Sistemas Operativos

Licenciatura em Ciências da Computação

Universidade do Minho

Controlo e Monitorização de Processos e Comunicação

Hugo Sousa, A76257 Matias Capitão, A82726
Rafael Antunes, A77457

Junho 2020

Resumo

Este é o relatório referente ao trabalho prático da cadeira de **Sistemas Operativos**, leccionada no 2º semestre dos cursos de Licenciatura em Ciências da Computação (LCC) e Mestrado Integrado em Engenharia Informática (MIEI).
Neste trabalho tentaremos implementar um serviço de monitorização de execução e de comunicação entre processos utilizando a matéria que fomos aprendendo ao longo do semestre.

Conteúdo

Capítulo 1

Introdução

Como já referido, pretende-se que implementemos um serviço de monitorização de execução e de comunicação entre processos. Este serviço permitirá a um utilizador a submissão de sucessivas tarefas, cada uma delas sendo uma sequência de *pipes* anónimos. Em termos de interface, o utilizador poderá interagir através da linha de comandos ou a partir da *shell*.

Capítulo 2

Análise e Especificação

2.1 Descrição do problema

Para criar este serviço teremos em conta o ponto de vista do Cliente e do Servidor.
No caso do Servidor, teremos de manter em memória as informações relevantes para suportar as funcionalidades que pretendemos. Em relação ao Cliente, teremos uma interface via linha de comando que permita suportar as funcionalidades descritas brevemente neste relatório.

2.2 Especificação de Requisitos

2.2.1 Requisitos

Para a realização deste trabalho seguimos as directrizes da UC. Como tal, iremos trabalhar com ambiente Linux e utilizar a linguagem de programação
,
Para uma boa conduta na realização é necessário ter conhecimentos prévios e bem estruturados de alguns dos seguintes temas:

- Gestão de processos;
  - O que é um processo;
  - Operações sobre processos.
  - Estados dos processos;
  - Atributos dos processos;
  - Arquitectura do Gestor de processos;
  - Utilização de *System Calls*;
  - Implementação de *signals*;
  - Algoritmos de Escalonamento.

- Gestão de memória;
  - Endereçamento e Espaço de Endereçamento;
  - Sistemas de Endereçamento Real;
  - Sistemas de Endereçamento Virtual.
- Noções de*Makefiles*;
- Noções de *header files*.

Um Sistema Operativo, independentemente do seu tipo, é algo bastante complexo, sendo necessária a procura de mais informação além da acima mencionada. De referir também que nos foi pedido pelos docentes para não utilizar a função *printf();* a não ser que seja usada para *debugging*.

2.2.2 Funcionalidades

Sempre que for necessário representar 'tempos', utilizaremos os 'segundos' como medição. Este serviço de monitorização deverá suportar as seguintes funcionalidades:

- Definir um tempo máximo de inactividade de comunicação num *pipe* anónimo (***-i*** *num* na linha de comando);
- Definir um tempo máximo de execução de uma tarefa (***-m*** *num* na linha de comando);
- Executar uma tarefa (***-e*** *"*p_1 | p_2 | ... | p_n*"* na linha de comando);
- Listar as tarefas em execução (***-l*** na linha de comando);
- Terminar uma tarefa em execução (***-t*** *num* na linha de comando);
- Listar registo histórico das tarefas terminadas (***-r*** na linha de comando);
- Disponibilizar ajuda à sua utilização (***-h*** na linha de comando).
- Tentaremos ainda adicionar a funcionalidade de consultar o *standard output* produzido por uma tarefa já executada (***-o*** na linha de comando).

Capítulo 3

Concepção da Resolução

Como descrito no enunciado foi necessário criar um Servidor e um Cliente.

3.1 Estrutura de Dados

3.1.1 Tasks

Temos duas estruturas para organizar as tarefas que vamos executar. Uma chamar-se-á "task"que será composta por um array de caratéres para o nome da tarefa, um identificador, um estado que poderá ser "FREE","ACTIVE", "ALIVE"ou "DEAD"e outra variável que será relativa á forma como terminou.

