

Sistemas Operativos

Mestrado Integrado em Engenharia Informática
Controlo e Monitorização de Processos e Comunicação

Grupo 102



Inês Bastos

José Nuno

Maria Sofia

13 de junho de 2020

Inês Sofia Paiva Bastos, a89522

José Nuno Baptista Martins, a90122

Maria Sofia Marques, a87963

Conteúdo

1- Introdução;

2- Makefile;

2.1-Comandos disponíveis

3- Funcionalidades do programa

3.1-Tempo de Inatividade;

3.1.1-Implementação;

3.2-Tempo de execução;

3.2.1-Implementação;

3.2.2-Exemplo de execução;

3.3-Executar;

3.3.1-Implementação;

3.3.2-Exemplo de execução;

3.4-Listagem de Tarefas em execução;

3.4.1-Implementação;

3.4.2-Exemplo de execução;

3.5-Terminar Tarefa;

3.5.1-Implementação;

3.5.2-Exemplo de execução;

3.6-Histórico;

3.6.1-Implementação;

3.6.2-Exemplo de execução;

3.7-Ajuda;

3.7.1-Implementação;

3.7.2-Exemplo de execução;

4- Comunicação Servidor/Cliente

4.1-Servidor(argusd)

4.2-Cliente(argus)

4.3 Relação Cliente e Servidor

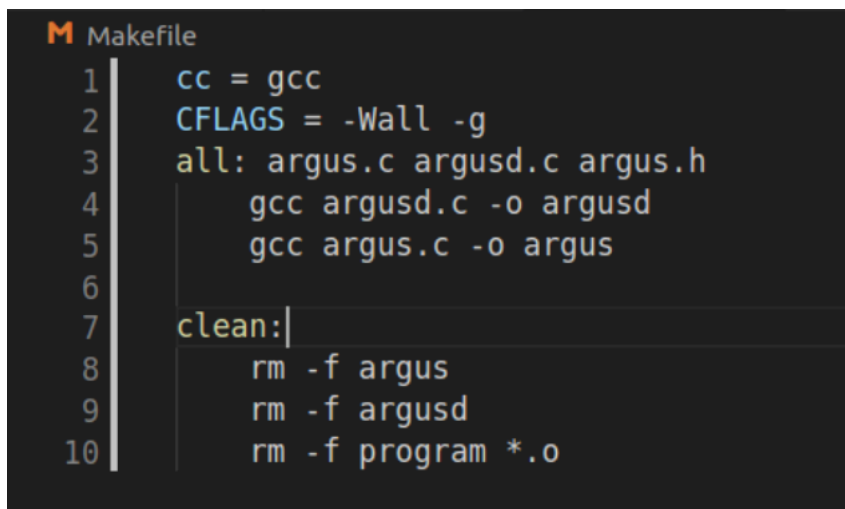
5- Conclusão

1- Introdução

Este relatório irá abordar a realização do projeto realizado no âmbito da Unidade Curricular Sistemas Operativos, que consiste na Controlo e Monitorização de Processos e Comunicação. Este trabalho foi realizado tendo como base todos os conteúdos lecionados nas aulas.

2- Makefile

A Makefile permite-nos compilar todo o nosso programa para que seja possível executá-lo.

A screenshot of a code editor showing a Makefile. The editor has a dark background with light-colored text. On the left, there is a vertical line with numbers 1 through 10. The text of the Makefile is as follows:

```
M Makefile
1  cc = gcc
2  CFLAGS = -Wall -g
3  all: argus.c argusd.c argus.h
4      gcc argusd.c -o argusd
5      gcc argus.c -o argus
6
7  clean:
8      rm -f argus
9      rm -f argusd
10     rm -f program *.o
```

2.1- Comandos disponíveis

Os comandos que temos para a Makefile são:

- make ou make all
 - Comando do Makefile para compilar o nosso programa.
- make clean
 - Comando para remover os executáveis criados no ato de compilação.

3- Funcionalidades do programa

3.1- Tempo de Inatividade;

3.1.1 Implementação;

Relativamente á implementação deste comando, surgiram algumas dúvidas, uma vez que, não arranjam uma solução para manter o tempo de execução e o tempo de inatividade a correrem sem problemas em simultâneo. Isto porque, a maneira como tratávamos desta funcionalidade utilizava também sinais de alarme, os quais também influenciavam os alarmes do comando do tempo de execução. Decidimos, portanto, colocar tudo o que fizemos relativamente ao tempo de inatividade em comentário para que o tempo de execução funcionasse da melhor forma.

