

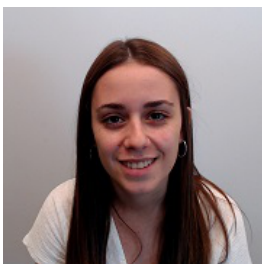
UNIVERSIDADE DO MINHO
MESTRADO INTEGRADO EM ENGENHARIA INFORMÁTICA



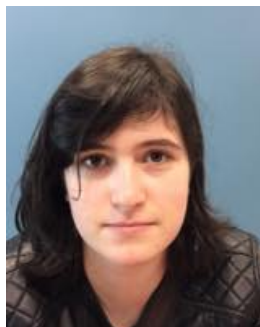
Relatório - Sistemas Operativos

GRUPO 88

Trabalho realizado por:



A75119
Adriana Gonçalves



A79987
Ana Rita Guimarães



A75480
Marco Gonçalves

Braga, 7 de Junho de 2020

Conteúdo

1	Introdução	2
2	Funcionalidades	2
2.1	Limites de tempo	2
2.1.1	Inactividade	2
2.1.2	Execução	2
2.2	Tarefas	2
2.2.1	Executar	3
2.2.2	Listar	3
2.2.3	Terminar	3
2.2.4	Histórico	3
2.3	Consulta de Outputs	3
2.4	Ajuda	4
3	Estrutura do trabalho	5
3.1	Common	5
3.1.1	Macros	6
3.1.2	Funções	6
3.2	Cliente	6
3.3	Servidor	7
3.3.1	Macros	7
3.3.2	Variáveis globais	7
3.3.3	Ficheiros	8
3.3.4	Modo de Funcionamento	8
3.4	Makefile	9
4	Conclusão	9

1 Introdução

Neste projeto foi-nos solicitado pelos docentes da unidade curricular de Sistemas Operativos que tem como principal objetivo a realização de um serviço de monitorização, de execução e de comunicação entre processos.

Assim sendo, foi necessária a criação de uma série de ficheiros. Três ficheiros para as variáveis globais (`current_task_id`, `max_execution_time`, `max_inactivity_time`), dois ficheiros para tarefas em execução (`task_info_ID`, `task_log_ID`) e finalmente dois ficheiros de log (`log`, `log.idx`).

2 Funcionalidades

Para uma correta monitorização das tarefas executadas pelo servidor por parte do cliente, definimos algumas funcionalidades para que seja possível listar as tarefas em execução, o tempo máximo de execução de uma tarefa, listar o histórico das tarefas terminadas, entre outras. Assim, neste capítulo iremos abordar todas as funcionalidades presentes no sistema e como utiliza-las.

2.1 Limites de tempo

2.1.1 Inatividade

Definir o tempo máximo que um processo passa sem escrever nada no pipe que por default está definido como 5 segundos.

```
argus$ tempo-inactividade 10
```

ou

```
./argus -i 10
```

2.1.2 Execução

Definir o tempo máximo que uma tarefa pode demorar a executar que por default está definido como 5 segundos.

```
argus$ tempo-execucao 10
```

ou

```
./argus -m 10
```

2.2 Tarefas

As tarefas podem ser executadas pelos cliente (`argus`) e caso seja pedido para executar mais do que uma estas serão executadas de forma concorrente pelo servidor.

2.2.1 Executar

Para o cliente mandar executar um comando ao servidor este pode ser feito da seguinte forma.

```
argus$ executar " ls | sort "
```

ou

```
./argus -e " ls | sort "
```

2.2.2 Listar

É possível listar os comandos em execução.

```
argus$ listar
```

ou

```
./argus -l
```

2.2.3 Terminar

Caso queira terminar uma tarefa imediatamente terá que especificar o identificador da tarefa que pretende terminar. (Assumindo que o identificador do comando a terminar é 1)

```
argus$ terminar 1
```

ou

```
./argus -t 1
```

2.2.4 Histórico

Ver o histórico de comandos já executados.

```
argus$ historico
```

ou

```
./argus -r
```

2.3 Consulta de Outputs

Depois de saber o identificador do comando é possível ver qual o seu output depois deste ter terminado. (Assumindo que o identificador do comando a consultar o output é 1)

```
argus$ output 1
```

ou

```
./argus -o 1
```

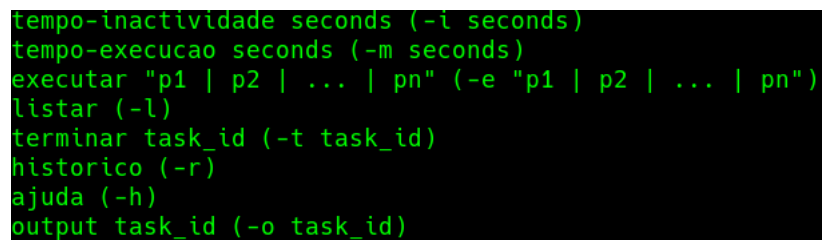
2.4 Ajuda

Caso não saiba como utilizar o programa este disponibiliza-se para lhe facilitar a vida com o comando ajuda

argus\$ ajuda

ou

./argus -h



```
tempo-inactividade seconds (-i seconds)
tempo-execucao seconds (-m seconds)
executar "p1 | p2 | ... | pn" (-e "p1 | p2 | ... | pn")
listar (-l)
terminar task_id (-t task_id)
historico (-r)
ajuda (-h)
output task_id (-o task_id)
```