A outra será chamada "Tasks"e será composta por um array da struct task, um inteiro para o tempo máximo de execução de uma tarefa, outro inteiro para o tempo máximo de execução de uma pipe anónima e ainda um inteiro para o tamanho.

```
struct task{
char *name;
int id;
int state;
int pid;
int c;
};

typedef struct ts{
struct task *tasks;
int taskTime;
int pipeTime;
int size;
}Tasks;
```

3.1.2 Cliente

O cliente começa por receber um dos comandos referidos acima como argumento. Este comando vai estar na forma -c ..."onde c poderá ser e, i, r, entre outros. Para avaliarmos o que fazer criamos a função "handle_cmd_line"que vai transformar os argumentos num só array de caratéres ao qual podemos chamar "req"que será enviado com a ajuda da nossa função "send_reply"para o Servidor. Esta função começa por abrir o FIFO "request"criado pelo Servidor e depois, com a system call **write**, escrevemos "req"para o FIFO "request".

3.1.3 Servidor

O Servidor começa por criar e abrir dois FIFOs, um que vai ter a flag O_RDONLY, uma vez que será utilizado apenas para ler o que recebemos do Cliente e o outro servirá para enviar respostas para o Cliente, ou seja, terá a flag O_WRONLY. Posto isto, vamos criar uma struct Tasks de tamanho 10 e entrar num ciclo infinito que irá receber pedidos do Cliente. Neste ciclo, abrimos a pipe, com a ajuda da função read_ln que criamos para ler linha a linha vamos ler os comandos para um buffer e avançamos para a função handle_client_request que vai avaliar o conteúdo do buffer, isto é, os comandos enviados pelo cliente.

Na função handle_client_request começamos por utilizar a função strtok da biblioteca string.h para separar o array de carateres recebido como argumento por espaços. Esta função observa o primeiro espaço e guarda numa variável token o resultado. Este resultado vai ser o comando que temos de executar, portanto, e com mais uma função da biblioteca string.h chamada strcmp, vamos comparar esta entrada com os comandos possíveis e chamar a função para cada caso sempre que strcmp for verdadeira.

Casos para o token:

- -i", -m"ou -t": vamos fazer mais uma vez strtok, comvertemos o valor do caractere para inteiro a definimos na nossa estrutura o nosso tempo máximo de execução de uma tarefa ou o nosso tempo máximo de execução de uma pipe ou procuramos na nossa estrutura as tarefas que estejam ALIVE, terminamo-las e colocamos a sua flag a DEAD, respetivamente.

- -l"ou -r": vamos procurar na nossa estrutura as tarefas que têm o estado "ALIVE"e imprimi-las ou vamos procurar as tarefas que estejam "DEAD"e imprimi-las, respetivamente.

- -h": será imprimido o menu de ajuda.

- -o": vamos utilizar mais uma vez a função strtok que nos irá devolver o id da tarefa da qual queremos observar o output obtido. Este id é convertido para inteiro e enviado para a função show_output onde vamos abrir os dois logs e imprimir o resultado lá escrito.

- -e": Vamos para a função execute_tasks que recebe como argumentos as Tasks e a tarefa a executar. Aqui, começamos por inicializar uma task, e depois criamos um processo filho onde chamaremos a função parse_execute. O processo pai espera que o filho termine e coloca o estado da tarefa a "DEAD". Na função parse_execute são contadas as pipes que temos de criar e os comandos são guardados num array. Estes parâmetros são passados como argumento na função "execute". Nesta função começamos por criar um array com tamanho igual ao número de pipes mais um, de descritores de ficheiro. Depois para cada comando vamos criar um processo filho, onde começamos por organizar os o comando e as opçoes se aplicavel, depois, se for o primeiro comando, redireccionamos o stdout para o descritor de escrita do pipe respectivo , se for outro qualquer, redirecionamos o stdin para o descritor de leitura e o stdout para o descritor de escrita. Por fim, fechamos as pipes e corremos o comando. No processo pai vamos fechar todas as copias dos pipes e caso seja o ultimos iremos ler o resultado e escrever o resultdado nos ficheiros.