A nossa ideia passava por utilizar um SIGALARM que tratava de um handler responsável por matar os pids relativos aos comandos que constituíam uma tarefa, quando os alarmes colocados antes dos pipes feitos na função executa disparavam. O tempo desses alarmes era dado por uma variável global chamada tempo de inatividade que é alterada aquando da chamada do comando -i (tempo - inatividade).

3.2-Tempo de execução;

3.2.1-Implementação;

A funcionalidade de tempo de execução tem como objetivo o término de um processo cujo tempo de execução excede o tempo dado como argumento.

Aquando da execução de algum comando, é atribuído ao alarme o tempo de execução máximo onde, ao chegar ao fim do mesmo, ativará o sinal de alarme, ao qual criamos um handler específico (timeout_handler). Assim sendo, nesse handler, iremos percorrer os nossos processos filhos em execução e matar os mesmos (através de um SIGKILL). No fim, mataremos também o pid da tarefa correspondente.

3.2.2-Exemplo de execução;

```
NO SIGCHLD 7401
EXIT 3
PID 7423
Added pid 7424
terminou 7424
Interrompeu a tarefa com pid 7423
NO SIGCHLD 7423
EXIT 0
PID 7430
Added pid 7432
  PID TTY          TIME CMD
  7373 pts/1        00:00:00 server
  7430 pts/1        00:00:00 server
  7432 pts/1        00:00:00 ps
19064 pts/1        00:00:01 bash
```

```
tempo-execucao 3
-e sleep 30
-e ps
-h
```

3.3-Executar;

3.3.1-Implementação;

A funcionalidade de executar permite-nos executar as tarefas passadas pela linha de comando. Esta chama a função executa e passa como argumento a tarefa a executar.

Dentro desta função executa, se a tarefa só tiver um comando, é criado um fork que utiliza a função executa_comando para executar o respetivo comando. Se a tarefa possuir diversos comandos ligados por pipes, esta função criará pipes anónimos que comunicarão entre si. O número de pipes é equivalente ao número de comandos -1. Dentro desses pipes são também criados forks responsáveis pela chamada da função executa_comando.

3.3.2-Exemplo de execução;

```
mkfifo: File exists
fifo is open.
PID 7378
Added pid 7379
PID 7384
Added pid 7385
  PID TTY          TIME CMD
 7373 pts/1    00:00:00 server
 7378 pts/1    00:00:00 server
 7379 pts/1    00:00:00 sleep
 7384 pts/1    00:00:00 server
 7385 pts/1    00:00:00 ps
19064 pts/1    00:00:01 bash
A tarefa 2 terminou
NO SIGCHLD 7384
EXIT 2
█
```

```
open is done
-e sleep 100
-e ps
█
```

3.4-Listagem de Tarefas em execução;

3.4.1-Implementação;

Esta funcionalidade é usada para nos listar todos os comandos que estão em execução nesse momento.

Isto é-nos possível obter através de um comando kill (pid,0) que nos permite verificar se um dado processo está em execução. Se o mesmo acontecer (ou seja, se o resultado retornado for igual a zero), então, iremos listar esse processo.

3.4.2-Exemplo de execução;

```
fifo is open.
PID 7559
Added pid 7560
PID 7563
Added pid 7564
PID 7566
Added pid 7567
Ajuda.txt      clienteT.c  fifoT.c      logs.txt      server
Algoritmo.c    fifo        Historico.c  log.txt       servidorT.c
client         fifo1       Historico.txt rascunhoclient.c
A tarefa 3 terminou
NO SIGCHLD 7566
EXIT 3
PID 7569
Added pid 7570
#1: sleep 30
#4: sleep 20
A tarefa 4 terminou
NO SIGCHLD 7569
EXIT 4
A tarefa 1 terminou
NO SIGCHLD 7559
EXIT 1
A tarefa 2 terminou
NO SIGCHLD 7563
EXIT 2
#1: Concluida: sleep 30
#2: Concluida: sleep 30
#3: Concluida: ls
#4: Concluida: sleep 20
█
```

```
open is done
-e sleep 30
-e sleep 30
-e ls
-e sleep 20
listar
-r
█
```

3.5-Terminar Tarefa;

3.5.1-Implementação;

Utilizamos esta funcionalidade com o objetivo de terminar a tarefa que nos é dada como argumento. Tendo em conta que temos os pids das nossas tarefas guardadas numa struct (uma struct “tarefa” para cada tarefa) e um array dessas structs, vamos buscar essa tarefa ao índice passado como argumento. Daí, e com o auxílio de um SIGQUIT, ao qual criamos um handler específico, iremos matar todos os filhos dessa mesma tarefa.

