



UNIVERSIDADE DO MINHO
DEPARTAMENTO DE INFORMÁTICA
SISTEMAS OPERATIVOS

Trabalho Prático

Controlo e Monitorização de Processos e Comunicação

Grupo 12

Autores:

Henrique Gabriel dos Santos Neto (A89618)



Sara Alexandra Gomes Marques (A89477)



Tiago Pinheiro (A82491)



30 de Abril de 2022

Conteúdo

1	Introdução	2
2	Código	3
2.1	Cliente - argus.c	3
2.2	Server - argusd.c	3
2.3	<i>Interface</i>	3
2.3.1	<i>ArgusStatus</i>	3
2.3.2	<i>TaskInfo</i>	3
2.3.3	<i>argusINT</i>	4
2.3.4	<i>argusKillAllTasks</i>	4
2.3.5	<i>taskCleaner</i>	4
2.3.6	Funções das funcionalidades	4
2.3.7	<i>argusRTE</i>	5
2.3.8	<i>argusRTE init</i>	5
2.3.9	<i>argusRTE kill</i>	5
2.3.10	<i>argusRTE readcomand</i>	5
2.3.11	<i>argusRTE run</i>	5
2.4	<i>taskexec.c</i>	5
2.4.1	<i>KillGroup</i>	5
2.4.2	<i>idlelimit</i>	6
2.4.3	<i>execSystem</i>	6
2.4.4	<i>task</i>	6
2.5	<i>Log Manager</i>	6
2.5.1	<i>Log IDX</i>	6
2.5.2	<i>Log</i>	7
2.6	Auxiliares	8
2.6.1	<i>list.c</i>	8
2.6.2	<i>constants.c</i>	8
3	Conclusão	9

Introdução

O presente relatório surge no âmbito da Unidade Curricular de Sistema Operativos integrada no 2º ano do Curso de Engenharia Informática.

Neste trabalho é implementado um serviço de monitorização de execução e de comunicação entre processos. Essencialmente, o projeto vai estar dividido em duas partes: cliente e servidor.

O cliente deve ter uma interface com o utilizador via linha de comandos permitindo suportar funcionalidades como executar comandos, definir tempos máximos para a execução destes, visualizar que o histórico de comandos executados, entre outros. Por outro lado, o cliente também deve oferecer uma *shell*, que irá ter uma funcionamento muito idêntico à linha de comandos.

O servidor deverá manter em memória a informação relevante para suportar o cliente e as funcionalidades que este permite.

Código

2.1 Cliente - argus.c

Este ficheiro contém o código que suporta a funcionalidade do cliente. O ficheiro é composto por duas funções: a função `main` para inicializar o programa e a função `shell` que vai ler, processar e comunicar ao servidor um comando, proveniente de uma shell, a ser executado.

2.2 Server - argusd.c

Este ficheiro contém o código que suporta a funcionalidade do servidor. Assim, ao executar o programa `./argusd`, o servidor é aberto e são criados dois *pipes* com nome, `ArgusInpunt` e `ArgusOutput`, que correspondem respetivamente ao canal de entrada e ao canal de saída de informação do servidor. Caso se der um *EOF* nos *FIFOs*, o servidor simplesmente procede a criação de novos *pipes*.

2.3 Interface

O ficheiro `interface.c` possuí o código que realiza boa parte do trabalho do programa.

2.3.1 ArgusStatus

A `ArgusStatus` é a estrutura principal do programa. Esta *struct* é o estado permanente do programa que é inicializada logo no inicio do programa. Sendo que, para este efeito, é criada, neste ficheiro, uma *struct ArgusStatus* como variável global, chamada `msystem`.

Esta *struct* é composta por uma lista de `TaskInfo`, que iremos abordar no capítulo seguinte. Para além disso, a estrutura possuí um inteiro que tem mantém a contagem do número das *taks* (ativas, acabadas ou abortadas). Nesta struct também são definidos os tempos máximos de execução e de inactividade (ambos *integer*) e ela mantém os inteiros resultantes da abertura dos ficheiros *log*. As funções `initArgusStatus` e `freeArgusStatus` iniciam e libertam uma *struct ArgusStatus*, respetivamente.