Capítulo 4

Codificação e Testes

4.1 Decisões

4.1.1 Execute

Uma das funcionalidades principais é a de executar uma tarefa, esta funcionalidade é feita da seguinte forma:

- É inicializada uma ”task”dentro da estrutura *Tasks*;
- É criado um processo filho que irá chamar a função *parse_execute(ts, id, cmd)* que irá tratar do comando e excutar a tarefa própriamente dita;
- O processo pai invoca a função waitpid com a flag WNOHANG o que faz com que retorne imediatamente se não existir nenhum filho terminado, assim o pai vai continuar a correr;
- Depois de feito o parse do comando, obtido o número de pipes etc.. o filho irá executar o comando, da forma já descrita em cima;
- O seguinte diagrama exemplifica o processo

```
filho    filho    filho   pai
cmd1  |  cmd2  |  cmd3  |
    pipe1 -> pipe2 -> pipe3
```

4.1.2 Output

Na implementação da opção de mostrar o output de uma tarefa foram tomadas as seguintes decisões

- É verificada se a tarefa já acabou;

- Se sim são abertos os ficheiros *log* e *log.idx* para leitura
- Vamos procurar pela informação do tamanho do números de bytes do output, isto é feito com o id da tarefa.
- Enquanto vamos procurando esta informação vamos somando o valor dos dos tamanhos das tarefas que foram escritas anteriormente, este valor será o *offset* para a leitura do ficheiro log.
- Depois destas duas informações obtidas usamos *lseek(log, offset, SEEK_SET)* para deslocar o inicio da leitura para a posição correto
- São lidos bytes do output e é enviado o resultado para o FIFO "reply"para ser lido pelo cliente

4.1.3 Terminar uma tarefa

Para terminar uma tarefa recebemos um id foi feito o seguinte

- Verificamos se a tarefa ainda está viva;
- Se sim mudamos o seu estado para DEAD;
- Mudamos a flag que indica a forma de terminação também, para TERM;
- É enviado ao processo um sinal com SIGKILL, (*kill(ts->tasks[id-1].pid, SIGKILL)*);

4.2 Problemas de Implementação

O maior problema que enfrentamos na resolução do trabalho foi na parte dos sinais. Aquilo que tentamos fazer foi acionar um alarme com os tempos definidos pelo utilizador depois da tarefa estar em execução, no entanto, apesar de estar a acionar o alarme para tarefas com tempos acima do estipulado não conseguimos arranjar forma de alterar, nem o estado da tarefa para "DEAD", nem a forma como terminou para "max-execucao"e, portanto, apesar de fazermos kill do processo, quando pedimos a listagem de tarefas em execução e o histórico de tarefas concluídas, a tarefa vai aparecer sempre na listagem de tarefas em execução. Uma solução que tentamos aplicar foi tornar a nossa estrutura Tasks pública, no entanto, o resultado nunca foi aquele que gostaríamos e, assim sendo, colocamos apenas o aviso no terminal que, de facto, a tarefa excedeu o tempo de execução.

Outra situação que nos desagradou, foi o facto de algumas respostas enviadas pelo servidor para o cliente aparecerem desformatadas.