3.5.2-Exemplo de execução;

```
mkfifo: File exists
fifo is open.
PID 7622
Added pid 7623
PID 7624
Added pid 7625
A tarefa 1 com pid 7622 foi interrompida
terminou 7623
NO SIGCHLD 7622
EXIT 0
PID 7632
Added pid 7633
A tarefa 3 terminou
NO SIGCHLD 7632
EXIT 3
PID 7634
Added pid 7635
A tarefa 4 terminou
NO SIGCHLD 7634
```

```
open is done
-e sleep 20
-e sleep 30
terminar 1
-e sleep 2
-e sleep 2
-r
-r
```

3.6-Histórico;

3.6.1-Implementação;

O Histórico permite-nos listar as tarefas que foram concluídas. Assim sendo, e utilizando mais uma vez o comando referido em cima (comando kill(pid,0)), e lembrando também que temos um array de estruturas que guarda os pids das tarefas, vamos então percorrer esse array e verificar se as tarefas estão concluídas. Aquelas cujo comando retorna diferente de 0 são aquelas que já foram concluídas e, portanto, serão essas que iremos verificar o estado e o nome da tarefa e fazer a sua devida impressão no ecrã.

No entanto, é necessário referir que, os estados das tarefas estão a ser guardados numa estrutura Tarefa que tanto guarda o estado de uma tarefa como o pid de uma tarefa.

Na execução deste tópico, deparamo-nos, por vezes, com um obstáculo que acabamos por não conseguir resolver. Este “bug” não acontece todas as vezes que executamos o histórico, pelo que, muitas das vezes em que tentamos recriar o problema, este não surgiu, tornando a correção do mesmo uma tarefa difícil. Acreditamos que, este problema se deve a um mau armazenamento da memória.

3.6.2-Exemplo de execução;

```
mkfifo: File exists
fifo is open.
PID 7622
Added pid 7623
PID 7624
Added pid 7625
A tarefa 1 com pid 7622 foi interrompida
terminou 7623
NO SIGCHLD 7622
EXIT 0
PID 7632
Added pid 7633
A tarefa 3 terminou
NO SIGCHLD 7632
EXIT 3
PID 7634
Added pid 7635
A tarefa 4 terminou
NO SIGCHLD 7634
EXIT 4
#1: Max Execucao: sleep 20
#3: Concluida: sleep 2
#4: Concluida: sleep 2
█

open is done
-e sleep 20
-e sleep 30
terminar 1
-e sleep 2
-e sleep 2
-r
█
```

3.7 Ajuda

3.7.1 Implementação

Usamos esta funcionalidade com o objetivo de permitir obter informação relativamente às funcionalidades do programa. Para isso, apenas utilizamos um write responsável por apresentar a ajuda necessária para a utilização do mesmo.

3.7.2 Exemplo de execução

```
mkfifo: File exists
fifo is open.
>tempo-inatividade segs
>tempo-execucao segs
>executar p1 | p2 | ... | pn
>listar
>terminar 'numero de tarefa'
>historico
>ajuda
█

open is done
-h
█
```

4.1 Servidor (argusd)

O servidor, quando é iniciado fica a correr em background. Fica, assim, à espera dos comandos do utilizador os quais são passados e, consequentemente, este irá executar as tarefas em programação paralela. Depois é responsável por enviar o output dessas tarefas (caso as tenham), para o cliente (argus).

4.2 Cliente (argus)

O Cliente oferece uma interface com o utilizador via linha de comando, que permite suportar todas as funcionalidades implementadas. O utilizador pode agir sobre o servidor através dos argumentos passados na linha de comandos do cliente, ou, no caso do cliente ser invocado sem argumentos, através da Shell.

4.3 Relação Cliente e Servidor

Estes dois elementos comunicam entre si através de dois pipes com nome, fifo e fifo1. O fifo corresponde á passagem das tarefas implementadas pelo utilizador, do cliente para o servidor de forma a que este último as possa executar. O fifo1 corresponde á passagem do output das tarefas executadas, do servidor para o cliente.

5. Conclusão

Este trabalho gerou algumas dificuldades para o grupo, apesar de ser um trabalho que exigia a matéria abordada na UC, exigiu também que usássemos o nosso espírito de autonomia procurando assim diferentes soluções para as diversas barreiras encontradas durante a realização do mesmo.

Dentro das dificuldades com que nos encontramos, a organizada comunicação entre cliente/servidor e a aplicação da possibilidade de concorrência de processos, em conciliação com a preocupação do aproximar da data limite foram sem margem para dúvida as maiores dificuldades que o grupo teve de encarar e ultrapassar. Reconhecemos também que com mais tempo teríamos conseguido aplicar mais funcionalidades ao trabalho, como as restantes funcionalidades opcionais propostas pelo professor docente.