2.3.2 TaskInfo

A `TaskInfo` é uma estrutura que representa a informação de cada tarefa executada, isto é, cada vez que há um pedido para executar, uma *struct* destas é criada e, caso a execução seja iniciada, ela é inserida na lista de tarefas da `ArgusStatus`, como foi dito anteriormente.

A *struct* é composta por duas *strings*: o comando a ser executado e o nome do ficheiro para onde o *output* irá ser escrito, temporariamente. Para além disso, a *struct* também é composta por 3 inteiros: o número do processo (*pid*), o *index* que o comando irá ter e ainda a variável *output*, que irá ser o resultado da função *open* ao abrir o ficheiro de *output*. Ainda relacionado com esta *struct* existem duas funções: `mkTaskInfo` e `TaskInfo_free`; A primeira recebe um comando, um *pid* e um *index* e cria uma `TaskInfo`. Neste processo de criação, caso não seja possível abrir o ficheiro de *output*, é indicado que o *output* seja o *standard output*. A `TaskInfo_free` liberta uma *struct TaskInfo* da memória.

2.3.3 argusINT

Esta função força a terminação de todas as tarefas do *argus* e dele mesmo. Para isso, ela chama a função `argusKillAllTasks`, para matar as tarefas, e depois envia um *signal kill* para terminar o *argus* (matar o processo do *argus*).

2.3.4 argusKillAllTasks

Esta função, como foi dito anteriormente, serve para terminar todas as tarefas do *argus*. Para isso, ela primeiro verifica se realmente existem tarefas a correr, caso não haja, não faz nada. Caso existam, ela percorre a lista que contém as tarefas e enviam um *signal kill* para o respetivo processo.

2.3.5 taskCleaner

A função `taskCleaner` é responsável por eliminar as tarefas inválidas ou que já terminaram do programa, isto é, remover as tarefas da estrutura. Caso as tarefas tenha acabado são feitos *updates* nas respetivas entradas dos ficheiros de *log*.

2.3.6 Funções das funcionalidades

- `setMaximumRunTime` - recebe um *integer* e atualiza o tempo máximo de execução que um processo pode ter. Isto é, sempre que é alterado aplica-se a todos os processos até ser alterado novamente.
- `setMaximumIdleTime` - a função `setMaximumIdleTime` tem um funcionamento muito semelhante ao `setMaximumRunTime`, mas em vez de alterar o tempo máximo de execução altera o tempo máximo de inatividade dos processos.
- `execute` - é a função que executa um comando. Ela recebe o comando que é suposto executar, cria uma `TaskInfo` e um processo filho. Caso o processo filho seja criado com sucesso ela passa o trabalho à função `task` do `taskexec.c`. Para além disso, ainda atualiza a `ArgusStatus msystem` com as novas informações. Por último, é enviado para o ecrã uma mensagem com o *index* da tarefa.
- `listTasks` - imprime as tarefas em execução. Neste caso, vai ser impresso o *index* () e o comando de cada elemento da lista que está guardada na `msystem`, note que, cada elemento da lista será uma *struct TaskInfo*.

- `terminate` - recebe o *index* de uma tarefa e termina essa tarefa. Para isso, a função vai procurar na lista a tarefa com o respetivo *index* e, caso encontre, vai utilizar a função *kill* para matar o processo (uma vez que o *pid* está na *struct*).
- `help` - imprime para o ecrã a lista de comandos existentes.
- `history` - verifica que tarefas já foram terminadas, vai buscar o seu *output* ao ficheiro de *logs* e imprime-as para o ecrã por ordem crescente. Caso não existam tarefas terminadas, imprime uma mensagem a dizer: "Nenhuma tarefa em historico".
- `output` - imprime o *output* de uma dada tarefa. A função retorna 0 se tiver sucesso, -1 em caso de insucesso e -2 se a tarefa ainda estiver em execução.

