagem que o cliente pretende. O valor do índice da próxima tarefa representa o número de linhas no ficheiro de *logs*, o *pipe bus* é utilizado para comunicação com o cliente e por fim o inteiro passado define se a função deve funcionar como "listar"ou como "histórico".

Como a estrutura do ficheiro de *logs* é previamente conhecida possibilita pegar no valor do *id*, estado da tarefa e o comando executado. Assim, é possível proceder à comparação do estado para verificar se este deve, ou não, ser listado segundo a opção pretendida.

A seguir apresentamos os valores escolhidos pelo grupo para fazer a associação de estados.

Estado	Valor associado
Em execução	0
Terminado normalmente	1
Terminado por tempo de execução	2
Terminado por tempo de inatividade	3
Terminado pelo utilizador	4

Tabela 3.1: Valores associados aos diferentes estados.

3.5 Ajuda

O último requisito do enunciado implementado pelo grupo foi a opção de ajuda. Com ela o *cliente* tem todos os comandos disponíveis assim como a sua respetiva descrição.

Posto isto, o grupo optou por uma solução bastante simples, que se baseia num ficheiro de texto (*help.txt*) com a informação pretendida e a cada vez que o comando $ajuda$ fosse executado, o *servidor* chama a função $printHelp()$, que lê o ficheiro anteriormente referido e apresenta o seu conteúdo ao *cliente*.

```
tempo-inactividade segs (Configura o tempo de inatividade para segs segundos)
tempo-execucao segs     (Configura o tempo de execução para no máximo segs segundos)
executar 'tarefa'       (Executa a tarefa escrita em "tarefa". Por exemplo, 'p1 | p2 | p3')
termina n               (Termina a tarefa com o id n)
historico               (Imprime o histórico de tarefas que foram dadas como terminadas)
listar                  (Lista as tarefas em execução)
ajuda                   (Imprime os comandos suportados)
```

Figura 3.1: Conteúdo do ficheiro *help.txt*.

Funcionalidade Extra

Em relação à funcionalidade adicional referida no enunciado, inicialmente o grupo optou por utilizar o ficheiro *output.txt* que tem como propósito coletar todo o *standard output* do $argusd.c$ e um outro ficheiro auxiliar *tams.txt*. Porém o facto de se ter de utilizar um ficheiro com a extensão *".idx"* passou despercebido e paramos a implementação.

Após uma longa pesquisa sobre esse tipo de ficheiros, começamos a entender mais sobre o quão prático seria a utilização dos ficheiros *.idx* para solucionar o problema. Mas não foi possível entender o seu modo de funcionamento em tempo útil, e então voltou-se à ideia original, apesar de talvez pouco eficiente, adotando a seguinte estratégia:

Após cada execução de uma tarefa o tamanho do ficheiro *output.txt* iria ser guardado no *tams.txt*. No final da sua execução, esse ficheiro guarda os tamanhos da tarefa sequencialmente e executando o comando *output* que vai buscar o tamanho registado ignorando os valores anteriores. Com a exceção do penúltimo valor, os outros valores eram depois usados para calcular a posição a partir da qual se iria ler no ficheiro *output.txt*, e subtraindo ambos os valores, teríamos o tamanho necessário para a leitura.

Apesar de ter esta ideia estruturada, o grupo apercebeu-se de um problema. Se um programa de índice mais baixo tivesse uma execução mais longa, o tamanho do ficheiro após a sua escrita ia ficar associado a outra tarefa, poder-se-ia optar por associar o índice da tarefa ao respetivo tamanho, mas isso não solucionaria o problema. Confrontados por este problema, foi possível entender o porquê da sugestão de utilizar os ficheiros *.idx*.

Testes de funcionalidades

Os testes do programa consistiram em executar os comandos que o grupo pretendia avaliar, tentar reproduzir em tempo real as condições que pudessem gerar problemas e verificar se o programa estava apto para superar tais situações.

A mesma estratégia foi utilizada para verificar se o programa estava a interpretar os comandos fornecidos corretamente.

No caso em que era necessário observar se o programa escrevia o *output* esperado nos ficheiros, o grupo deixava um editor de texto com os ficheiros abertos para ver em tempo real as alterações que eram feitas nos mesmos.

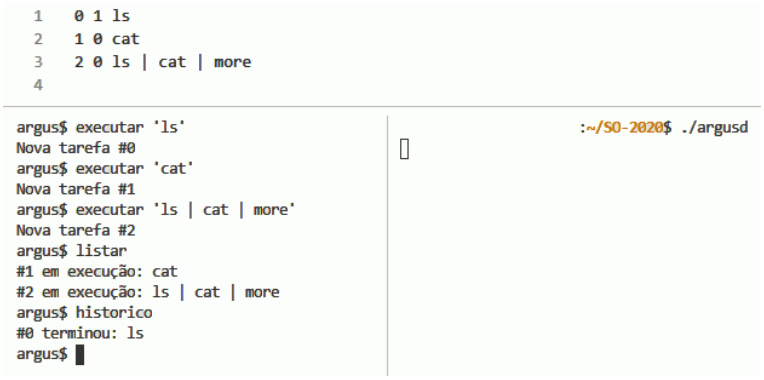

Figura 5.1: Exemplo da utilização do comando $executar$, $listar$ e $historico$.

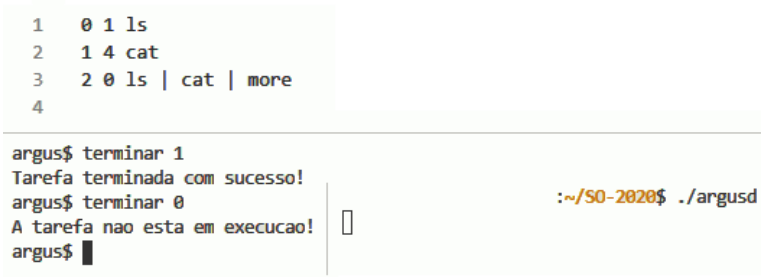

Figura 5.2: Exemplo da utilização do comando $terminar$.

Figura 5.3: Exemplo da utilização do comando $tempoexecucao$.

Conclusão

Este trabalho foi importante para aplicar a matéria dada nas aulas ao longo deste semestre.

Foi possível aplicar conceitos relacionados com todos os conteúdos abordados pelos docentes. Desde as criações, leituras e escritas em ficheiro, até a utilização de *pipes* (quer anónimos, quer com nome) e sinais.

Analisando o trabalho realizado, apesar de todo o trabalho do grupo, não foi possível implementar dois pontos pedidos no enunciado, o tempo de inatividade e a funcionalidade adicional. No ponto do tempo de inatividade o grupo teve bastante dificuldade em entender a forma de medir a inatividade de um processo, isso levou à não conclusão do requisito. Já no requisito adicional, apesar de o grupo saber como fazer as implementações, por falta de tempo para melhor compreensão da extensão de ficheiros *.idx*, não foi possível realiza-la.

Por fim, O grupo ainda reparou que existem processos "defuntos". Isto é, o processo pai termina antes que os processos filhos terminarem. O problema é causado pelo facto processo pai não esperar que os processos filho terminem, pois o grupo pretendia oferecer ao cliente a possibilidade de realizar tarefas em paralelo. Como a conclusão do processo pai se dá no ficheiro $argisd$ os processos fica listados como *argusd (defunt)*.