Figura 2: Output do comando ajuda

3 Estrutura do trabalho

Para uma melhor compreensão foi decidido colocar todo o código escrito dentro de uma pasta a qual lhe demos o nome de **src**. Dentro desta pasta encontra-se todo o código escrito para este trabalho, este código está dividido em 6 ficheiros, 3 *.h* e 3 *.c*. Estes ficheiros são o servidor (*argusd.c* e o respectivo *argusd.h*) o cliente (*argus.c* e o respectivo *argus.h*) e o ficheiro que contem as funções comuns entre o servidor e o cliente de forma a evitar código repetido (*common.c* e o respectivo *common.h*). E para a sua utilização e maior facilidade de utilização criamos uma *Makefile* que iremos explicar mais á frente como funciona. A comunicação entre o servidor e o cliente passa apenas pelos pipes com nome **server_to_client_fifo**, em que apenas o servidor escreve e o cliente lê, e o pipe com nome **client_to_server_fifo** que apenas o cliente escreve e o servidor lê.

3.1 Common

De forma a evitar código repetido no cliente e no servidor foi decidido criar o seguinte pacote para posteriormente ser importado e utilizado pelos mesmos.

```
1  #ifndef __COMMON__
2  #define __COMMON__
3
4  #include <unistd.h>
5  #include <sys/wait.h>
6  #include <stdio.h>
7  #include <fcntl.h>
8  #include <stdlib.h>
9  #include <string.h>
10 #include <sys/types.h>
11 #include <time.h>
12 #include <ctype.h>
13 #include <sys/stat.h>
14 #include <signal.h>
15
16 #define READ    0
17 #define WRITE   1
18 #define STD_IN  0
19 #define STD_OUT 1
20 #define STD_ERR 2
21
22 ssize_t readln_v1(int fd, char* line, size_t size);
23
24 //Compara 2 strings usando o strcmp
25 int equals(char* str1, char* str2);
26 // Verifica se o comando tem argumentos inteiros
27 int command_with_integer_args(char* command);
28 // Verifica se o comando tem argumentos
29 int command_with_args(char* command);
30
31 #endif
```

Figura 3: common.h

3.1.1 Macros

Para uma melhor compreensão do código foram criadas as seguintes que são utilizadas pelo servidor e pelo cliente:

- **READ** que tem o valor 0, que é associado ao descritor de ficheiro de leitura.
- **WRITE** que tem o valor 1, que é associado ao descritor de ficheiro de escrita.
- **STD_IN** que tem o valor 0, que é associado ao descritor de ficheiro padrão de entrada.
- **STD_OUT** que tem o valor 1, que é associado ao descritor de ficheiro padrão de saída.
- **STD_ERR** que tem o valor 2, que é associado ao descritor de ficheiro padrão de erro.

3.1.2 Funções

- **readln_v1** Lê os valores de um ficheiro e armazena o resultado em variáveis. Depois disso passa para a próxima linha do ficheiro.
- **equals** Compara 2 strings utilizando o *strcmp*.
- **command_with_integer_args** Serve para verificar se o comando recebido é um dos que precisa de inteiros (tempo-inactividade, tempo-execucao, terminar ou output) utilizando a função *equals*.
- **command_with_args** Verifica se o comando recebido é executar utilizando a função *equals*.

3.2 Cliente

```
1  #ifndef __ARGUS__
2  #define __ARGUS__
3
4  // Verifica se o comando tem argumentos inteiros
5  int option_with_integer_args(char* option);
6  // Verifica se o comando tem argumentos
7  int option_with_args(char* option);
8  // Vai buscar comando a partir da opção
9  char* get_command(char* option);
10
11 // Verifica se um comando é valido
12 int valid_command(char* command);
13
14 // Escreve no stdout "argus$ "
15 void print_prompt();
16 // Imprime as respostas que vêm do servidor
17 void print_response(int prompt);
18
19 #endif
```

Figura 4: argus.h

O *argus* funciona como cliente do servidor. Como explicado acima o cliente apenas usa os pipes com nome **server_to_client_fifo** para lêr e o **client_to_server_fifo** para escrever. Este programa tem 2 modos de utilização caso não tenha argumentos funciona como um terminal, mas caso receba argumentos (-i, -m, -e, -l, -t, -r, -h, -o) ele verifica se existe o comando pretendido e este é válido enviando ao servidor o comando pretendido. Caso esteja em modo consola pode utilizar "quit" ou "exit" para terminar a execução do cliente.