2.3.7 argusRTE

É o *run time environment* do programa *argus*.

2.3.8 argusRTE init

Inicializa o estado do ambiente de execução do programa.

2.3.9 argusRTE kill

Esta função mata todas as tarefas do programa e liberta a *struct msystem*.

2.3.10 argusRTE readcommand

Apenas lê um comando do descritor de ficheiro dado.

2.3.11 argusRTE run

Função responsável pela execução de um comando, apenas verificar qual é o comando a executar e passa o trabalho a outra função.

2.4 taskexec.c

O `taskexec.c` é responsável por executar as tarefas impostas à interface. Desta forma, dado um comando, o `taskExec` procederá a montar toda a infraestrutura, quer sejam *pipes*, quer seja o processamento de sinais, entre outros. Isto para computar as várias operações pedidas.

Neste ficheiro é definida uma lista ligada que vai conter *ids* de processos.

2.4.1 KillGroup

Envia uma sinal (*SIGKILL*) ao grupo do processo, matando todos os seus descendentes.

2.4.2 *idlelimit*

Esta função despeja o conteúdo de **STDINT** para **STDOUT**, se ao fim de um dado tempo não houver transferência de dados envia um sinal de alarme ao programa principal. O tempo limite é passado como parâmetro.

2.4.3 *execSystem*

A função recebe uma *string* **comando** e, através do **execvp**, executa esse comando.

2.4.4 *task*

A função recebe uma *string*, **command**, e separa esse comando por operações a executar (as várias operações a serem executadas estão separadas por *pipes* - barras). A função depois tem dois ciclos, sendo um deles um falso ciclo, uma vez que, se o tempo de inactividade for 1, ele itera duas vezes, caso contrário apenas uma. Se o programa não estiver na primeira execução do círculo (onde executa a tarefa), o processo vai ficar a medir a variação de fluxo do *pipe*, desta forma, se a variação não satisfizer o tempo limite de inactividade é enviado um *SIGTERM* ao processo pai.

2.5 *Log Manager*

Este ficheiro é responsável pelas funções que cuidam dos ficheiros de *log* do programa, tanto o ficheiro de *logs* normal como o ficheiro do tipo **idx**.

2.5.1 *Log IDX*

O ficheiro **idx** é composto por um "número mágico" que identifica o tipo de dados nele guardado, seguido do número total de linhas escritas no ficheiro. Depois destas informações, estão registados os valores de todos os índices associados aos dados guardados no ficheiro logs.

openIDX

A *openIDX* abre um ficheiro do tipo *idx* ou, caso o ficheiro não exista ou seja inválido, cria um.

A função vai receber um parâmetro: um apontador para um inteiro (**nLinhas**) que, caso o ficheiro seja aberto com sucesso, irá ser atualizado. Este número representa o número de linhas do ficheiro, logo, se a função criar um novo ficheiro, o número de linhas será igual a zero.

O valor retornado pela *openIDX* é o descritor do ficheiro *idx*, caso obtenha sucesso, ou *-1*, no caso de ter ocorrido algum erro.

updateIDX

A função *updateIDX* atualiza uma entrada no ficheiro *idx*. A função recebe três parâmetros: o descritor do ficheiro *idx*, o *index* a atualizar e o valor a colocar na entrada.

Caso o *index* seja um valor negativo ou superior ao último *index*, será adicionada uma nova entrada no fim do ficheiro com o respetivo **valor**, e o número de linhas guardado no ficheiro *idx* será atualizado.

Por outro lado, caso o *index* esteja compreendido entre os os valores anteriormente mencionados, a entrada apenas será atualizada com o novo **valor**.

readIndexIDX

A função `readIndexIDX` lê um dado índice de um ficheiro *idx*. Nesta função, se o *index* for inválido ou caso não seja possível fazer a leitura do ficheiro no determinado *index*, é retornado o valor `-1`. Por sua vez, se a operação for realizada com sucesso, é retornado o valor encontrado no *index* dado.