4.3 Resultados

```
utilizador@utilizador-N550JK:~/Desktop/Trabalhof$ ./argus
> executar cut -f7 -d: /etc/passwd | uniq | wc -l
nova tarefa #1
> executar cat | cat
nova tarefa #2
> executar ls -l
nova tarefa #3
> executar ls -l | sort | wc -l
nova tarefa #4
> historico
#1, concluida: cut -f7 -d: /etc/passwd | uniq | wc -l
#3, concluida: ls -l
#4, concluida: ls -l | sort | wc -l
> listar
#2, cat | cat
> terminar 2
> listar
> historico
#1, concluida: cut -f7 -d: /etc/passwd | uniq | wc -l
#2, terminada: cat | cat
#3, concluida: ls -l
#4, concluida: ls -l | sort | wc -l
> output 3
total 228
-rwxrwxr-x 1 utilizador utilizador 17640 jun 15 18:08 argus
-rw-rw-r-- 1 utilizador utilizador   873 jun 15 18:07 argus.c
-rw-rw-r-- 1 utilizador utilizador  3232 jun 15 18:08 argus.o
-rw-rw-r-- 1 utilizador utilizador  2802 jun 15 18:07 clientParser.c
-rw-rw-r-- 1 utilizador utilizador   314 jun 15 18:07 clientParser.h
-rw-rw-r-- 1 utilizador utilizador  6720 jun 15 18:08 clientParser.o
-rw-rw-r-- 1 utilizador utilizador 88484 jun 15 18:07 enunciado.pdf
-rw-rw-r-- 1 utilizador utilizador   122 jun 15 18:07 input
-rw-rw-r-- 1 utilizador utilizador    90 jun 15 18:07 input2
-rwx------ 1 utilizador utilizador     3 jun 15 18:08 log
-rwx------ 1 utilizador utilizador     4 jun 15 18:08 log.idx
-rw-rw-r-- 1 utilizador utilizador   377 jun 15 18:07 Makefile
prwx------ 1 utilizador utilizador     0 jun 15 18:08 reply
prwx------ 1 utilizador utilizador     0 jun 15 18:08 request
-rw-rw-r-- 1 utilizador utilizador  8167 jun 15 18:07 requestHandler.c
-rw-rw-r-- 1 utilizador utilizador   352 jun 15 18:07 requestHandler.h
-rw-rw-r-- 1 utilizador utilizador 16152 jun 15 18:08 requestHandler.o
-rwxrwxr-x 1 utilizador utilizador 26936 jun 15 18:08 server
-rw-rw-r-- 1 utilizador utilizador   547 jun 15 18:07 server.c
-rw-rw-r-- 1 utilizador utilizador  2608 jun 15 18:08 server.o
-rw-rw-r-- 1 utilizador utilizador   809 jun 15 18:07 task.c
-rw-rw-r-- 1 utilizador utilizador   651 jun 15 18:07 task.h
-rw-rw-r-- 1 utilizador utilizador  2376 jun 15 18:08 task.o
> output 1
13
>
```

Figura 4.1: Ficheiro do tipo .3d para uma esfera

```
utllizador@utillizador-N550JK:~/Desktop/Trabalhof$ ./server
PASSOU O TEMPO
```

```
utllizador@utillizador-N550JK:~/Desktop/Trabalhof$ ./argus
> ajuda
Execute a task -> (OPTION) executar p1 | p2 ... | pn
Define pipe-time -> (OPTION) tempo-inactividade <seconds>
Define task-time -> (OPTION) tempo-execucao <seconds>
List executing tasks -> (OPTION) listar
List finished tasks -> (OPTION) historico
End a running task -> (OPTION) terminar <task id>
Show output -> (OPTION) output <task id>
Predefined file -> ficheiro <nome ficheiro>
Help -> (OPTION) ajuda
> tempo-inactividade 3
> tempo-execucao 3
> executar cat | cat
nova tarefa #1
>
```

Figura 4.2: Ficheiro do tipo .3d para uma esfera

Capítulo 5

Bibliografia

"Sistemas Operativos", Editora FCA - José Alves Marques, Paulo Ferreira, Carlos Ribeiro, Luís Veiga, Rodrigo Rodrigues

Capítulo 6

Conclusão

Em suma, com este trabalho conseguimos compreender melhor a utilização de FIFO's para a comunicação entre Cliente e Servidor. Foi necessário, também, aprofundar os conhecimentos de pipes anónimos e dups, uma vez que, era algo essencial para o comando de execução de tarefas. De resto, acreditamos que fomos de encontro ao pedido e lamentamos apenas não termos conseguido concluir os sinais corretamente.