3.3 Servidor

O servidor é o responsável por grande parte das operações. Para a implementação deste programa, tivemos de ter em conta no desenvolvimento da arquitetura, a concorrência existente entre os processos que nele ocorrem, sendo estes processos as múltiplas tarefas em execução. Para resolver este processo de concorrência procedemos à implementação de um FIFO que receberá todas as tarefas a serem executadas no servidor, tratando de uma de cada vez.

3.3.1 Macros

Para uma melhor compreensão do código foram criadas as seguintes que são utilizadas apenas pelo servidor:

- **DEFAULT_MAX_EXECUTION_TIME** = 5,
- **DEFAULT_MAX_INACTIVITY_TIME** = 5,
- **CONCLUDED_EXIT_STATUS** = 10,
- **INACTIVITY_TIMEOUT_EXIT_STATUS** = 11,
- **EXECUTION_TIMEOUT_EXIT_STATUS** = 12,
- **TERMINATED_EXIT_STATUS** = 13.

3.3.2 Variáveis globais

Partiu-se do pressuposto que não se quis utilizar estruturas e baseamos toda a manutenção do servidor em ficheiros. Utilizou-se variáveis globais porque se tem muitas funções a utilizarem estas variáveis e não fazia muito sentido passar andar com estas, passando como argumentos. Apenas o servidor contém variáveis globais e estas são:

- **current_task_id**: Serve para identificar a tarefa actual.
- **max_execution_time**: Indica o limite de tempo que cada tarefa tem para execução.
- **max_inactivity_time**: Indica o limite de tempo máximo que uma tarefa pode demorar a executar.

```
int current_task_id    = 0;
int max_execution_time = DEFAULT_MAX_EXECUTION_TIME;
int max_inactivity_time = DEFAULT_MAX_INACTIVITY_TIME;
```

Figura 5: Variáveis globais

3.3.3 Ficheiros

Assim sendo, temos três ficheiros para as variáveis globais:

- **current_task_id:** Só é criada quando se cria uma tarefa.
- **max_execution_time:** Define o tempo máximo (em segundos) de execução de uma tarefa.
- **max_inactivity_time:** Define o tempo máximo (em segundos) de inatividade de comunicação.

Dois ficheiros para tarefas em execução:

- **task_info_ID:** Contém a informação necessária da tarefa em execução. Três linhas com o PID da tarefa a ser executada, tarefa (por exemplo, cat | wc) e a data de criação da tarefa.
- **task_log_ID:** Corresponde ao log da tarefa em execução.

Dois ficheiros de log:

- **log:** Contém o output sequencial de todas as tarefas terminadas (este pode ser vazio).
- **log.idx:** Contém o índice do log. Cada linha tem o ID, start_byte e count_bytes. O ID é o identificador da tarefa executada, o start_byte é o byte em que começa o log da tarefa e o count_bytes é o tamanho do output.

3.3.4 Modo de Funcionamento

O servidor é executado e executa a função *cleanup_and_start_server* que vai actualizar as variáveis globais e limpar os ficheiros criados caso estes existam. Depois entra num ciclo infinito em que fica a lêr do pipe com nome **client_to_server_fifo** e a executar os comandos recebidos do cliente pelo pipe através da função *execute_command*. Esta função identifica o comando e executa as tarefas correspondentes.

3.4 Makefile

```
1 all:
2     gcc -Wall src/common.c src/argusd.c -o argusd
3     gcc -Wall src/common.c src/argus.c -o argus
4
5 run:
6     ./argusd &>/dev/null &
7     ./argus
8
9 c:
10    ./argus
11
12 s:
13    ./argusd
14
15 clean:
16    rm -f argus
17    rm -f argusd
18    rm -f client_to_server_fifo
19    rm -f server_to_client_fifo
20    rm -f argus_history
21    rm -f current_task_id
22    rm -f max_execution_time
23    rm -f max_inactivity_time
24    rm -f log
25    rm -f log.idx
26
27 power:
28    make clean && make
```

Figura 6: Makefile

Esta Makefile foi criada por nós permite-nos tratar de uma maneira mais clara toda a aplicação, tal como compilar, executar entre outras coisas.

- **all:** compila os programas.
- **run:** corre um servidor em background ignorando o output e um cliente na mesma janela.
- **c:** corre um cliente.
- **s:** corre um servidor.
- **clean:** remove todos os executáveis.
- **power:** limpa todo o histórico e recompila.

4 Conclusão

A realização deste trabalho pratico apesar das dificuldades sentidas ficou muito mais fácil após a resolução dos guiões das aulas praticas uma vez que o servidor apresenta funções como

my_bash, *my_popen* e até mesmo a *my_system* que foram recicladas de forma a nos ajudarem na realização deste trabalho de modo a conseguirmos ultrapassar todos os requisitos impostos pela equipa docente de uma forma gratificante.

Em suma, consideramos assim, que foi feito um bom trabalho. Cumprimos todos os requisitos propostos pela equipa docente e sentimos que foi um trabalho que nos enriqueceu bastante no que diz respeito a programação concorrente e também a conhecimentos do ambiente UNIX.