2.5.2 Log

O ficheiro *logs* é inicializado com um byte 0, possuindo depois várias entradas com informações relativas aos comandos executados pelo programa - nomeadamente, o seu tempo de inatividade e de execução, o nome do comando executado, a data e hora em que foi executado o comando e, o seu *output*.

openLogs

A `openLogs` tem um funcionamento relativamente semelhante à `openIDX`, ou seja, a função ou abre um ficheiro de *logs*, ou cria um novo caso não exista. O valor de retorno será ou o descritor do ficheiro ou `-1` caso tenha ocorrido um erro.

updateLogs

A `updateLogs` adiciona uma entrada ao ficheiro de *logs* a partir dos parâmetros dados.
Esta função recebe cinco parâmetros:

- *logfile* - ficheiro de *logs* onde irá ser adicionada uma entrada.
- *comando* - o comando que foi executado.
- *inatividade* - o tempo máximo de inatividade.
- *execucao* - o tempo máximo de execução.
- *filetocopy* - descritor do ficheiro onde está o *output* resultante da execução do comando.

Com este parâmetros, a função vai abrir o ficheiro de *logs* e escrever no final do mesmo os parâmetros recebidos. No caso do ficheiro de *output*, o seu conteúdo será lido e copiado para o ficheiro de *logs*. A função irá também registar a data e hora atual no ficheiro *logs*.

writeOutputTo

A `writeOutputTo` lê um ficheiro de *logs* partindo de um dado *index*, e copia o nome do comando aí registado, juntamente com o seu *output* (que inclui a data de execução associada) para um certo ficheiro, cujo descritor é dado como parâmetro.

getCommandInfo

A `getCommandInfo` funciona de modo semelhante à `writeOutputTo`, mas à exceção do descriptor do ficheiro de *logs* e do *index*, recebe parâmetros diferentes. Nomeadamente, um array *output-command*, o seu *size* e dois apontadores: *inatividade* e *execução*.

A função tem então como objetivo ler o ficheiro de *logs* a partir do *index*, e guardar no array e nos apontadores respetivos o nome do comando e os tempos de execução e de inatividade aí registados.

2.6 Auxiliares

2.6.1 list.c

Este ficheiro auxiliar contém as funções necessárias para gerir uma lista ligada. A função `List_alloc` serve para alocar um espaço na lista. Enquanto que as funções `List_prepend` e `List_append` adicionam um dado elemento ao inicio e ao fim de uma lista dada, respetivamente. A lista tem duas funções para libertar memória: a `List_free` e a `List_lfree`. A única diferença entre elas é que a primeira recebe uma função para libertar a data e a outra não (ambas libertam a lista toda). A `list_tail` remove o primeiro elemento da lista e devolve a cauda. Por fim, existe uma função simples chamada `List_length` que dada uma lista determina o seu tamanho.

2.6.2 constants.c

Este ficheiro apenas serve para definir algumas variáveis globais que irão ser necessárias noutras partes do programa. No ficheiro estão definidas 5 *strings* e dois *unsigned ints*. Sobre as *strings* não há muito a dizer porque apenas são nomes para os *FIFOS* ou ficheiros, no caso dos *ints*, o `ReadBufferSize` tem o valor de 4096, uma vez que é o tamanho mínimo que se encontra nas páginas de memória dos sistemas operativos atuais. Por outro lado, o `MaxLineSize` é 1024, sendo que, não existe nenhuma razão específica por detrás desta escolha.

Conclusão

Em suma, este trabalho contribuiu para a nossa melhor compreensão e maior apreciação das utilidades práticas dos vários temas abordados ao longo do semestre na cadeira de Sistemas Operativos, nomeadamente no que respeita à comunicação entre *pipes* (principalmente entre *pipes* com nome, *FIFOs*).

No entanto, apesar das dificuldades encontradas ao longo do desenvolvimento do trabalho, consideramos que foram dadas respostas adequadas a todos os desafios que nos foram propostos, e estamos satisfeitos com o resultado final do projeto.